Fast Documentation

Release 3.4

/FAST/MU-01/V3.4

Jun 29, 2022

CONTENTS

CHAPTER ONE

PREAMBLE

Fast is a common/front module for all Fast series solvers.

Fast is only available for use with the pyTree interface. You must import the module:

```
import Fast.PyTree as Fast
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Actions

Fast.PyTree.setNum2Base(a, num)	Set numeric data dictionary in bases.
Fast.PyTree.setNum2Zones(a, num)	Set numeric data dictionary in zones.
Fast.PyTree.load([fileName, fileNameC, ...])	Load tree and connectivity tree.
Fast.PyTree.save(t[, fileName, split, NP, ...])	Save tree and connectivity tree.
Fast.PyTree.loadFile([fileName, split, ...])	Load tree and connectivity tree.
Fast.PyTree.saveFile(t[, fileName, split, ...])	Save tree in file.

CHAPTER

THREE

CONTENTS

3.1 Actions

`Fast.PyTree.`**`setNum2Base`**(*a*, *numb*)

Set the numb dictionary to bases. Exists also as in place version (_setNum2Base) that modifies a and returns None.

Parameters

- **a** (`Base, pyTree`) – input data
- **numb** (`dictionary`) – numerics for base

the **keys** of **numb** dictionary are:

- **'temporal_scheme'**: possible values are
  - 'explicit' (RK3 scheme, see p49 http://publications.onera.fr/exl-php/util/documents/accede_document.php)
  - 'implicit' (BDF2 or BDF1 if local time stepping)
  - 'implicit_local' (see p107 http://publications.onera.fr/exl-php/docs/ILS_DOC/227155/DOC356618_s1.pdf)
  - default value is 'implicit'
- **'ss_iteration'**:
  - Newton Iterations for implicit schemes
  - default value is 30
- **'modulo_verif'**:
  - period of computation for: cfl (RK3 or BDF2), newton convergence (all temporal_scheme) and predictor estimation for 'implicit_local' scheme
  - default value is 200

Example of use:

- Set numerics to base (pyTree):

```
# - setNum2Base (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Fast.PyTree as Fast

a = C.newPyTree(['Base'])

numb = { 'temporal_scheme': 'explicit',
         'ss_iteration': 30,
         'modulo_verif':20 }

Fast._setNum2Base(a, numb)
Internal.printTree(a)
```

`Fast.PyTree.`**`setNum2Zones`**(*a*, *numz*)

Set the numz dictionary to zones. Exists also as in place version (_setNum2Zones) that modifies a and returns None.

Parameters

- **a** (`Zone, Base, pyTree`) – input data
- **numz** (`dictionary`) – input data

the **keys** of **numz** dictionary are:

- **'scheme'**: possible values are
  - 'ausmpred' (see p49 https://tel.archives-ouvertes.fr/pastel-00834850/document)
  - 'senseur' (for DNS/LES, see p50 https://tel.archives-ouvertes.fr/pastel-00834850/document)
  - 'senseur_hyper' (for DNS/LES with shock, see `Lugrin scheme`)
  - 'roe'
  - default value is 'ausmpred'
- **'slope'**: possible values are
  - 'o3' (third order, see p50 https://tel.archives-ouvertes.fr/pastel-00834850/document)
  - 'o1' (first order, only valid for roe scheme)
  - 'minmod' (only valid for roe scheme)
  - default value is 'o3'

- **'senseurType'**: only valid for 'senseur' scheme. Possible values are
  - 0 : correction for speed only
  - 1 : correction for speed, density and pressure
- **'coef_hyper'**: only valid for 'senseur_hyper' scheme. Possible values are
  - [coeff1, coeff2] (see pdf M. Lugrin)
  - default value are [0.009, 0.015]
- **'motion'**: possible values are
  - 'none' (no motion)
  - 'rigid' (ALE without deformation see p47 https://tel.archives-ouvertes.fr/tel-01011273/document)
  - 'deformation' (ALE with deformation)
  - default value is 'none'
- **'time_step'**:
  - value of time step
  - default value is 1e-4
- **'time_step_nature'**:
  - 'global'
  - 'local'
  - default value is 'global'
- **'epsi_newton'**:
  - newton stopping criteria on Loo norm
  - default value is 0.1
- **'inj1_newton_tol'**:
  - Newton tolerence for BCinj1 inflow condition
  - default value is 1e-5
- **'inj1_newton_nit'**:
  - Newton Iteration for BCinj1 inflow condition
  - default value is 10
- **'psiroe'**:
  - Harten correction

  - default value is 0.1
- **'cfl'**:
  - usefull only if 'time_step_nature'='local'
  - default value is 1
- **'model'**: possible values are
  - 'Euler'
  - 'NSLaminar'
  - 'NSTurbulent'(only Spalart available)
  - 'LBMLaminar'
  - default value is 'Euler'
- **'prandtltb'**:
  - turbulent Prandtl number (only active for 'model'='NSTurbulent')
  - default value is 0.92
- **'ransmodel'**: possible values are
  - "SA" (Standard Spalart-Allmaras model)
  - "SA_comp" (SA with mixing layer compressible correction https://turbmodels.larc.nasa.gov/spalart.html)
  - default value is 'SA'
- **'DES'**: possible values are
  - "none" (SA computation)
  - "zdes1" (mode 1, https://link.springer.com/content/pdf/10.1007%2Fs00162-011-0240-z.pdf)
  - "zdes1_w" (mode 1 by Chauvet)
  - "zdes2" (mode 2)
  - "zdes2_w" (mode 2 by Chauvet)
  - "zdes3" (mode 3, see p118 https://tel.archives-ouvertes.fr/tel-01365361/document)
  - default value is 'none'
- **'DES_debug'**: possible values are
  - "none"
  - "active" (save delta and fd functions in the FlowSolution#Centers node)

  - default value is 'none'

- **'sgsmodel'**: possible values are
  - "Miles" (ViscosityEddy==LaminarViscosity)
  - "smsm" (Selective Mixed Scale model, Lenormand et al, (2000), LES of sub and supersonic channel flow at moderate Re. Int. J. Numer. Meth. Fluids, 32: 369–406)
  - default value is 'Miles'
- **'extract_res'**: possible values are
  - 0
  - 1 (save div(F_Euler-F_viscous) in the FlowSolution#Centers node)
  - 2 (save dqdt + div(F_Euler-F_viscous) in the FlowSolution#Centers node)
  - default value is 0
- **'source'**: possible values are
  - 0
  - 1 (read a source terme in the FlowSolution#Centers node. The conservative variables centers:Density_src, centers:MomentumX_src, centers:MomentumY_src, centers:MomentumZ_src and centers:EnergyStagnationDensity_src are used.)
  - default value is 0
- **'ratiom'**:
  - cut-off max of mut/mu
  - default value is 10000.

Example of use:

- Set numerics to zone (pyTree):

```
# - setNum2Zones (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Fast.PyTree as Fast
import Generator.PyTree as G

a = G.cart((0,0,0), (1,1,1), (10,10,10) )
t = C.newPyTree(['Base', a])

numz = { 'scheme': 'ausmpred',
         'time_step_nature': 'global',
```

(continues on next page)

(continued from previous page)

```
        'time_step': 0.001 }

Fast._setNum2Zones(t, numz)
Internal.printTree(t)
```

`Fast.PyTree.load`(*fileName='t.cgns'*, *fileNameC='tc.cgns'*, *fileNameS='tstat.cgns'*, *split='single'*)

Load computation tree t from file. Optionaly load tc (connectivity file) or tstat (statistics file). Returns also the graph as a dictionary {'graphID', 'graphIBC', 'procDict', 'procList'}. If split='single', a single file is loaded. If split='multiple', mulitple file format is loaded (restart/restart_0.cgns, . . .).

Parameters

- **a** (pyTree) – input data
- **fileName** (string) – name of file for save
- **split** (string) – 'single' or 'multiple'

Returns t, tc, ts, graph

Return type tuple

- Load pyTree (pyTree):

```
# - load (pyTree) -
import Fast.PyTree as Fast
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
import Connector.PyTree as X
import Converter.Mpi as Cmpi

# Create case
if Cmpi.rank == 0:
    a = G.cart((0,0,0),(1,1,1),(11,11,11))
    a = Cmpi.setProc(a,0)
    b = G.cart((10,0,0),(1,1,1),(11,11,11))
    b = Cmpi.setProc(b,1)
    t = C.newPyTree(['Base',a,b])

    t = X.connectMatch(t, dim=3)
    t = Internal.addGhostCells(t,t,2,adaptBCs=0)
    C.convertPyTree2File(t, 't.cgns')
    tc = C.node2Center(t)
```

(continues on next page)

(continued from previous page)

```
    tc = X.setInterpData(t, tc, loc='centers', storage='inverse')
    C.convertPyTree2File(tc, 'tc.cgns')
Cmpi.barrier()

t,tc,ts,graph = Fast.load('t.cgns', 'tc.cgns', split='single')

#print(t)
#print(tc)
#print(ts)
#print('procDict = ', graph['procDict'])
#print('graphID = ', graph['graphID'])
```

Fast.PyTree.**save**(*t*, *fileName='restart.cgns'*, *split='single'*, *temporal_scheme='implicit'*, *NP=0*)

Save computation tree t in file. If you run in mpi, NP must be the number of processor. If you run in seq mode, NP must be 0 or a negative number. If split='single', a single file is written. If split='multiple', different files are created depending on the proc number of each zone (restart/restart_0.cgns, . . .).

Parameters

- **a** (pyTree) – input data
- **fileName** (string) – name of file for save
- **split** (string) – 'single' or 'multiple'
- **NP** (int) – number of processors

Example of use:

- Save pyTree (pyTree):

```
# - save (pyTree) -
import Fast.PyTree as Fast
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Mpi as Cmpi

a = G.cart((0,0,0),(1,1,1),(11,11,11))
a = Cmpi.setProc(a,0)
b = G.cart((10,0,0),(1,1,1),(11,11,11))
b = Cmpi.setProc(b,1)
t = C.newPyTree(['Base',a,b])
Fast.save(t, fileName='t.cgns', split='single', NP=0)
```

`Fast.PyTree.`**`loadFile`**(*fileName='t.cgns', split='single', mpirun=False*)

Load tree from file. The tree must be already distributed (with 'proc' nodes). The file can be a single CGNS file ("t.cgns") or a splitted per processor CGNS file ("t/t_1.cgns", "t/t_2.cgns", ...)

If you run in sequential mode, mpirun must be false. The function returns a full tree.

If you run in mpi mode, mpirun must be true. The function returns a partial tree on each processor.

Parameters

- **fileName** (string) – name of file for load
- **split** (string) – 'single' or 'multiple'
- **mpirun** (boolean) – true if python is run with mpirun

Returns t

Return type CGNS tree

- Load single pyTree (pyTree):

```
# - loadFile (pyTree) -
import Fast.PyTree as Fast
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Mpi as Cmpi

# Save a file
a = G.cart((0,0,0),(1,1,1),(11,11,11))
a = Cmpi.setProc(a,0)
b = G.cart((10,0,0),(1,1,1),(11,11,11))
b = Cmpi.setProc(b,1)
t = C.newPyTree(['Base',a,b])
Fast.saveFile(t, fileName='t.cgns', split='single', mpirun=False)

# Load it back
Fast.loadFile(fileName='t.cgns', split='single', mpirun=False)
```

`Fast.PyTree.`**`saveFile`**(*fileName='t.cgns', split='single', mpirun=False*)

Save tree to file. The tree must be already distributed (with 'proc' nodes).

The file can be a single CGNS file ("t.cgns") or a splitted per processor CGNS file ("t/t_1.cgns", "t_2.cgns", ...)

If you run in seq mode, mpirun must be false.

If you run in mpi mode, mpirun must be true.

Parameters

- **fileName** (string) – name of file for load
- **split** (string) – 'single' or 'multiple'
- **mpirun** – true if python is run with mpirun

- Save single pyTree (pyTree):

```
# - saveFile (pyTree) -
import Fast.PyTree as Fast
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Mpi as Cmpi

a = G.cart((0,0,0),(1,1,1),(11,11,11))
a = Cmpi.setProc(a,0)
b = G.cart((10,0,0),(1,1,1),(11,11,11))
b = Cmpi.setProc(b,1)
t = C.newPyTree(['Base',a,b])
Fast.saveFile(t, fileName='t.cgns', split='single', mpirun=False)
```

CHAPTER FOUR

INDEX

- genindex
- modindex
- search