FastS Documentation

Release 3.4

/FAST/MU-02/V3.4

Jun 29, 2022

CONTENTS

CHAPTER

ONE

PREAMBLE

FastS is an efficient Navier-Stokes solver for use on structured grids.

FastS module works on CGNS/python trees (pyTrees) containing grid information (coordinates must be defined, boundary conditions and flow solution).

This module is only available for the pyTree interface:

```
import FastS.PyTree as FastS
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Computation Preparation

FastS.PyTree.warmup(t, tc[, graph, ...])	Perform necessary operations for the solver to run.
FastS.PyTree.createConvergenceHistory(t, nrec)	Create a node in tree to store convergence history.
FastS.PyTree.createStatNodes(t[, dir, vars, ...])	Create node in tree to store stats.
FastS.PyTree.createStressNodes(t[, BC, windows])	Create nodes to store stress data.

– Running computation

FastS.PyTree._compute(t, metrics, nitrun[, ...])	Compute a given number of iterations.
FastS.PyTree.displayTemporalCriteria(t, ...)	Display CFL and convergence information.

– Post

FastS.PyTree._computeStats(t, tmy, metrics)	Compute the space/time average of flowfields in tmy.
FastS.PyTree._computeStress(t, teff, metrics)	Compute efforts in teff.
FastS.PyTree._computeVariables(t, metrics, ...)	Compute given variables.
FastS.PyTree._computeGrad(t, metrics, varlist)	Compute gradient of fiven variables.
FastS.PyTree.extractConvergenceHistory(t, ...)	Extract residuals in an ascii file.

CHAPTER

THREE

CONTENTS

3.1 Preparation

FastS.PyTree.**warmup**(*t*, *tc*, *graph=None*, *infos_ale=None*, *tmy=None*, *verbose=0*)

Compute all necessary pre-requisites for solver:

1. Metrics (face normales, volume)
2. Primitive variables initialization and delete of conservative ones
3. Memory optimimization (Numa access and contiguous access of Density, VelocityX,.. VelocityZ,. . . , Temperature)
4. Work array creation (Storage of RHS, Matrix coef for implicit time scheme, lock for openmp,. . .)
5. Initialization of grid Velocities (ALE)
6. Memory optimimization access for the connectivity tree, tc.
7. Ghostcell cells filling with BC and connectivity
8. Init of ViscosityEddy for NSLaminar, LES and SA computations

Parameters

- **t** (pyTree) – input pyTree
- **tc** (pyTree) – input tree containing connection information
- **graph** (dictionary) – input communication graph
- **infos_ale** (list) – input [position angle (rad), angle rotation speed (rad s^ 1)]
- **tmy** (pyTree) – stat tree if any
- **verbose** – if 1, display information about threads distribution
- **verbose** – int (0 or 1)

Returns (t, tc, metrics)

Return type tuple

CAUTION!!!

MUST be called before compute or everytime the solution tree is modified by a non in place function

CAUTION!!!

Example of use:

- Warming up (PyTree):

```
# - warmup (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155; dx = 100./(ni-1); dz = 0.01
a = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a = C.fillEmptyBCWith(a, 'far', 'BCFarfield', dim=2)
I._initConst(a, MInf=0.4, loc='centers')
C._addState(a, 'GoverningEquations', 'Euler')
C._addState(a, MInf=0.4)
t = C.newPyTree(['Base', a])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.00004444
numz["scheme"]             = "ausmpred"
FastC._setNum2Zones(t, numz); FastC._setNum2Base(t, numb)

(t, tc, metrics)  = FastS.warmup(t, None)
```

FastS.PyTree.**createConvergenceHistory**(*t*, *nrec*)

Create a node in each zone with convergence information (residuals) MUST be called before displayTemporalCriteria() and only for steady case. t is a pyTree, nrec is the size of the data arrays to store the residuals. The data arrays are stored during the call to **FastS.displayTemporalCriteria**

Parameters

- **t** (pyTree) – input pyTRee
- **nrec** – ??

Example of use:

- ConvergenceHistory (Pytree):

```
# - convergenceHistory (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Connector.PyTree as X
import Converter.Internal as Internal
import Initiator.PyTree as I

ni = 155; dx = 100./(ni-1); dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
modulo_verif = 20
numb = {}
numb["temporal_scheme"]    = "implicit"
numb["ss_iteration"]       = 3
numb["modulo_verif"]       = modulo_verif
numz = {}
numz["time_step"]          = 0.0007
numz["time_step_nature"]   = "local"
numz["cfl"]                = 4.0
numz["scheme"]             = "roe"
numz["slope"]              = "minmod"
FastC._setNum2Zones(t, numz) ; FastC._setNum2Base(t, numb)

(t, tc, metrics) = FastS.warmup(t, None)

# Number or records to store residuals
nrec = 100//modulo_verif

#To remove old ConvergenceHistory nodes
t = C.rmNodes(t, "ZoneConvergenceHistory")

#Convergence history with nrec records
```

(continues on next page)

(continued from previous page)

```
FastS.createConvergenceHistory(t, nrec)

nit = 100; time = 0
time_step = Internal.getNodeFromName(t, 'time_step')
time_step = Internal.getValue(time_step)
for it in range(nit):
    FastS._compute(t, metrics, it, tc)
    if it%modulo_verif == 0:
            FastS.display_temporal_criteria(t, metrics, it)
    time += time_step

# time stamp
Internal.createUniqueChild(t, 'Iteration', 'DataArray_t', value=nit)
Internal.createUniqueChild(t, 'Time', 'DataArray_t', value=time)
C.convertPyTree2File(t, 'out.cgns')
```

FastS.PyTree.**createStatNodes**(*t*, *dir='0'*)

Create a tree, tmy, used by FastS._computeStats, to compute and store space and time averaged value of the flowfield.

The size of zones in tmy and t differs if space averaging occurs.

Each zone of tmy has a node, named **'parameter_int'** with contains a numpy. The number of time **samples** is stored in parameter_int[2].

The averaged fields, computed at the center of the cell, are :

1. Density
2. MomentumX
3. MomentumY
4. MomentumZ
5. Pressure
6. Pressure^2
7. ViscosityEddy
8. $\text{MomentumX}^2/\text{Density}$
9. $\text{MomentumY}^2/\text{Density}$
10. $\text{MomentumZ}^2/\text{Density}$
11. MomentumX * MomentumY / Density
12. MomentumX * MomentumZ / Density

13. MomentumY * MomentumZ / Density

Return a new tree in tmy:

Parameters

- **t** (pyTree) – input pyTree
- **dir** (character) – input to determine homogeneous direction in a block structured sense

Returns tmy

Return type tree

the **dir** character can have the following values:

- '0' : (no space average)
- 'i' : space average along the I direction of the structured block
- 'j' : space average along the J direction of the structured block
- 'k' : space average along the K direction of the structured block
- 'ij': space average along the I and J directions of the structured block
- 'ik': space average along the I and K directions of the structured block
- 'jk': space average along the J and K directions of the structured block

Example of use:

- Create Stat Node (pyTree):

```
# - createStatNodes (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155 ; dx = 100./(ni-1) ; dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}
```

(continues on next page)

(continued from previous page)

```
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.00004444
numz["scheme"]             = "ausmpred"
FastC._setNum2Zones(t, numz) ; FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

tmy = FastS.createStatNodes(t, dir='0')
C.convertPyTree2File(tmy, 'out.cgns')
```

FastS.PyTree.**createStressNodes**(*t*, *BC= BCTypes*)

Create a tree, used by FastS._computeStress, to compute and store numerical fluxes, Gradient and Cp on a list of boundary conditions. Return a new tree in teff.

Parameters

- **t** (pyTree) – input pyTree
- **BCTypes** (`list of strings`) – types of BC to extract

Returns tmy

Return type tree

Default value is BC=None.

Example of use:

- Create stress Nodes(pyTree):

```
# - createStressNodes (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155 ; dx = 100./(ni-1) ; dz = 1.
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'wall', 'BCWall', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
```

(continues on next page)

(continued from previous page)

```
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}; numz = {}
FastC._setNum2Zones(t, numz); FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

teff = FastS.createStressNodes(t, BC=['BCWall','BCWallViscous'])

FastS._computeStress(t, teff, metrics)

C.convertPyTree2File(teff, 'stress.cgns')
```

3.2 Running computation

FastS.PyTree.**_compute**(*t*, *metrics*, *nit*, *tc=None*, *graph=None*)

Perform one iteration of solver to advance (in place) the solution from t^n to t^(n+1).

nit is the current iteration number; metrics contains normale and volume informations; tc is the connectivity tree (if any):

Parameters

- **t** (pyTree) – input pyTree
- **metrics** (list) –
- **nit** (int) – current iteration number
- **tc** (pyTree) – connecting pyTree
- **graph** (dictionary) – communication graph

Example of use:

- compute 200 timesteps of Euler Eqs (PyTree):

```
# - compute (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I
```

(continues on next page)

(continued from previous page)

```
ni = 155; dx = 100./(ni-1); dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.00004444
numz["scheme"]             = "ausmpred"
FastC._setNum2Zones(t, numz) ; FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

# Compute
for it in range(1,200):
    FastS._compute(t, metrics, it)

# Save
C.convertPyTree2File(t, 'out.cgns')
```

FastS.PyTree.**displayTemporalCriteria**(*t*, *metrics*, *nit*, *format=None*)

Display CFL and implicit convergence informations.

Parameters

- **t** (pyTree) – input pyTree
- **metrics** –
- **nit** –
- **format** (string) – format for residual output ('None', 'double', 'store')

format can take 2 values:

- None (display residuals on the stdout with f7.2 Fortran format)
- 'store' (store residual in the tree if CongergenceHistory node has been created by FastS.createConvergenceHistory)

Example of use:

- Display temporal criteria in stdout (PyTree):

```
# - display_temporal_criteria (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I
import KCore.Adim as Adim

ni = 155; dx = 100./(ni-1); dz = 0.01
a = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a = C.fillEmptyBCWith(a, 'far', 'BCFarfield', dim=2)
I._initConst(a, MInf=0.4, loc='centers')
C._addState(a, 'GoverningEquations', 'Euler')
C._addState(a, MInf=0.4)
t = C.newPyTree(['Base',a])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numb["modulo_verif"]       = 1
numz = {}
#numz["time_step"]          = 0.00004444
numz["time_step_nature"]    = "local"
numz["cfl"]    = 0.6
numz["scheme"]             = "ausmpred"
FastC._setNum2Zones(t, numz); FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

# Compute
for it in range(1,5):
    FastS._compute(t, metrics, it)
    FastS.display_temporal_criteria(t, metrics, it)

# Save
C.convertPyTree2File(t, 'out.cgns')
```

3.3 Post

FastS.PyTree.**_computeStats**(*t*, *tmy*, *metrics*)

Compute the space/time average of the flowfield in a tree tmy (in place).

Parameters

- **t** (pyTree) – input pyTree
- **tmy** (pyTree) – stat tree
- **metrics** (metrics) – metrics of t

Example of use:

- Compute flowfield average over 200 timesteps and in the k direction (pyTree):

```
# - computeStats (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155; dx = 100./(ni-1); dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,15))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=3)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.00004444
numz["scheme"]             = "ausmpred"
FastC._setNum2Zones(t, numz) ; FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

## Statsnodes creation from scratch
tmy = FastS.createStatNodes(t, dir='k')

### Statsnodes creation from restart file
```

(continues on next page)

(continued from previous page)

```
#tmy = FastS.initStats('stat.cgns')

# Compute
for nitrun in range(1,200):
    print('it=', nitrun)
    FastS._compute(t, metrics, nitrun)
    FastS._computeStats(t, tmy, metrics)
C.convertPyTree2File(tmy, 'stat.cgns')
```

FastS.PyTree.**_computeStress**(*t*, *teff*, *metrics*, *xyz_ref=(0., 0., 0.)*)

Compute in teff (in place) data related to a list of Boundary conditions defined by FastS._createStressNodes.

Parameters

- **t** (pyTree) – input pyTree
- **teff** (pyTree) – stress tree
- **metrics** (metrics) – metrics of t
- **xyz_ref** (tupple of 3 floats) – reference point for momentum computation

Returns effort

Return type list

in the tree **teff**, the following variables are updated in the **FlowSolution#Centers** node thanks to primitive variable of t:

1. Density (contains the numerical fluxes linked to the mass conservation: rho (U . n) x S)
2. MomentumX (contains the normalized numerical fluxes linked to the MomentumX conservation minus P_inf*n: ((rho (U . n) Ux + (P-P_inf).nx) x S) x 0.5/rho_inf/U_inf^2
3. MomentumY...
4. MomentumZ...
5. EnergyStagnationDensity (contains the normalized numerical fluxes of the linked to the energy conservation)
6. gradxVelocityX (gradient in the x direction of VelocityX at the position of the BC)
7. gradyVelocityX
8. gradzVelocityX

9. gradxVelocityY
10. gradyVelocityY
11. gradzVelocityY
12. gradxVelocityZ
13. gradyVelocityZ
14. gradzVelocityZ
15. gradxTemperature
16. gradyTemperature
17. gradzTemperature
18. CoefPressure
19. ViscosityMolecular + ViscosityEddy
20. Density2: contains density (kg/m ^ 3)
21. Pressure

Normalized data are normalized by 0.5 rho_inf U_inf ^ 2 defined in the ReferenceState CGNS node

the return of the function, effort, is a list of 8 items which contains integral over the surface of the BC of different variables of teff:

1. integral of MomentumX (normalized numerical fluxes linked to the MomentumX conservation) give access to cx (stored in effort[0])
2. integral of MomentumY (normalized numerical fluxes linked to the MomentumY conservation) give access to cy (stored in effort[1])
3. integral of MomentumZ (normalized numerical fluxes linked to the MomentumZ conservation) give access to cz (stored in effort[2])
4. give access to cmx (stored in effort[3])
5. give access to cmy (stored in effort[4])
6. give access to cmz (stored in effort[5])
7. give access to surface of the BC (stored in effort[6])
8. integral of Density (numerical fluxes linked to the Density conservation) give access to mass flow rate (stored in effort[7])
9. integral of MomemtumX (numerical fluxes linked to the MomentumX conservation) give access to the stress (Newton) in the X direction (stored in effort[8])
10. integral of MomemtumY (numerical fluxes linked to the MomentumY conservation) give access to the stress (Newton) in the Y direction (stored in effort[9])

11. integral of MomemtumZ (numerical fluxes linked to the MomentumZ conservation) give access to the stress (Newton) in the Z direction (stored in effort[10])

For a 2D computation in (x,y) plan, with an angle of attack of theta:

- drag = eff[0]*cos(theta) + eff[1]*sin(theta)
- lift = eff[1]*cos(theta) - eff[0]*sin(theta)

Example of use:

- Compute load (pyTree):

```
# - computeStress (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155 ; dx = 100./(ni-1) ; dz = 1.
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'wall', 'BCWall', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}; numz = {}
FastC._setNum2Zones(t, numz); FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

teff = FastS.createStressNodes(t, ['BCWall'])


# Compute
for nitrun in range(1,200):
    FastS._compute(t, metrics, nitrun)

effort = FastS._computeStress(t, teff, metrics)

C.convertPyTree2File(teff, 'stress.cgns')
```

- Compute debit (pyTree) for Inflow BC condition:

```
# - computeStress (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Converter.Internal as Internal
import Initiator.PyTree as I
import numpy

ni = 155 ; dx = 100./(ni-1) ; dz = 1.
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'wall', 'BCInflow', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}; numz = {}
FastC._setNum2Zones(t, numz); FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

debit_inflow = FastS.createStressNodes(t, BC=['BCInflow'])


# Compute
for nitrun in range(1,200):
    FastS._compute(t, metrics, nitrun)

effort = FastS._computeStress(t, debit_inflow , metrics)

# compute of the mass flow rate with numerical flux old fashion
zones_debit = Internal.getZones(debit_inflow)
debit =0.
for z in zones_debit:
    sol      = Internal.getNodesFromName1(z,'FlowSolution#Centers')
    density  = Internal.getNodesFromName1(sol,'Density')[1]
    debit   +=  numpy.sum(density)

print('the mass flow rate accross the Inflow BC is', debit)

# compute of the mass flow rate with numerical flux new fashion
print('the mass flow rate accross the Inflow BC is', zones_debit[7])
```

(continues on next page)

(continued from previous page)

```
C.convertPyTree2File(debit_inflow, 'debit.cgns')
```

FastS.PyTree.**_computeVariables**(*t*, *metrics*, *variables*)

Compute specified variables.

Parameters

- **t** (pyTree) – input/output pyTree
- **metrics** (list) – input metrics of t
- **variables** (list of strings) – input variables to compute

The available variables are:

- QCriterion
- QpCriterion = QCriterion* min(1, abs(dDensitydt))
- Enstrophy

Example of use:

- Compute variables (pyTree):

```
# - computeVariables (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import FastS.PyTree as FastS
import FastC.PyTree as FastC
import Initiator.PyTree as I

ni = 155; dx = 100./(ni-1); dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=2)
a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.00004444
numz["scheme"]             = "ausmpred"
```

(continues on next page)

(continued from previous page)

```
FastC._setNum2Zones(t, numz) ; FastC._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

# Compute
for it in range(1,200):
    FastS._compute(t, metrics, it)
    FastS._computeVariables(t, metrics, ['Enstrophy'])

# Save
C.convertPyTree2File(t, 'out.cgns')
```

FastS.PyTree.**_computeGrad**(*t*, *metrics*, *variables*, *order=2*)

Compute specified variables gradients at cell centers with FV Green Gauss approach.

Parameters

- **t** (pyTree) – input/output pyTree
- **metrics** (list) – metrics of t
- **variables** (list of strings) – list of variable names to extract grad
- **order** (int (2 or 4)) – order for gradients

In case a variable is not in t or in a zone, the computation of the gradients is skipped.

order can take 2 values:

- 2: classical flux reconstruction (2nd order)
- 4: flux reconstruction formula giving 4th order accurate scheme on cartesian grid

Example of use:

- Compute grad (pyTree):

```
# - computeGrad (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G
import FastS.PyTree as FastS
import Fast.PyTree as Fast
import Initiator.PyTree as I
import KCore.test as test
```

(continues on next page)

(continued from previous page)

```
ni = 155; dx = 100./(ni-1); dz = 0.01
a1 = G.cart((-50,-50,0.), (dx,dx,dz), (ni,ni,2))
a1 = C.fillEmptyBCWith(a1, 'far', 'BCFarfield', dim=2)
a1 = I.initLamb(a1, position=(0.,0.), Gamma=2., MInf=0.4, loc='centers')
#a1 = I.initConst(a1, MInf=0.4, loc='centers')
a1 = C.addState(a1, 'GoverningEquations', 'Euler')
a1 = C.addState(a1, MInf=0.4)
t = C.newPyTree(['Base', a1])

# Numerics
numb = {}
numb["temporal_scheme"]    = "explicit"
numb["ss_iteration"]       = 20
numz = {}
numz["time_step"]          = 0.0004444
numz["scheme"]             = "ausmpred"
Fast._setNum2Zones(t, numz) ; Fast._setNum2Base(t, numb)

# Prim vars, solver tag, compact, metric
(t, tc, metrics) = FastS.warmup(t, None)

# Compute
#for it in range(0,1):
for it in range(1,50):
    FastS._compute(t, metrics, it)
    #FastS._computeGrad(t, metrics, ['Density'])

C.convertPyTree2File(t,'bug.cgns')
Internal._rmNodesByName(t, '.Solver#Param')
Internal._rmNodesByName(t, '.Solver#ownData')
test.testT(t, 1)
```

FastS.PyTree.**extractConvergenceHistory**(*t*, *fileout*)

Extract convergence information (residuals) in each zone (residuals) t is a pyTree, fileout is the name of the output file in the tecplot ascii format.

Parameters

- **t** (pyTree) – input pyTree
- **fileout** (string) – name of file for resiudal extraction

Example of use:

- Extract convergence history (pyTree):

```
# - extractConvergenceHistory (pyTree) -
import Converter.PyTree as C
import FastS.PyTree as FastS

# lecture de l'arbre cgns
t = C.convertFile2PyTree('restart.cgns')

# extraction des residus et creation du fichier "residus.dat"
FastS.extractConvergenceHistory(t,"residus.dat")
```

CHAPTER

FOUR

INDEX

- genindex
- modindex
- search