

Open-Source CFD Course

A course at Chalmers University of Technology

Taught by Håkan Nilsson

Based on OpenFOAM v2112

Presenter:

Saeed Salehi



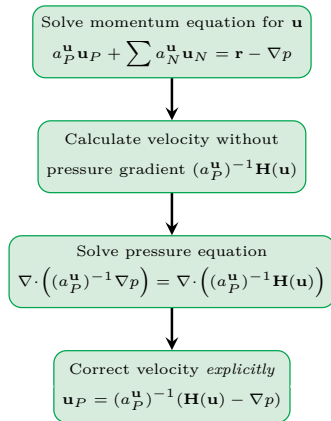
CHALMERS
UNIVERSITY OF TECHNOLOGY

Division of Fluid Dynamics
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

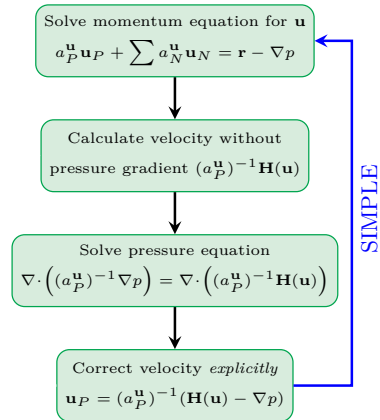
September 2022

PIMPLE algorithm and pimpleFoam solver

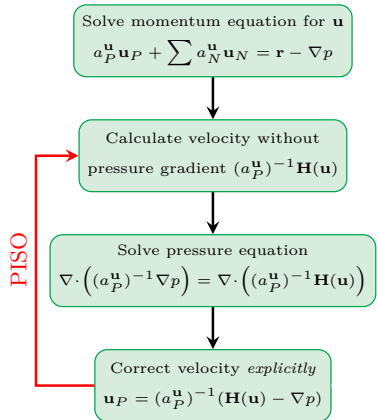
- The main steps are similar for both SIMPLE and PISO pressure correction algorithms.



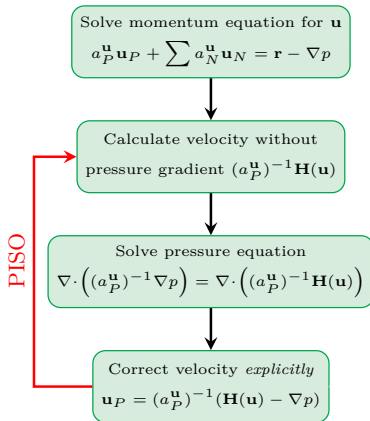
- The main steps are similar for both SIMPLE and PISO pressure correction algorithms.
- SIMPLE algorithm, originally developed for steady physics, does not care about the fact the $\mathbf{H}(\mathbf{u})$ operator was constructed using a velocity that did not fulfill continuity.
- SIMPLE goes all the way back to the beginning of the loop and proceeds in time.



- The main steps are similar for both SIMPLE and PISO pressure correction algorithms.
- SIMPLE algorithm, originally developed for steady physics, does not care about the fact the $\mathbf{H}(\mathbf{u})$ operator was constructed using a velocity that did not fulfill continuity.
- SIMPLE goes all the way back to the beginning of the loop and proceeds in time (iteration).
- PISO was developed for unsteady physics and is intended to provide time accurate results.
- PISO goes back to correction of $\mathbf{H}(\mathbf{u})$ and performs the momentum corrector step again. Then it proceeds in time.

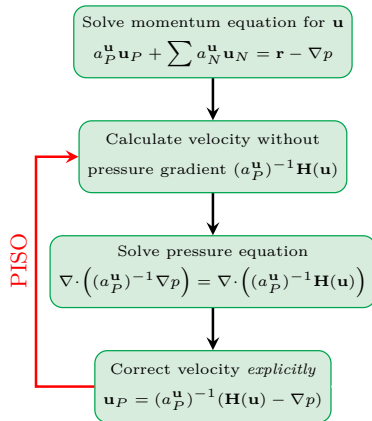


- The main steps are similar for both SIMPLE and PISO pressure correction algorithms.
- SIMPLE algorithm, originally developed for steady physics, does not care about the fact the $\mathbf{H}(\mathbf{u})$ operator was constructed using a velocity that did not fulfill continuity.
- SIMPLE goes all the way back to the beginning of the loop and proceeds in time (iteration).
- PISO was developed for unsteady physics and is intended to provide time accurate results.
- PISO goes back to correction of $\mathbf{H}(\mathbf{u})$ and performs the momentum corrector step again. Then it proceeds in time.



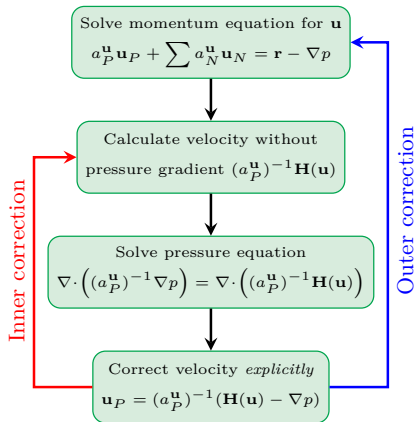
- What is the drawback with the PISO algorithm for unsteady problems?

- The main steps are similar for both SIMPLE and PISO pressure correction algorithms.
- SIMPLE algorithm, originally developed for steady physics, does not care about the fact the $\mathbf{H}(\mathbf{u})$ operator was constructed using a velocity that did not fulfill continuity.
- SIMPLE goes all the way back to the beginning of the loop and proceeds in time (iteration).
- PISO was developed for unsteady physics and is intended to provide time accurate results.
- PISO goes back to correction of $\mathbf{H}(\mathbf{u})$ and performs the momentum corrector step again. Then it proceeds in time.



- What is the drawback with the PISO algorithm for unsteady problems?
- The momentum predictor is only solved once at each time step. The linearized momentum equation uses fluxes and pressure gradient from the previous time-step. This assumption is only acceptable for very small time-steps ($Co_{\max} < 1$). Under-relaxation cannot be used.

- PIMPLE algorithm merges PISO (inner corrector) and SIMPLE (outer corrector) loops.
- The main idea behind the PIMPLE loop is to seek a fully converged steady-state solution with under-relaxation in each time step and proceed in time.
- PIMPLE works for $C_{O_{\max}} \gg 1$.
- If only one outer correction loop with multiple inner loops are performed, the PIMPLE loop resembles PISO.
- Performing one inner corrector with multiple outer loops makes the procedure similar to the SIMPLE algorithm (in each time step).
- Under-relaxations can be used to have a smooth convergence in each time step.

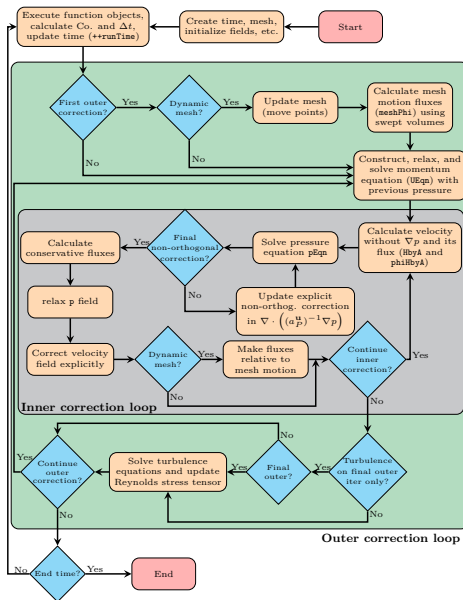


- PIMPLE algorithm is widely used in OpenFOAM. In fact, 59 solvers out of total of 108 OpenFOAM solvers employ this pressure correction algorithm, such as: `pimpleFoam`, `buoyantPimpleFoam`, `rhoPimpleFoam`, `sonicFoam`, `fireFoam`, `reactingFoam`, `sprayFoam`, `engineFoam`, `interFoam`, `twoPhaseEulerFoam`, `compressibleInterIsoFoam`, `cavitatingFoam`, `interPhaseChangeFoam`
- `grep -irl "while (pimple.loop())" $FOAM_SOLVERS | wc -l`
- `pimpleFoam` is a more sophisticated unsteady solver compared to `icoFoam` and enables several additional functionalities, such as
 - ✓ outer correction, inner correction, and non-orthogonal correction loops
 - ✓ Mesh motion
 - ✓ Adjustable time step (based on Courant number)
 - ✓ `CorrectPhi`
 - ✓ Turbulence modeling
 - ✓ Multiple Reference Frame (MRF)
 - ✓ `fvOptions` (such as porosity)
 - ✓ Under-relaxations
 - ✓ Residual control
- `pimpleFoam` provides a high level of flexibility using different switches and parameters.
- All switches and parameters can be set inside the PIMPLE dictionary of `fvSolution`.

PIMPLE switches and parameters

Parameter	Type	Default	Description
nOuterCorrectors	Integer	1	Maximum number of outer correction loops
nCorrectors	Integer	1	Maximum number of inner (PISO) correction loops
nNonOrthogonalCorrectors	Integer	0	Number of non-orthogonal correction loops
momentumPredictor	Boolean	true	Indicates whether to solve for momentum
transonic	Boolean	false	Indicates whether to use transonic algorithm (compressible solvers)
consistent	Boolean	false	Indicates whether to use "consistent" approach (SIMPLEC)
frozenFlow	Boolean	false	Indicates whether the flow system of equations should not be evolved
turbOnFinalIterOnly	Boolean	true	Indicates whether to only solve turbulence on final outer loop
finalOnLastPimpleIterOnly	Boolean	false	Indicates whether the final solver is used only on the final PIMPLE loop
ddtCorr	Boolean	true	Indicates whether the ddtCorr should be applied
correctPhi	Boolean	Dynamic mesh?	Perform flux correction functions to ensure continuity before solving momentum equation.
checkMeshCourantNo	Boolean	false	Calculate Courant number of the mesh motion
moveMeshOuterCorrectors	Boolean	false	Indicates whether to perform dynamic mesh calculations in each outer loop
solveFlow	Boolean	true	Indicates whether to solve the flow (only in sprayFoam)
SIMPLErho	Boolean	false	indicate whether to update density in SIMPLE rather than PISO (In a few solvers e.g. rhoPimpleFoam)

Flowchart of the pimpleFoam solver (some details are omitted in this flowchart, e.g., correctPhi, fvOptions, MRF, etc.)



Included header files before the main function

```

77 #include "fvCFD.H"
78 #include "dynamicFvMesh.H"
79 #include "singlePhaseTransportModel.H"
80 #include "turbulentTransportModel.H"
81 #include "pimpleControl.H"
82 #include "CorrectPhi.H"
83 #include "fvOptions.H"
84 #include "localEulerDdtScheme.H"
85 #include "fvcSmooth.H"

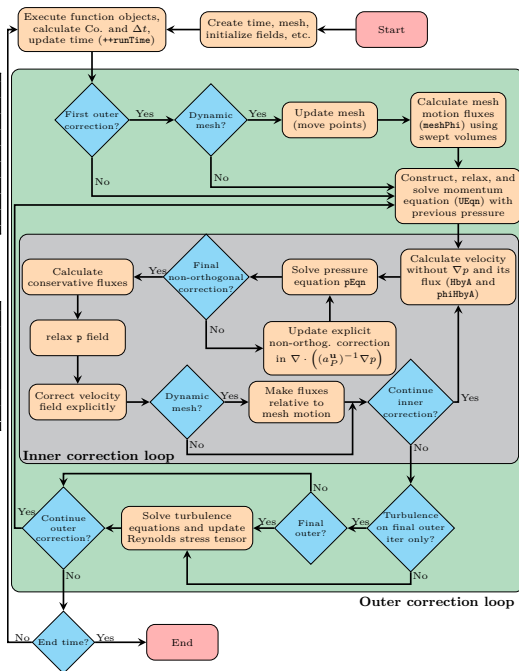
```

pimpleControl.H

```

39 #ifndef pimpleControl_H
40 #define pimpleControl_H
41
42 #include "solutionControl.H"
43
44 //- Declare that pimpleControl will be used
45 #define PIMPLE_CONTROL

```



Create time, mesh, fields, etc.

```

89 int main(int argc, char *argv[])
90 {
91     argList::addNote
92     (
93         "Transient solver for incompressible,
94         turbulent flow"
95         " of Newtonian fluids on a moving mesh."
96     );
97     #include "postProcess.H"
98
99     #include "addCheckCaseOptions.H"
100    #include "setRootCaseLists.H"
101    #include "createTime.H"
102    #include "createDynamicFvMesh.H"
103    #include "initContinuityErrs.H"
104    #include "createDyMControls.H"
105    #include "createFields.H"
106    #include "createUfIfPresent.H"
107    #include "CourantNo.H"
108    #include "setInitialDeltaT.H"
109
110    turbulence->validate();

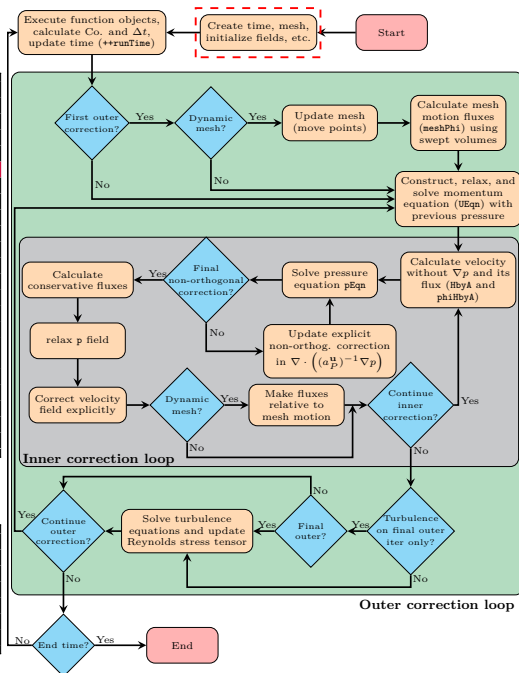
```

createDynamicFvMesh.H

```

1 Info<< "Create mesh for time = "
2     << runTime.timeName() << nl << endl;
3
4 autoPtr<dynamicFvMesh> meshPtr(dynamicFvMesh::New(args
5     , runTime));
6
7 dynamicFvMesh& mesh = meshPtr();

```

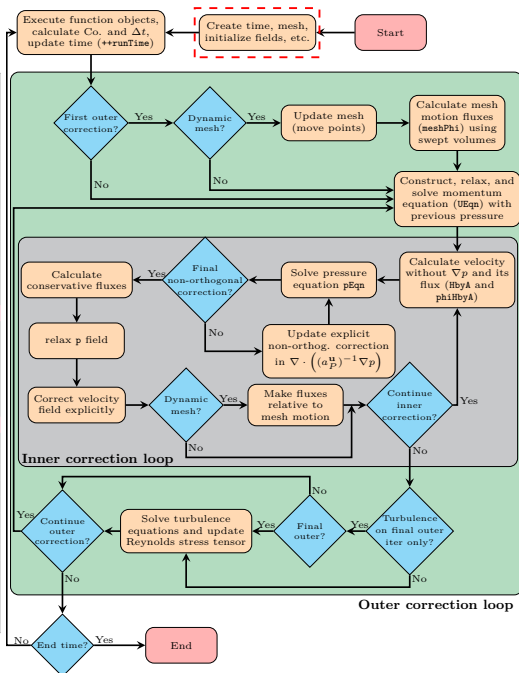


createDyMControls.H

```

1 bool correctPhi
2 (
3     pimple.dict().getOrDefault("correctPhi", true)
4 );
5
6 bool checkMeshCourantNo
7 (
8     pimple.dict().getOrDefault("checkMeshCourantNo",
9         false)
10 );
11 bool moveMeshOuterCorrectors
12 (
13     pimple.dict().getOrDefault("
14         moveMeshOuterCorrectors", false)
15 );
16
17 bool massFluxInterpolation
18 (
19     pimple.dict().getOrDefault("massFluxInterpolation",
20         false)
21 );
22
23 bool adjustFringe
24 (
25     pimple.dict().getOrDefault("oversetAdjustPhi",
26         false)
27 );
28
29 bool ddtCorr
30 (
31     pimple.dict().getOrDefault("ddtCorr", true)
32 );

```



createControl.H

```

1  #if defined(NO_CONTROL)
2  #elif defined(PISO_CONTROL)
3      #include "createPisoControl.H"
4  #elif defined(PIMPLE_CONTROL)
5      #include "createPimpleControl.H"
6  #elif defined(SIMPLE_CONTROL)
7      #include "createSimpleControl.H"
8  #endif

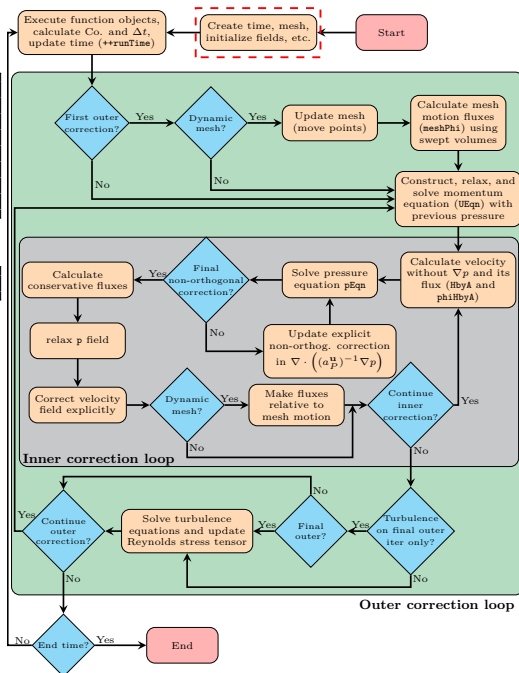
```

createPimpleControl.H

```

1  pimpleControl pimple(mesh);

```



Function object, Co., Δt , update time

```

122 while (runTime.run())
123 {
124     #include "readDyMControls.H"
125
126     if (LTS)
127     {
128         #include "setRDeltaT.H"
129     }
130     else
131     {
132         #include "CourantNo.H"
133         #include "setDeltaT.H"
134     }
135
136     ++runTime;

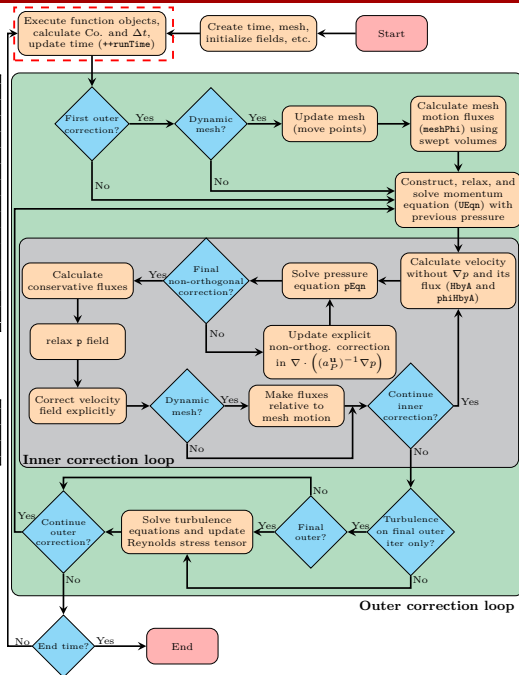
```

● runTime is an instance of class Time.
createTime.H

```

1 Foam::Info<< "Create time\n" << Foam::endl;
2
3 Foam::Time runTime(Foam::Time::controlDictName, args);

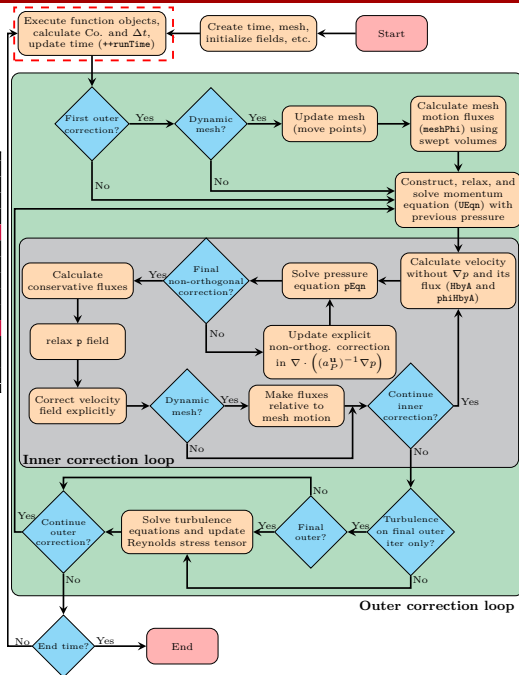
```



- run function of class Time returns a boolean about whether the simulation should be continued or not. It also invokes function objects.
- Can you spot the small difference between code and flowchart?

run function

```
...
if (timeIndex_ == startTimeIndex_)
{
    addProfiling(functionObjects, "functionObjects.
        start()");
    functionObjects_.start();
}
else
{
    addProfiling(functionObjects, "functionObjects.
        execute()");
    functionObjects_.execute();
}
...
```



Function object, Co., Δt , update time

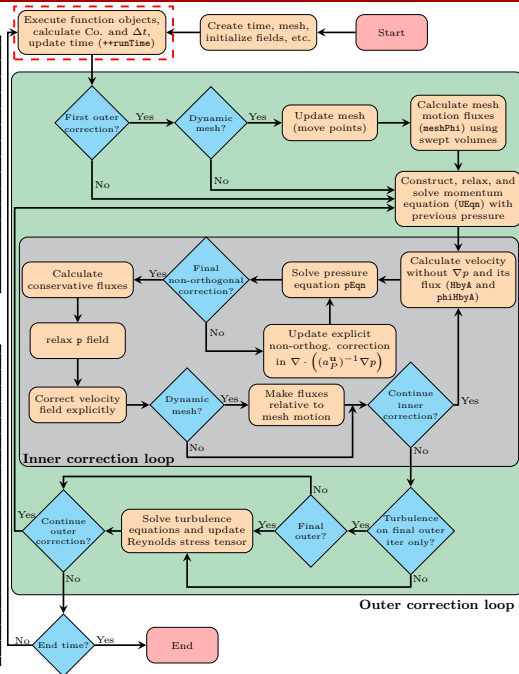
```

122 while (runTime.run())
123 {
124     #include "readDyMControls.H"
125
126     if (LTS)
127     {
128         #include "setRDeltaT.H"
129     }
130     else
131     {
132         #include "CourantNo.H"
133         #include "setDeltaT.H"
134     }
135
136     ++runTime;
    
```

readDyMControls.H

```

1 #include "readTimeControls.H"
2
3 correctPhi = pimple.dict().getOrDefault
4 (
5     "correctPhi",
6     correctPhi
7 );
8
9 checkMeshCourantNo = pimple.dict().getOrDefault
10 (
11     "checkMeshCourantNo",
12     checkMeshCourantNo
13 );
14
15 moveMeshOuterCorrectors = pimple.dict().getOrDefault
16 (
17     "moveMeshOuterCorrectors",
18     moveMeshOuterCorrectors
19 );
    
```



Function object, Co., Δt , update time

```

122 while (runTime.run())
123 {
124     #include "readDyMControls.H"
125
126     if (LTS)
127     {
128         #include "setRDeltaT.H"
129     }
130     else
131     {
132         #include "CourantNo.H"
133         #include "setDeltaT.H"
134     }
135
136     ++runTime;

```

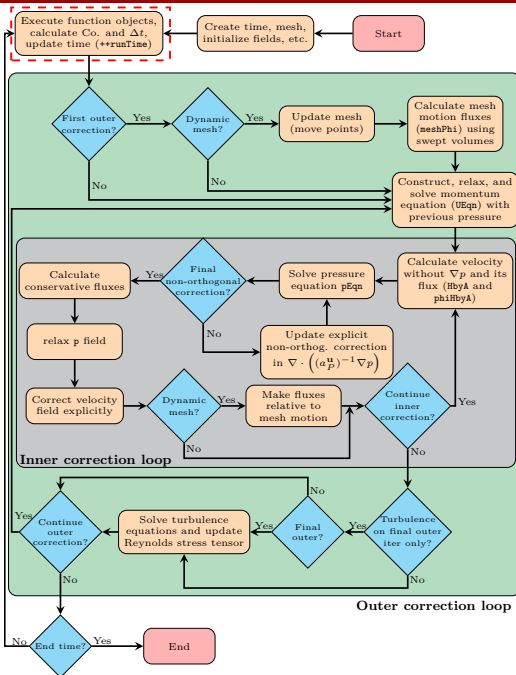
- Increment time with "+" overloaded operator
- ++runTime or runTime++?
- Both have the same effect. However, in C++ pre-increment (++i) first increases i and then return the value but post-increment (i++) first return the value and then increase it.
- The performance of overloaded operator++ could be significantly affected since post-increment needs the creation of the temporary object.
- From OpenFOAM-v1812 the runTime increment operator switched to pre-increment (only in ESI version).

Foam::Time& Foam::Time::operator++()

```

...
setTime(value() + deltaT_, timeIndex_ + 1);
...

```



Start of outer (PIMPLE) loop

```

141 while (pimple.loop())
142 {
143     if (pimple.firstIter() ||
        moveMeshOuterCorrectors)
144     {
145         // Do any mesh changes
146         mesh.controlledUpdate();
    
```

- All the dynamic mesh calculations happen in the controlledUpdate function.
- How about static mesh? (Hint: dynamic binding)
- controlledUpdate calls the correct update function on the mesh.

update function of dynamicMotionSolverFvMesh class

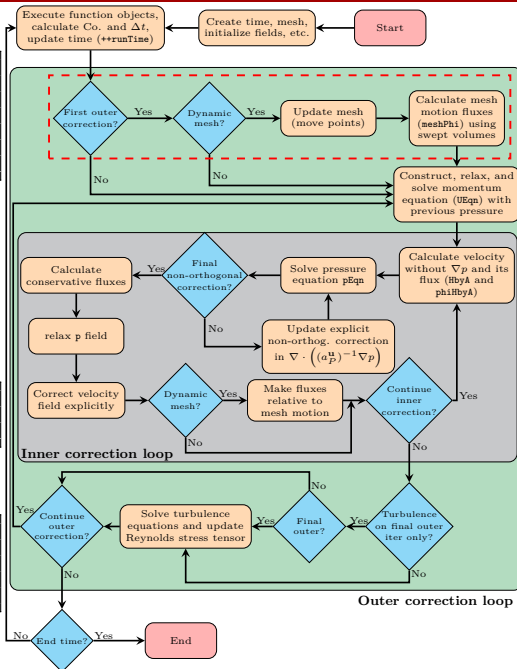
```

...
fvMesh::movePoints(motionPtr->newPoints());
...
    
```

movePoints function of fvMesh class

```

...
phi.primitiveFieldRef() =
scalarField::subField(sweptVols, nInternalFaces());
phi.primitiveFieldRef() *= rDeltaT;
...
    
```



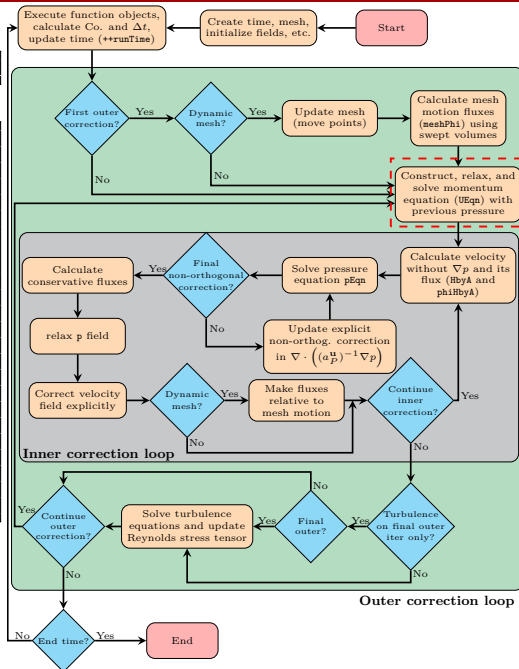
Including UEqn in pimpleFoam

```
#include "UEqn.H"
```

UEqn.H

```
1 // Solve the Momentum equation
2 MRF.correctBoundaryVelocity(U);
3
4 tmp<fvVectorMatrix> tUEqn
5 (
6     fvm::ddt(U) + fvm::div(phi, U)
7     + MRF.DDt(U)
8     + turbulence->divDevReff(U)
9     ==
10    fvOptions(U)
11 );
12 fvVectorMatrix& UEqn = tUEqn.ref();
13 UEqn.relax();
14 fvOptions.constrain(UEqn);
15
16 if (pimple.momentumPredictor())
17 {
18     solve(UEqn == -fvc::grad(p));
19     fvOptions.correct(U);
20 }
21
22 }
```

- UEqn is constructed.
- It is implicitly relaxed (will be discussed later).
- If momentumPredictor is true, UEqn is solved.
- “==” operator



```
// --- Pressure corrector loop
while (pimple.correct())
{
    #include "pEqn.H"
}
```

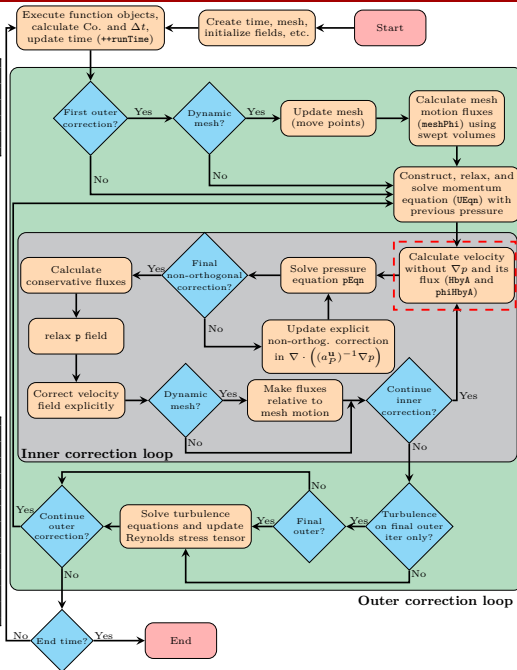
- Velocity without pressure gradient ($(a_P^u)^{-1} \mathbf{H}$, \mathbf{HByA}) and its corresponding flux ($\phi_{\mathbf{HByA}}$) are constructed.
- `ddtCorr` correction is applied in case needed.
- From OpenFOAM-v2006 the `ddtCorr` switch is added. The `else` statement throws dimensionality check error (run time error).
- Details on MRF, pressure reference, and consistent PIMPLE are not shown and explained here.

Beginning of pEqn.H

```

1 volScalarField rAU(1.0/UEqn.A());
2 volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
3 surfaceScalarField phiHbyA("phiHbyA", fvc::flux(HbyA));
4
5 if (pimple.ddtCorr())
6 {
7     phiHbyA += MRF.zeroFilter(fvc::interpolate(rAU)*fvc
8                               ::ddtCorr(U, phi, Uf));
9 }
10
11 MRF.makeRelative(phiHbyA);

```



Non-orthogonal pressure corrector loop

```

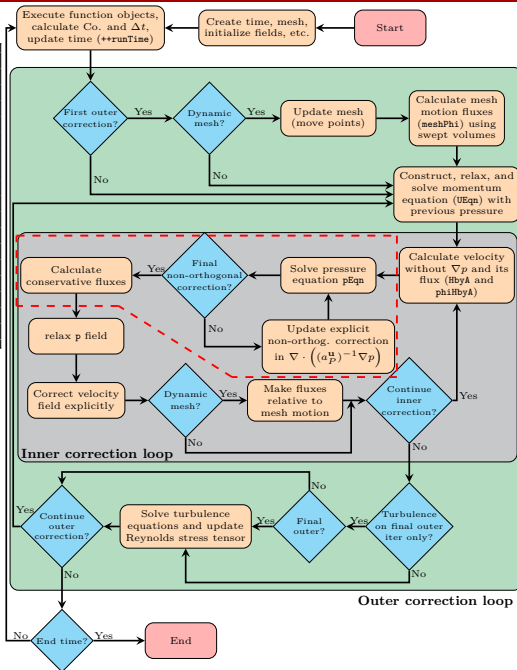
38 // Non-orthogonal pressure corrector loop
39 while (pimple.correctNonOrthogonal())
40 {
41     fvScalarMatrix pEqn
42     (
43         fvm::laplacian(rAtU(), p) == fvc::div(phiHbyA)
44     );
45     pEqn.setReference(pRefCell, pRefValue);
46     pEqn.solve(mesh.solver(p.select(pimple.
47         finalInnerIter())));
48     if (pimple.finalNonOrthogonalIter())
49     {
50         phi = phiHbyA - pEqn.flux();
51     }
52 }
53
54

```

- The pressure (continuity) equation is constructed as

$$\nabla \cdot \left((a_P^u)^{-1} \nabla p \right) = \nabla \cdot \left((a_P^u)^{-1} \mathbf{H}(\mathbf{u}) \right)$$

- The appropriate solver is read from fvSolution based on p.select(pimple.finalInnerIter()) that returns a word type which is either p or pFinal.
- In each loop the explicit non-orthogonal correction term is updated. In other words, the non-orthogonal correction loop only updates the left-hand side of pEqn, while in the pressure corrector loop both sides are updated.
- In the final non-orthogonal correction loop, the conservative face fluxes are calculated.



End of pEqn.H

```

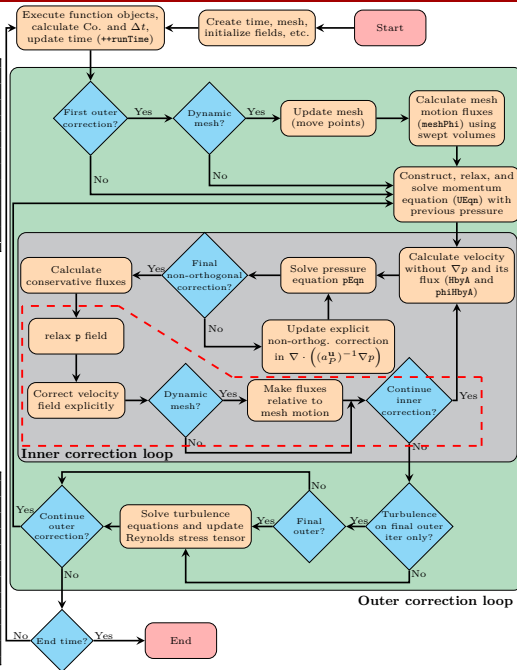
59 p.relax();
60
61 U = HbyA - rAtU*fvc::grad(p);
62 U.correctBoundaryConditions();
63 fvcOptions.correct(U);
64
65 // Correct Uf if the mesh is moving
66 fvc::correctUf(Uf, U, phi);
67
68 // Make the fluxes relative to the mesh motion
69 fvc::makeRelative(phi, U);
    
```

- field p is relaxed explicitly using the stored value from the previous PIMPLE iteration (will be discussed later).
- Velocity field is corrected through new pressure.
- In the case of mesh motion, the conservative fluxes are made relative to the mesh motion (meshPhi).

Definition of makeRelative in fvcMeshPhi.C

```

void Foam::fvc::makeRelative
(
    surfaceScalarField& phi,
    const volVectorField& U
)
{
    if (phi.mesh().moving())
    {
        phi -= fvc::meshPhi(U);
    }
}
    
```



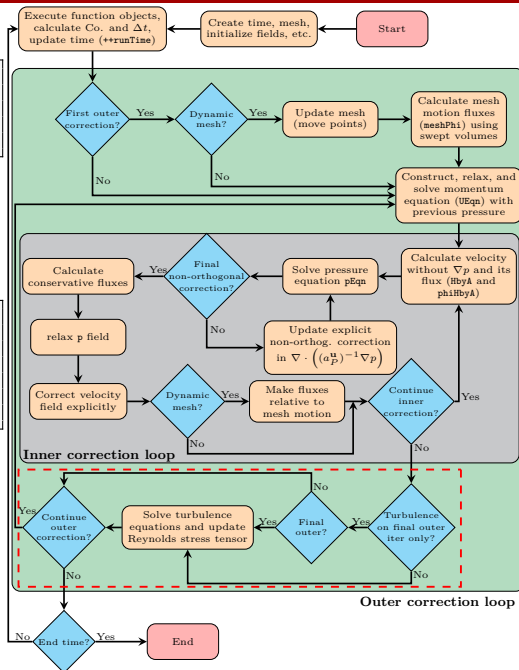
End of PIMPLE loop

```
179     if (pimple.turbCorr())
180     {
181         laminarTransport.correct();
182         turbulence->correct();
183     }
```

- If `turbCorr()` returns true, the turbulence equations are solved and the Reynolds stress tensor is updated.

Definition of `turbCorr` in `pimpleControlI.H`

```
inline bool Foam::pimpleControl::turbCorr()
{
    ...
    return !turbOnFinalIterOnly_ || finalIter();
}
```



Under relaxation factors

- Under-relaxations are required to improve the stability of computations by decreasing the change of variables between iterations.
- They should be chosen low enough to ensure stability but high enough to have a fast convergence.
- In OpenFOAM, the numerical solution is *relaxed* in two different ways, i.e., fields and equations.

An example of `relaxationFactors` dictionary in the `fvSolution`

```
relaxationFactors
{
    fields
    {
        p                0.6;
    }
    equations
    {
        "(U|k|omega)"    0.7; //table ((0 0.4) (0.5 0.7));
        "(U|k|omega)Final" 1.0;
    }
}
```

- For instance, in PIMPLE loop, `UEqn` and `p` are relaxed using `UEqn.relax()` and `p.relax()`.

- UEqn is a fvVectorMatrix which represents an *equation*. UEqn.relax() calls the relax() function defined in fvMatrix class.
- p is a volScalarField which represents a *field*. p.relax() calls the relax() function defined in GeometricField class.
- Note that UEqn equation is first relaxed and then solved, while a solution for p is first obtained and then the field is relaxed.
- Relaxing equations are also known as implicit relaxation while applying under-relaxation factor on a field is an explicit relaxation.

UEqn.H

```

5 tmp<fvVectorMatrix> tUEqn
6 (
7     fvm::ddt(U) + fvm::div(phi, U)
8     + MRF.DDt(U)
9     + turbulence->divDevReff(U)
10    ==
11    fvOptions(U)
12 );
13 fvVectorMatrix& UEqn = tUEqn.ref();
14
15 UEqn.relax();
16
17 fvOptions.constrain(UEqn);
18
19 if (pimple.momentumPredictor())
20 {
21     solve(UEqn == -fvc::grad(p));
22
23     fvOptions.correct(U);
24 }

```

pEqn.H

```

38 // Non-orthogonal pressure corrector loop
39 while (pimple.correctNonOrthogonal())
40 {
41     fvScalarMatrix pEqn
42     (
43         fvm::laplacian(rAtU(), p) == fvc::div(phiHbyA)
44     );
45
46     pEqn.setReference(pRefCell, pRefValue);
47
48     pEqn.solve(mesh.solver(p.select(pimple.
49         finalInnerIter())));
50
51     if (pimple.finalNonOrthogonalIter())
52     {
53         phi = phiHbyA - pEqn.flux();
54     }
55
56 #include "continuityErrs.H"
57
58 // Explicitly relax pressure for momentum corrector
59 p.relax();

```

Implicit relaxation, theory

- Implicit relaxation method is also known as Patankar's Under-relaxation.
- It is shown in the linear algebra that increasing diagonal dominance of a linear system enhances the stability of the iterative linear solver.
- For example, the Jacobi and Gauss-Seidel methods only converge when the matrix is strictly diagonally dominant.
- Consider the following discretized equation for the general variable ϕ

$$a_P \phi_P^n + \sum_N a_N \phi_N^n = \mathbf{r}$$

- The diagonal dominance of this linear system could be improved by adding artificial terms to both sides of equation

$$a_P \phi_P^n + \frac{1 - \alpha}{\alpha} a_P \phi_P^n + \sum_N a_N \phi_N^n = \mathbf{r} + \frac{1 - \alpha}{\alpha} a_P \phi_P^{n-1}$$

$$\frac{a_P}{\alpha} \phi_P^n + \sum_N a_N \phi_N^n = \mathbf{r} + \left(\frac{a_P}{\alpha} - a_P \right) \phi_P^{n-1}$$

ϕ_P^{n-1} : value of ϕ from the previous iteration α : under-relaxation factor

- If the linear system reaches an adequate steady-state convergence (i.e., $\phi_P^{n-1} \approx \phi_P^n$), the artificial terms cancel out and the linear system will be equivalent to the original one.

Implicit relaxation, implementation

relax() function in fvMatrix class

```

1239 template<class Type>
1240 void Foam::fvMatrix<Type>::relax()
1241 {
1242     word name = psi_.select
1243     (
1244         psi_.mesh().data::template getOrDefault<bool>
1245         ("finalIteration", false)
1246     );
1247
1248     if (psi_.mesh().relaxEquation(name))
1249     {
1250         relax(psi_.mesh().equationRelaxationFactor(name));
1251     }
1252 }
```

- The correct name is first constructed based on the outer correction state (e.g., U or UFinal).
- relax() reads the value of under-relaxation from fvSolution and then calls the relax(const scalar alpha) (with argument) function.

Implicit relaxation, implementation

$$\frac{a_P}{\alpha} \phi_P^n + \sum_N a_N \phi_N^n = \mathbf{r} + \left(\frac{a_P}{\alpha} - a_P \right) \phi_P^{n-1}$$

- `relax(const scalar alpha)` function first store the current diagonal coefficients (D0).
- Regardless of α value the matrix diagonal equality/dominance is ensured.
- The diagonal coefficients are replaced by the maximum value of absolute diagonal coefficients and sum of absolute off-diagonals. This step is carried out regardless of α value.
- The matrix is relaxed by dividing diagonal members by α .
- The contribution of relaxation is added to the source term. It can be easily shown that with the current implementation, any manipulation can be done to D as long as its corresponding contribution is added to the source term.

`relax(const scalar alpha)` in `fvMatrix` class

```
...

Field<Type>& S = source();
scalarField& D = diag();

// Store the current unrelaxed diagonal for use in
// updating the source
scalarField D0(D);

// Calculate the sum-mag off-diagonal from the
// interior faces
scalarField sumOff(D.size(), Zero);
sumMagOffDiag(sumOff);
...

// Ensure the matrix is diagonally dominant...
// Assumes that the central coefficient is positive
// and ensures it is
forAll(D, celli)
{
    D[celli] = max(mag(D[celli]), sumOff[celli]);
}
// ... then relax
D /= alpha;
...

// Finally add the relaxation contribution to the
// source.
S += (D - D0)*psi_.primitiveField();
```

Implicit relaxation, implementation

- It is seen that relaxing an equation manipulates the source term and makes the equation more explicit.
- More iterations (outer corrections) should be performed to make sure that the convergence is achieved. The under-relaxation should be used only if they are needed.
- Regardless of α value, implicit relaxation guarantees matrix diagonal equality/dominance.
- Therefore, it is common to have `relaxationFactors` dictionary as

An exmple of `relaxationFactors` dictionary in the `fvSolution`

```
relaxationFactors
{
    equations
    {
        ".*"          1;
    }
}
```


Explicit relaxation

- The explicit relaxation is applied on the fields (e.g., `p.relax()`).
- In each iteration, after obtaining a new solution, the field is relaxed using the value from previous iteration:

$$\phi_{\text{relaxed}}^n = \phi^{n-1} + \alpha (\phi^n - \phi^{n-1})$$

- Here again, the correct name is first constructed based on the outer correction state (e.g., `p` or `pFinal`).
- The value of under-relaxation is read and the `relax(const scalar alpha)` function is subsequently called.

`relax()` function in `GeometricField` class

```

1110 void Foam::GeometricField<Type, PatchField, GeoMesh>::relax()
1111 {
1112     word name = this->name();
1113
1114     if
1115     (
1116         this->mesh().data::template getDefault<bool>
1117         (
1118             "finalIteration",
1119             false
1120         )
1121     )
1122     {
1123         name += "Final";
1124     }
1125
1126     if (this->mesh().relaxField(name))
1127     {
1128         relax(this->mesh().fieldRelaxationFactor(name));
1129     }
1130 }

```

Explicit relaxation

relax(const scalar alpha) function in GeometricField class

```

971 void Foam::GeometricField<Type, PatchField, GeoMesh>::relax(const scalar alpha)
972 {
973     DebugInFunction
974         << "Relaxing" << nl << this->info() << " by " << alpha << endl;
975
976     operator==(prevIter() + alpha*(*this - prevIter()));
977 }
  
```

$$\phi_{\text{relaxed}}^n = \phi^{n-1} + \alpha (\phi^n - \phi^{n-1})$$

- The value from the previous iteration is called using prevIter().
- What exactly does the previous iteration mean?
- In PIMPLE different iterations/loops exists (Time step, outer correction, inner correction, non-orthogonal correction, linear solver iteration).
- prevIter() returns the value of fieldPrevIterPtr_ pointer. i.e.,
 return *fieldPrevIterPtr_;
- The pointer itself is set inside storePrevIter() function of the same class. Therefore, one should look for the place that storePrevIter() is called inside the solver.

Explicit relaxation

- The `loop()` function is called on the `pimple` object at the beginning of each PIMPLE outer correction loop, while `(pimple.loop())`.
- In the `loop()` function, the `storePrevIterFields()` from `solutionControl` is executed:

`storePrevIterFields()` function in `solutionControl`

```

160 void Foam::solutionControl::storePrevIterFields() const
161 {
162     //     storePrevIter<label>();
163     storePrevIter<scalar>();
164     storePrevIter<vector>();
165     storePrevIter<sphericalTensor>();
166     storePrevIter<symmTensor>();
167     storePrevIter<tensor>();
168 }
```

- `storePrevIterFields` calls the `storePrevIter`.
- Therefore, the explicit relaxation in PIMPLE loop uses contribution from previous outer corrector loop.
- Unlike implicit relaxations of equations, setting explicit relaxation of fields to $\alpha = 1$ does not have any effects.