`writeCompression` Switch to specify whether files are compressed with gzip when written: `on`/`off` (`yes`/`no`, `true`/`false`)

`timeFormat` Choice of format of the naming of the time directories.

- `fixed`: $\pm m.dddddd$ where the number of ds is set by `timePrecision`.
- `scientific`: $\pm m.dddddde\pm xx$ where the number of ds is set by `timePrecision`.
- `general` (default): Specifies `scientific` format if the exponent is less than -4 or greater than or equal to that specified by `timePrecision`.

`timePrecision` Integer used in conjunction with `timeFormat` described above, 6 by default.

`graphFormat` Format for graph data written by an application.

- `raw` (default): Raw ASCII format in columns.
- `gnuplot`: Data in gnuplot format.
- `xmgr`: Data in Grace/xmgr format.
- `jplot`: Data in jPlot format.

4.4.3 Other settings

`adjustTimeStep` Switch used by some solvers to adjust the time step during the simulation, usually according to `maxCo`.

`maxCo` Maximum Courant number, *e.g.* `0.5`

`runTimeModifiable` Switch for whether dictionaries, *e.g.controlDict*, are re-read during a simulation at the beginning of each time step, allowing the user to modify parameters during a simulation.

`libs` List of additional libraries (on $LD_LIBRARY_PATH) to be loaded at run-time, *e.g.*`("libNew1.so" "libNew2.so")`

`functions` Dictionary of functions, *e.g.* `probes` to be loaded at run-time; see examples in *$FOAM_TUTORIALS*

4.5 Numerical schemes

The *fvSchemes* dictionary in the *system* directory sets the numerical schemes for terms, such as derivatives in equations, that are calculated during a simulation. This section describes how to specify the schemes in the *fvSchemes* dictionary.

The terms that must typically be assigned a numerical scheme in *fvSchemes* range from derivatives, *e.g.* gradient ∇, to interpolations of values from one set of points to another. The aim in OpenFOAM is to offer an unrestricted choice to the user, starting with the choice of discretisation practice which is generally standard Gaussian finite volume integration. Gaussian integration is based on summing values on cell faces, which must be interpolated from cell centres. The user has a wide range of options for interpolation scheme, with certain schemes being specifically designed for particular derivative terms, especially the advection divergence $\nabla \bullet$ terms.

The set of terms, for which numerical schemes must be specified, are subdivided within the *fvSchemes* dictionary into the categories below.

- `timeScheme`: first and second time derivatives, *e.g.* $\partial/\partial t, \partial^2/\partial^2 t$
- `gradSchemes`: gradient ∇
- `divSchemes`: divergence $\nabla \bullet$
- `laplacianSchemes`: Laplacian ∇^2
- `interpolationSchemes`: cell to face interpolations of values.
- `snGradSchemes`: component of gradient normal to a cell face.
- `wallDist`: distance to wall calculation, where required.

Each keyword in represents the name of a sub-dictionary which contains terms of a particular type, *e.g.*`gradSchemes` contains all the gradient derivative terms such as `grad(p)` (which represents ∇p). Further examples can be seen in the extract from an *fvSchemes* dictionary below:

```

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default             none;

    div(phi,U)          Gauss linearUpwind grad(U);
    div(phi,k)          Gauss upwind;
    div(phi,epsilon)    Gauss upwind;
    div(phi,R)          Gauss upwind;
    div(R)              Gauss linear;
    div(phi,nuTilda)    Gauss upwind;

    div((nuEff*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}


// ************************************************************************* //
```

The example shows that the *fvSchemes* dictionary contains 6 . . . *Schemes* subdictionaries containing keyword entries for each term specified within including: a `default` entry;

other entries whose names correspond to a word identifier for the particular term specified, *e.g.*`grad(p)` for ∇p If a `default` scheme is specified in a particular *...Schemes* sub-dictionary, it is assigned to all of the terms to which the sub-dictionary refers, *e.g.* specifying a `default` in *gradSchemes* sets the scheme for all gradient terms in the application, *e.g.* ∇p, $\nabla \mathbf{U}$. When a `default` is specified, it is not necessary to specify each specific term itself in that sub-dictionary, *i.e.* the entries for `grad(p)`, `grad(U)` in this example. However, if any of these terms are included, the specified scheme overrides the `default` scheme for that term.

Alternatively the user can specify that `no default` scheme by the `none` entry, as in the `divSchemes` in the example above. In this instance the user is obliged to specify all terms in that sub-dictionary individually. Setting `default` to `none` may appear superfluous since `default` can be overridden. However, specifying `none` forces the user to specify all terms individually which can be useful to remind the user which terms are actually present in the application.

OpenFOAM includes a vast number of discretisation schemes, from which only a few are typically recommended for real-world, engineering applications. The user can get help with scheme selection by interrogating the tutorial cases for example scheme settings. They should look at the schemes used in relevant cases, *e.g.* for running a large-eddy simulation (LES), look at schemes used in tutorials running LES. Additionally, foamSearch provides a useful tool to get a quick list of schemes used in all the tutorials. For example, to print all the `default` entries for `ddtSchemes` for cases in the *$FOAM_TUTORIALS* directory, the user can type:

```
foamSearch $FOAM_TUTORIALS fvSchemes ddtSchemes.default
```

which prints:

```
default         backward;
default         CrankNicolson 0.9;
default         Euler;
default         localEuler;
default         none;
default         steadyState;
```

The schemes listed using foamSearch are described in the following sections.

4.5.1 Time schemes

The first time derivative ($\partial/\partial t$) terms are specified in the *ddtSchemes* sub-dictionary. The discretisation schemes for each term can be selected from those listed below.

- `steadyState`: sets time derivatives to zero.
- `Euler`: transient, first order implicit, bounded.
- `backward`: transient, second order implicit, potentially unbounded.

- `CrankNicolson`: transient, second order implicit, bounded; requires an off-centering coefficient ψ where:

$$\psi = \begin{cases} 1 & \text{corresponds to pure } \texttt{CrankNicolson}, \\ 0 & \text{corresponds to } \texttt{Euler}; \end{cases}$$

  generally $\psi = 0.9$ is used to bound/stabilise the scheme for practical engineering problems.

- `localEuler`: pseudo transient for accelerating a solution to steady-state using local-time stepping; first order implicit.

Solvers are generally configured to simulate either transient or steady-state. Changing the time scheme from one which is steady-state to transient, or visa versa, does not affect the fundamental nature of the solver and so fails to achieve its purpose, yielding a nonsensical solution.

Any second time derivative ($\partial^2/\partial t^2$) terms are specified in the *d2dt2Schemes* sub-dictionary. Only the `Euler` scheme is available for *d2dt2Schemes*.

4.5.2 Gradient schemes

The *gradSchemes* sub-dictionary contains gradient terms. The `default` discretisation scheme that is primarily used for gradient terms is:

```
default         Gauss linear;
```

The `Gauss` entry specifies the standard finite volume discretisation of Gaussian integration which requires the interpolation of values from cell centres to face centres. The interpolation scheme is then given by the `linear` entry, meaning linear interpolation or central differencing.

In some tutorials cases, particular involving poorer quality meshes, the discretisation of specific gradient terms is overridden to improve boundedness and stability. The terms that are overridden in those cases are the velocity gradient

```
grad(U)         cellLimited Gauss linear 1;
```

and, less frequently, the gradient of turbulence fields, *e.g.*

```
grad(k)         cellLimited Gauss linear 1;
grad(epsilon)   cellLimited Gauss linear 1;
```

They use the `cellLimited` scheme which limits the gradient such that when cell values are extrapolated to faces using the calculated gradient, the face values do not fall outside the bounds of values in surrounding cells. A limiting coefficient is specified after the underlying scheme for which 1 guarantees boundedness and 0 applies no limiting; 1 is invariably used.

Other schemes that are rarely used are as follows.

- `leastSquares`: a second-order, least squares distance calculation using all neighbour cells.

- `Gauss cubic`: third-order scheme that appears in the dnsFoam simulation on a regular mesh.

4.5.3 Divergence schemes

The *divSchemes* sub-dictionary contains divergence terms, *i.e.* terms of the form $\nabla \bullet \ldots$, excluding Laplacian terms (of the form $\nabla \bullet (\Gamma \nabla \ldots)$). This includes both advection terms, *e.g.* $\nabla \bullet (\mathbf{U}k)$, where velocity **U** provides the advective flux, and other terms, that are often diffusive in nature, *e.g.* $\nabla \bullet \nu(\nabla \mathbf{U})^{\mathrm{T}}$.

The fact that terms that are fundamentally different reside in one sub-dictionary means that the `default` scheme in generally set to `none` in *divSchemes*. The non-advective terms then generally use the `Gauss` integration with `linear` interpolation, *e.g.*

```
div(U)          Gauss linear;
```

The treatment of advective terms is one of the major challenges in CFD numerics and so the options are more extensive. The keyword identifier for the advective terms are usually of the form `div(phi,...)`, where `phi` generally denotes the (volumetric) flux of velocity on the cell faces for constant-density flows and the mass flux for compressible flows, *e.g.* `div(phi,U)` for the advection of velocity, `div(phi,e)` for the advection of internal energy, `div(phi,k)` for turbulent kinetic energy, *etc.* For advection of velocity, the user can run the foamSearch script to extract the `div(phi,U)` keyword from all tutorials.

```
foamSearch $FOAM_TUTORIALS fvSchemes "divSchemes.div(phi,U)"
```

The schemes are all based on `Gauss` integration, using the flux `phi` and the advected field being interpolated to the cell faces by one of a selection of schemes, *e.g.* `linear`, `linearUpwind`, *etc.* There is a `bounded` variant of the discretisation, discussed later.

Ignoring 'V'-schemes (with keywords ending "`V`"), and rarely-used schemes such as `Gauss cubic` and `vanLeerV`, the interpolation schemes used in the tutorials are as follows.

- `linear`: second order, unbounded.
- `linearUpwind`: second order, upwind-biased, unbounded (but much less so than `linear`), that requires discretisation of the velocity gradient to be specified.
- `LUST`: blended 75% `linear`/ 25%`linearUpwind` scheme, that requires discretisation of the velocity gradient to be specified.
- `limitedLinear`: `linear` scheme that limits towards `upwind` in regions of rapidly changing gradient; requires a coefficient, where 1 is strongest limiting, tending towards `linear` as the coefficient tends to 0.
- `upwind`: first-order bounded, generally too inaccurate to be recommended.

Example syntax for these schemes is as follows.

```
div(phi,U)      Gauss linear;
div(phi,U)      Gauss linearUpwind grad(U);
div(phi,U)      Gauss LUST grad(U);
div(phi,U)      Gauss LUST unlimitedGrad(U);
div(phi,U)      Gauss limitedLinear 1;
div(phi,U)      Gauss upwind;
```

'V'-schemes are specialised versions of schemes designed for vector fields. They differ from conventional schemes by calculating a single limiter which is applied to all components of the vectors, rather than calculating separate limiters for each component of the vector. The 'V'-schemes' single limiter is calculated based on the direction of most rapidly changing gradient, resulting in the strongest limiter being calculated which is most stable but arguably less accurate. Example syntax is as follows.

```
div(phi,U)      Gauss limitedLinearV 1;
div(phi,U)      Gauss linearUpwindV grad(U);
```

The `bounded` variants of schemes relate to the treatment of the material time derivative which can be expressed in terms of a spatial time derivative and convection, *e.g.* for field e in incompressible flow

$$\frac{De}{Dt} = \frac{\partial e}{\partial t} + \mathbf{U} \bullet \nabla e = \frac{\partial e}{\partial t} + \nabla \bullet (\mathbf{U} e) - (\nabla \bullet \mathbf{U}) e \tag{4.1}$$

For numerical solution of incompressible flows, $\nabla \bullet \mathbf{U} = 0$ at convergence, at which point the third term on the right hand side is zero. Before convergence is reached, however, $\nabla \bullet \mathbf{U} \neq 0$ and in some circumstances, particularly steady-state simulations, it is better to include the third term within a numerical solution because it helps maintain boundedness of the solution variable and promotes better convergence. The `bounded` variant of the `Gauss` scheme provides this, automatically including the discretisation of the third-term with the advection term. Example syntax is as follows, as seen in *fvSchemes* files for steady-state cases, *e.g.* for the simpleFoam tutorials

```
div(phi,U)      bounded Gauss limitedLinearV 1;
div(phi,U)      bounded Gauss linearUpwindV grad(U);
```

The schemes used for advection of scalar fields are similar to those for advection of velocity, although in general there is greater emphasis placed on boundedness than accuracy when selecting the schemes. For example, a search for schemes for advection of internal energy (`e`) reveals the following.

```
foamSearch $FOAM_TUTORIALS fvSchemes "divSchemes.div(phi,e)"

div(phi,e)      bounded Gauss upwind;
div(phi,e)      Gauss limitedLinear 1;
div(phi,e)      Gauss LUST grad(e);
div(phi,e)      Gauss upwind;
div(phi,e)      Gauss vanLeer;
```

In comparison with advection of velocity, there are no cases set up to use `linear` or `linearUpwind`. Instead the `limitedLinear` and `upwind` schemes are commonly used, with the additional appearance of `vanLeer`, another limited scheme, with less strong limiting than `limitedLinear`.

There are specialised versions of the limited schemes for scalar fields that are commonly bounded between 0 and 1, *e.g.* the laminar flame speed regress variable b. A search for the discretisation used for advection in the laminar flame transport equation yields:

```
div(phiSt,b)    Gauss limitedLinear01 1;
```

The underlying scheme is `limitedLinear`, specialised for stronger bounding between 0 and 1 by adding `01` to the name of the scheme.

The `multivariateSelection` mechanism also exists for grouping multiple equation terms together, and applying the same limiters on all terms, using the strongest limiter calculated for all terms. A good example of this is in a set of mass transport equations for fluid species, where it is good practice to apply the same discretisation to all equations for consistency. The example below comes from the smallPoolFire3D tutorial in *$FOAM_TUTORIALS/combustion/fireFoam/les*, in which the equation for enthalpy h is included with the specie mass transport equations in the calculation of a single limiter.

```
div(phi,Yi_h)   Gauss multivariateSelection
{
    O2 limitedLinear01 1;
    CH4 limitedLinear01 1;
    N2 limitedLinear01 1;
    H2O limitedLinear01 1;
    CO2 limitedLinear01 1;
    h limitedLinear 1 ;
}
```

4.5.4 Surface normal gradient schemes

It is worth explaining the *snGradSchemes* sub-dictionary that contains surface normal gradient terms, before discussion of *laplacianSchemes*, because they are required to evaluate a Laplacian term using Gaussian integration. A surface normal gradient is evaluated at a cell face; it is the component, normal to the face, of the gradient of values at the centres of the 2 cells that the face connects.

A search for the `default` scheme for *snGradSchemes* reveals the following entries.

```
default         corrected;
default         limited corrected 0.33;
default         limited corrected 0.5;
default         orthogonal;
default         uncorrected;
```

The basis of the gradient calculation at a face is to subtract the value at the cell centre on one side of the face from the value in the centre on the other side and divide by the distance. The calculation is second-order accurate for the gradient *normal to the face* if the vector connecting the cell centres is orthogonal to the face, *i.e.* they are at right-angles. This is the `orthogonal` scheme.

Orthogonality requires a regular mesh, typically aligned with the Cartesian co-ordinate system, which does not normally occur in meshes for real world, engineering geometries. Therefore, to maintain second-order accuracy, an explicit non-orthogonal correction can be added to the orthogonal component, known as the `corrected` scheme. The correction increases in size as the non-orthogonality, the angle α between the cell-cell vector and face normal vector, increases.

As α tends towards 90°, *e.g.* beyond 70°, the explicit correction can be so large to cause a solution to go unstable. The solution can be stabilised by applying the `limited` scheme to the correction which requires a coefficient $\psi, 0 \leq \psi \leq 1$ where

$$\psi = \begin{cases} 0 & \text{corresponds to } \texttt{uncorrected}, \\ 0.333 & \text{non-orthogonal correction} \leq 0.5 \times \text{orthogonal part}, \\ 0.5 & \text{non-orthogonal correction} \leq \text{orthogonal part}, \\ 1 & \text{corresponds to } \texttt{corrected}. \end{cases} \tag{4.2}$$

Typically, psi is chosen to be 0.33 or 0.5, where 0.33 offers greater stability and 0.5 greater accuracy.

The corrected scheme applies under-relaxation in which the implicit orthogonal calculation is increased by $cos^{-1}\alpha$, with an equivalent boost within the non-orthogonal correction. The `uncorrected` scheme is equivalent to the `corrected` scheme, without the non-orthogonal correction, so includes is like `orthogonal` but with the $cos^{-1}\alpha$ under-relaxation.

Generally the `uncorrected` and `orthogonal` schemes are only recommended for meshes with very low non-orthogonality (*e.g.* maximum 5°). The `corrected` scheme is generally recommended, but for maximum non-orthogonality above 70°, `limited` may be required. At non-orthogonality above 80°, convergence is generally hard to achieve.

4.5.5 Laplacian schemes

The *laplacianSchemes* sub-dictionary contains Laplacian terms. A typical Laplacian term is $\nabla \bullet (\nu \nabla \mathbf{U})$, the diffusion term in the momentum equations, which corresponds to the keyword `laplacian(nu,U)` in *laplacianSchemes*. The `Gauss` scheme is the only choice of discretisation and requires a selection of both an interpolation scheme for the diffusion coefficient, *i.e.* ν in our example, and a surface normal gradient scheme, *i.e.* $\nabla \mathbf{U}$. To summarise, the entries required are:

```
Gauss <interpolationScheme> <snGradScheme>
```

The user can search for the `default` scheme for *laplacianSchemes* in all the cases in the *$FOAM_TUTORIALS* directory.

```
foamSearch $FOAM_TUTORIALS fvSchemes laplacianSchemes.default
```

It reveals the following entries.

```
default         Gauss linear corrected;
default         Gauss linear limited corrected 0.33;
default         Gauss linear limited corrected 0.5;
default         Gauss linear orthogonal;
default         Gauss linear uncorrected;
```

In all cases, the `linear` interpolation scheme is used for interpolation of the diffusivity. The cases uses the same array of `snGradSchemes` based on level on non-orthogonality, as described in section 4.5.4.

4.5.6 Interpolation schemes

The *interpolationSchemes* sub-dictionary contains terms that are interpolations of values typically from cell centres to face centres, primarily used in the interpolation of velocity to face centres for the calculation of flux `phi`. There are numerous interpolation schemes in OpenFOAM, but a search for the `default` scheme in all the tutorial cases reveals that `linear` interpolation is used in almost every case, except for 2-3 unusual cases, *e.g.* DNS on a regular mesh, stress analysis, where `cubic` interpolation is used.

4.6 Solution and algorithm control

The equation solvers, tolerances and algorithms are controlled from the *fvSolution* dictionary in the *system* directory. Below is an example set of entries from the *fvSolution* dictionary required for the icoFoam solver.

```

solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.05;
    }

    pFinal
    {
        $p;
        relTol          0;
    }

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-05;
        relTol          0;
    }
}

PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}


// ************************************************************************* //
```

fvSolution contains a set of subdictionaries, described in the remainder of this section that includes: *solvers*; *relaxationFactors*; and, *PISO*, *SIMPLE* or *PIMPLE*.

4.6.1 Linear solver control

The first sub-dictionary in our example is `solvers`. It specifies each linear-solver that is used for each discretised equation; here, the term *linear*-solver refers to the method of number-crunching to solve a matrix equation, as opposed to an *application* solver, such as simpleFoam which describes the entire set of equations and algorithms to solve a particular problem. The term 'linear-solver' is abbreviated to 'solver' in much of what follows; hopefully the context of the term avoids any ambiguity.