English

OpenFOAM

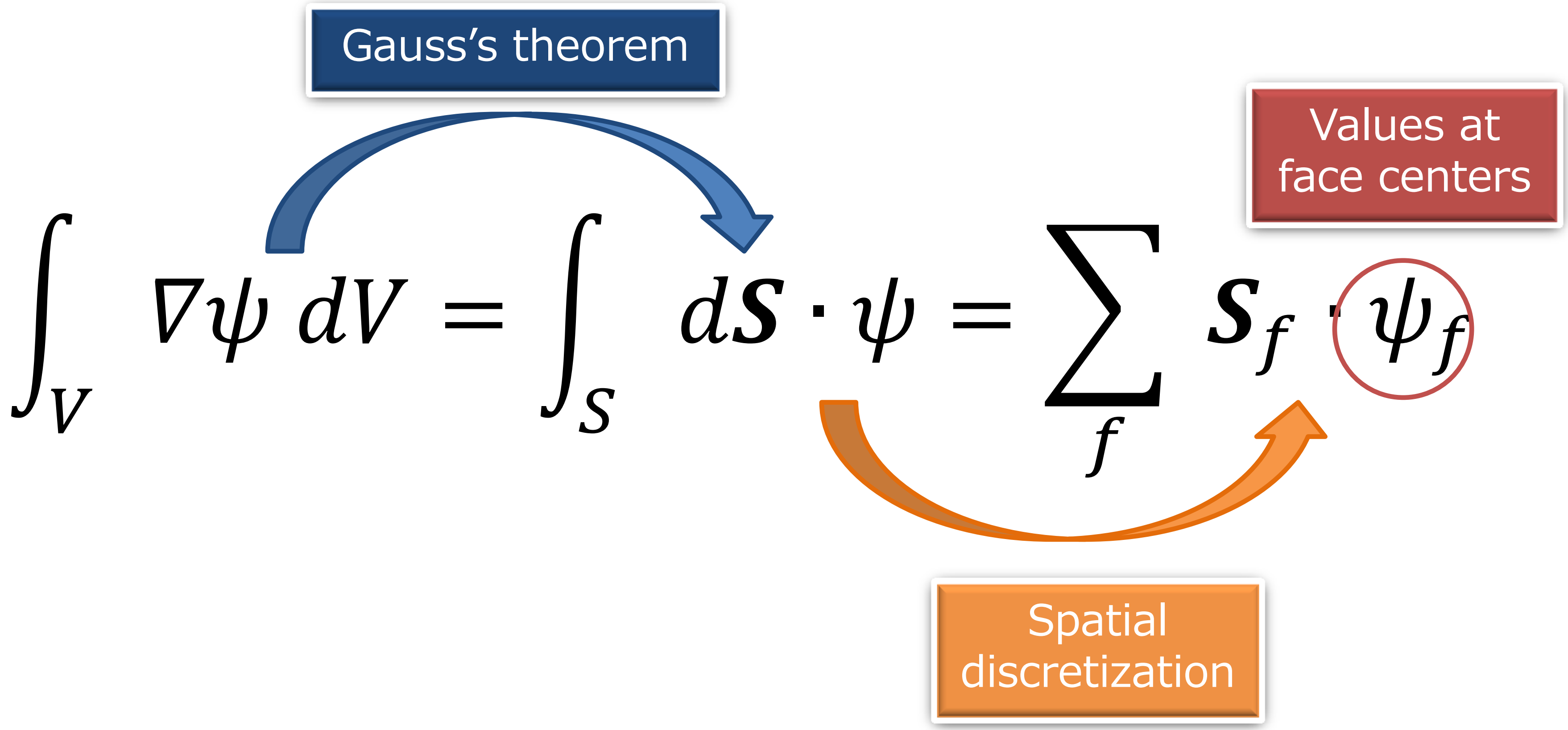
Spatial Interpolation

Fumiya Nozaki

Last Updated: 16 June 2014

FVM discretization

- Most spatial derivative terms are first integrated over a cell volume V and then converted to integrals over the cell surface bounding the volume using **Gauss’s theorem**



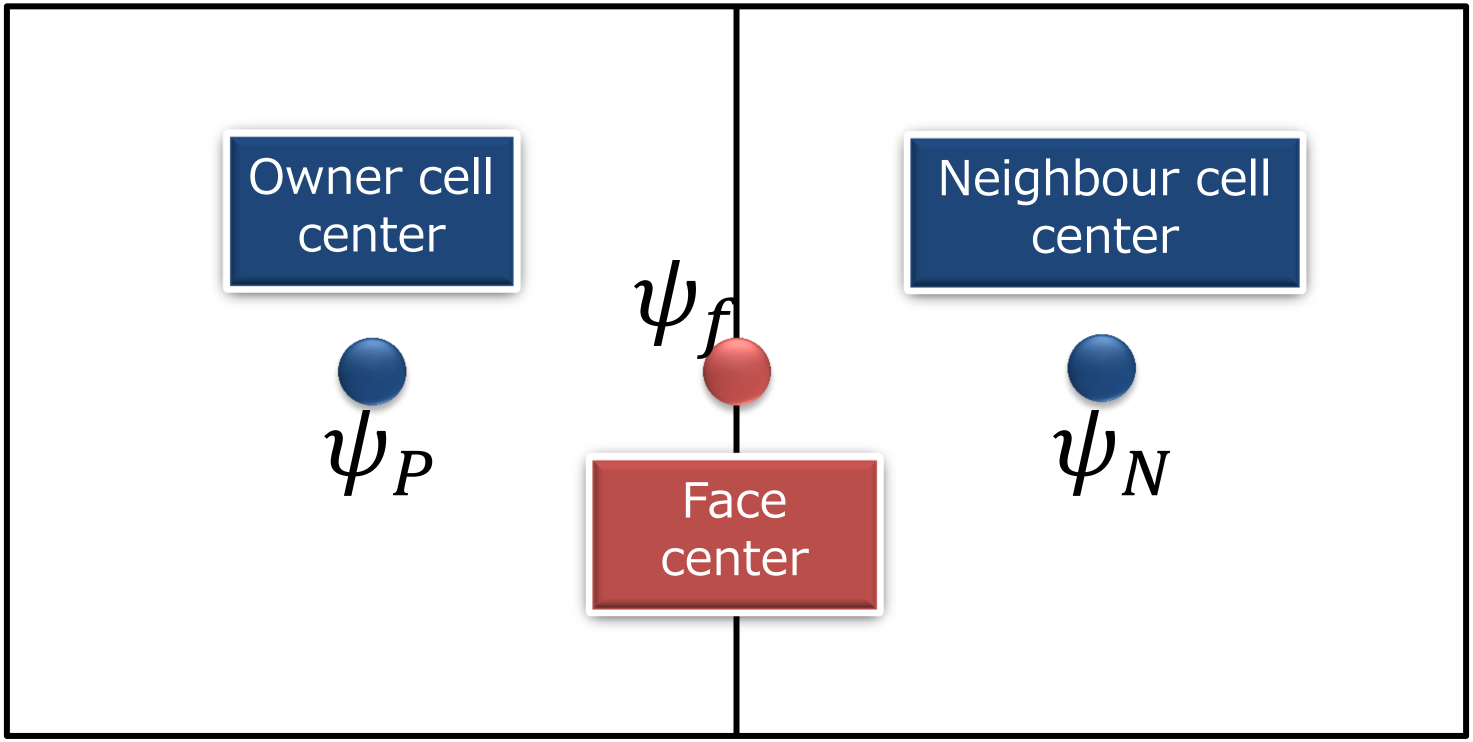
Gauss’s theorem

Values at face centers

$$\int_V \nabla\psi \, dV = \int_S d\boldsymbol{S}\cdot\psi = \sum_f \boldsymbol{S}_f\cdot\psi_f$$

Spatial discretization

Interpolation from cells to faces



- What we need is some algebraic relational expressions

$$\psi_f = f(\psi_P, \psi_N)$$

Available interpolation schemes in OpenFOAM

- Typical interpolation schemes in OpenFOAM

$$\psi_f = f\,(\psi_P, \psi_N)$$

- ✓ upwind
- ✓ linearUpwind
- ✓ linear
- ✓ limitedLinear

etc.

Many other schemes are available for use
in the spatial interpolation [1].

Specification of interpolation schemes

- Interpolation schemes are chosen on a term-by-term basis

```
gradSchemes
{
    default Gauss linear;
}

divSchemes
{
    default none;
    div(phi,U) bounded Gauss linearUpwind grad(U);
    div((nuEff*(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default Gauss linear corrected;
}

interpolationSchemes
{
    default linear;
}
```

Specification of interpolation schemes

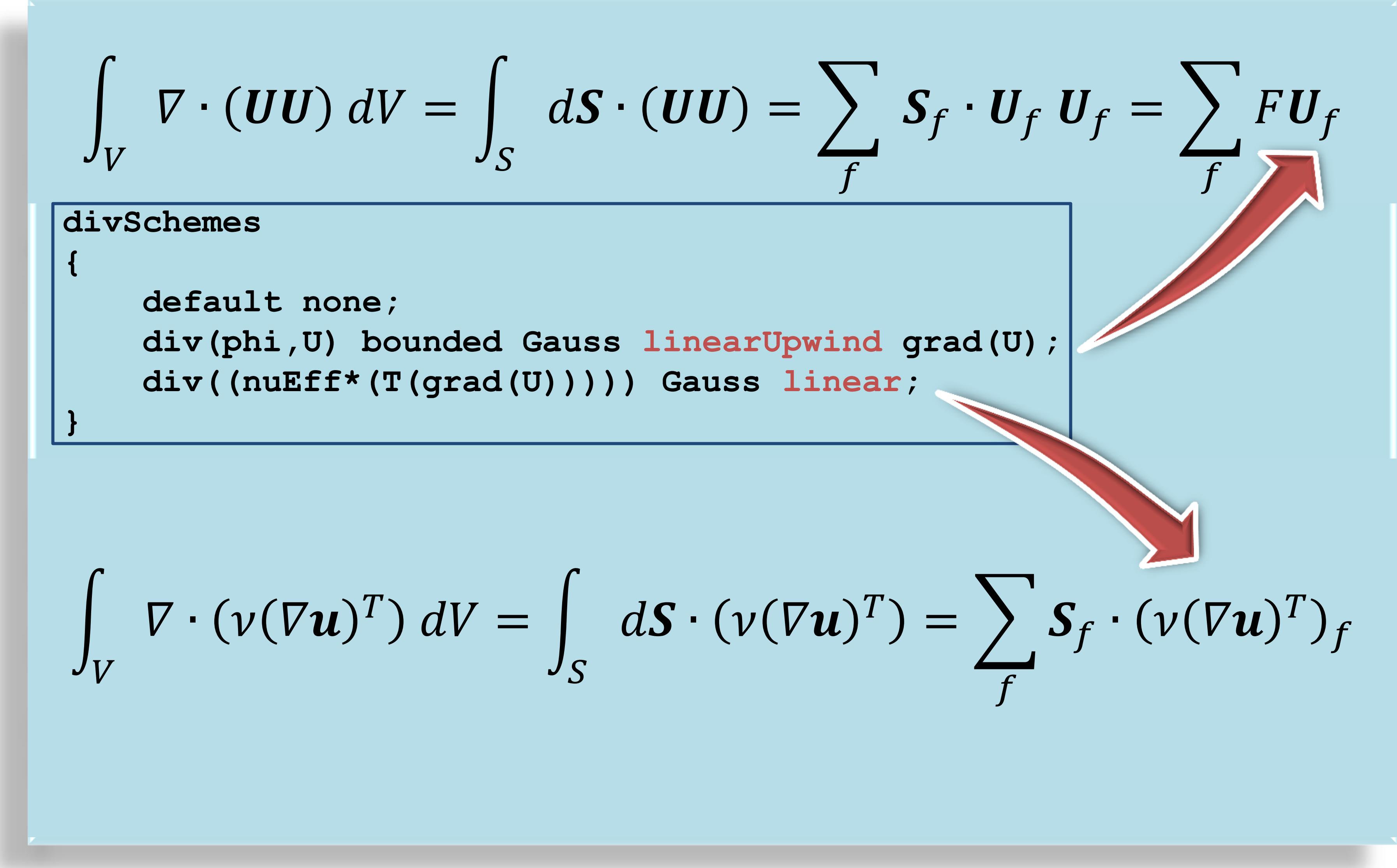
- Interpolation schemes are chosen on a term-by-term basis

```
gradSchemes
{
    default Gauss linear;
}
```

$$\int_V \nabla p \, dV = \int_S d\boldsymbol{S} \cdot p = \sum_f \boldsymbol{S}_f \cdot p_f$$

Specification of interpolation schemes

- Interpolation schemes are chosen on a term-by-term basis



$$\int_V \nabla\cdot(\mathbf{U}\mathbf{U})\,dV = \int_S d\mathbf{S}\cdot(\mathbf{U}\mathbf{U}) = \sum_f \mathbf{S}_f\cdot\mathbf{U}_f\,\mathbf{U}_f = \sum_f F\,\mathbf{U}_f$$

```
divSchemes
{
    default none;
    div(phi,U) bounded Gauss linearUpwind grad(U);
    div((nuEff*(T(grad(U))))) Gauss linear;
}
```

$$\int_V \nabla\cdot(\nu(\nabla\mathbf{u})^T)\,dV = \int_S d\mathbf{S}\cdot(\nu(\nabla\mathbf{u})^T) = \sum_f \mathbf{S}_f\cdot(\nu(\nabla\mathbf{u})^T)_f$$

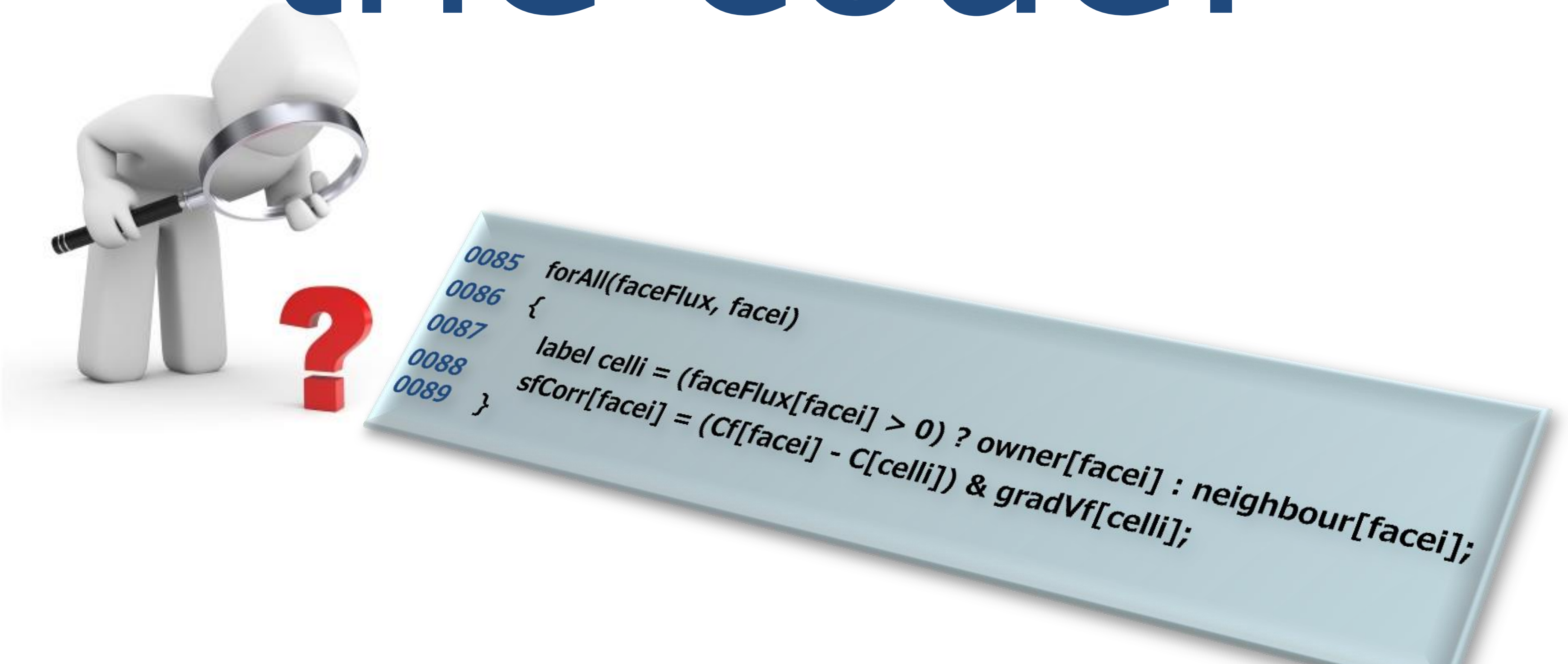
Specification of interpolation schemes

- Interpolation schemes are chosen on a term-by-term basis

$$\int_V \nabla\cdot(\Gamma\nabla\phi)\,dV = \int_S d\boldsymbol{S}\cdot(\Gamma\nabla\phi) = \sum_f \Gamma_f\ \boldsymbol{S}_f\cdot(\nabla\phi)_f$$

```
laplacianSchemes
{
    default Gauss linear corrected;
}
```

Let’s look into the code!



```
forAll(faceFlux, facei)
{
   label celli = (faceFlux[facei] > 0) ? owner[facei] : neighbour[facei];
   sfCorr[facei] = (Cf[facei] - C[celli]) & gradVf[celli];
}
```

surfaceInterpolationScheme.C

surfaceInterpolationScheme.C

```
//- Return the face-interpolate of the given cell field
// with explicit correction
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
surfaceInterpolationScheme<Type>::interpolate
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    if (surfaceInterpolation::debug)
    {
        Info<< "surfaceInterpolationScheme<Type>::interpolate"
               "(const GeometricField<Type, fvPatchField, volMesh>&) : "
               "interpolating "
            << vf.type() << " "
            << vf.name()
            << " from cells to faces"
            << endl;
    }

    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tsf
        = interpolate(vf, weights(vf));

    if (corrected())
    {
        tsf() += correction(vf);
    }

    return tsf;
}
```

Without explicit correction

Addition of explicit correction

surfaceInterpolationScheme.C

surfaceInterpolationScheme.C

```
//- Return the face-interpolate of the given cell field
// with explicit correction
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
surfaceInterpolationScheme<Type>::interpolate
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    if (surfaceInterpolation::debug)
    {
        Info<< "surfaceInterpolationScheme<Type>::interpolate"
               "(const GeometricField<Type, fvPatchField, volMesh>&) : "
               "interpolating "
            << vf.type() << " "
            << vf.name()
            << " from cells to faces"
            << endl;
    }

    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tsf
        = interpolate(vf, weights(vf));

    if (corrected())
    {
        tsf() += correction(vf);
    }

    return tsf;
}
```

“weights()” are different depending on the interpolation scheme.

surfaceInterpolationScheme.C

surfaceInterpolationScheme.C

```
//- Return the face-interpolate of the given cell field
// with explicit correction
template<class Type>
tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >
surfaceInterpolationScheme<Type>::interpolate
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
) const
{
    if (surfaceInterpolation::debug)
    {
        Info<< "surfaceInterpolationScheme<Type>::interpolate"
               "(const GeometricField<Type, fvPatchField, volMesh>&) : "
               "interpolating "
            << vf.type() << " "
            << vf.name()
            << " from cells to faces"
            << endl;
    }

    tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tsf
        = interpolate(vf, weights(vf));

    if (corrected())
    {
        tsf() += correction(vf);
    }

    return tsf;
}
```

“interpolate()” calculates interpolated face value without explicit correction.

Turn to the next page.

surfaceInterpolationScheme.C

surfaceInterpolationScheme.C

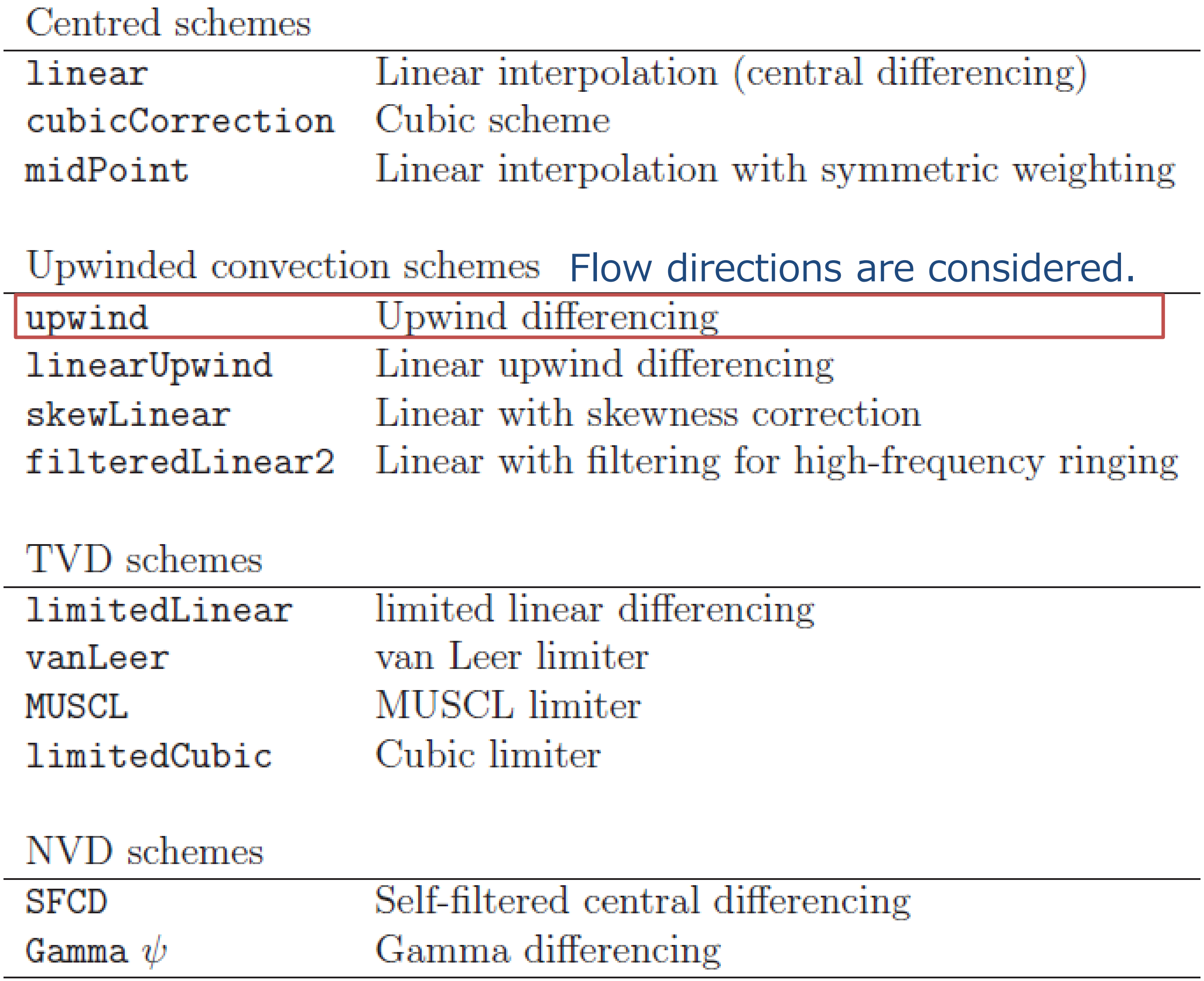
```
const surfaceScalarField& lambdas = tlambdas();

const Field<Type>& vfi = vf.internalField();
const scalarField& lambda = lambdas.internalField();

const fvMesh& mesh = vf.mesh();
const labelUList& P = mesh.owner();
const labelUList& N = mesh.neighbour();

tmp<GeometricField<Type, fvsPatchField, surfaceMesh> > tsf
(
    new GeometricField<Type, fvsPatchField, surfaceMesh>
    (
        IOobject
        (
            "interpolate("+vf.name()+')',
            vf.instance(),
            vf.db()
        ),
        mesh,
        vf.dimensions()
    )
);
GeometricField<Type, fvsPatchField, surfaceMesh>& sf = tsf();

Field<Type>& sfi = sf.internalField();

for (register label fi=0; fi<P.size(); fi++)
{
    sfi[fi] = lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]];
}
```

sfi[fi] represents the interpolated value at the fi-th face center.

upwind

First look into “upwind” scheme

Table 4.6: Interpolation schemes [1]



Flow directions are considered.

Centred schemes	
`linear`	Linear interpolation (central differencing)
`cubicCorrection`	Cubic scheme
`midPoint`	Linear interpolation with symmetric weighting
Upwinded convection schemes	
`upwind`	Upwind differencing
`linearUpwind`	Linear upwind differencing
`skewLinear`	Linear with skewness correction
`filteredLinear2`	Linear with filtering for high-frequency ringing
TVD schemes	
`limitedLinear`	limited linear differencing
`vanLeer`	van Leer limiter
`MUSCL`	MUSCL limiter
`limitedCubic`	Cubic limiter
NVD schemes	
`SFCD`	Self-filtered central differencing
`Gamma` ψ	Gamma differencing

Calculation of weights

upwind.H

```
0128    //- Return the interpolation weighting factors
0129    tmp<surfaceScalarField> weights() const
0130    {
0131        return pos(this->faceFlux_);
0132    }
0133
0134    //- Return the interpolation weighting factors
0135    virtual tmp<surfaceScalarField> weights
0136    (
0137        const GeometricField<Type, fvPatchField, volMesh>&
0138    ) const
0139    {
0140        return weights();
0141    }
```

- *l. 0131*

$$weights()[\mathrm{facei}] = \begin{cases} 1 & \text{if } faceFlux[\mathrm{facei}] > 0 \\ 0 & \text{if } faceFlux[\mathrm{facei}] \leq 0 \end{cases}$$

upwind

Explicit correction

- For “upwind” scheme, corrected() returns “false”

```
//- Return true if this scheme uses an explicit correction
virtual bool corrected() const
{
    return false;
}
```

surfaceInterpolationScheme.H

So, “upwind” interpolation has **no** explicit correction.

- Evaluation of the interpolated value

  - If phi[facei] > 0

```
sfi[fi] = lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = weights()[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = 1*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = vfi[P[fi]] = owner cell value
```

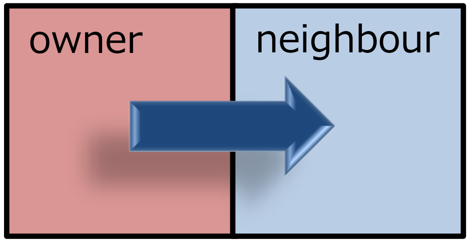
  - If phi[facei] ≦ 0

```
sfi[fi] = lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = 0*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = vfi[N[fi]] = neighbour cell value
```

upwind

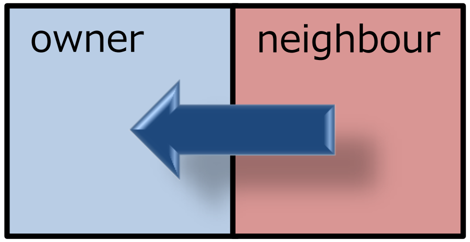
Interpolated value

phi[facei] > 0



owner is the upstream cell

phi[facei] ≦ 0



neighbour is the upstream cell

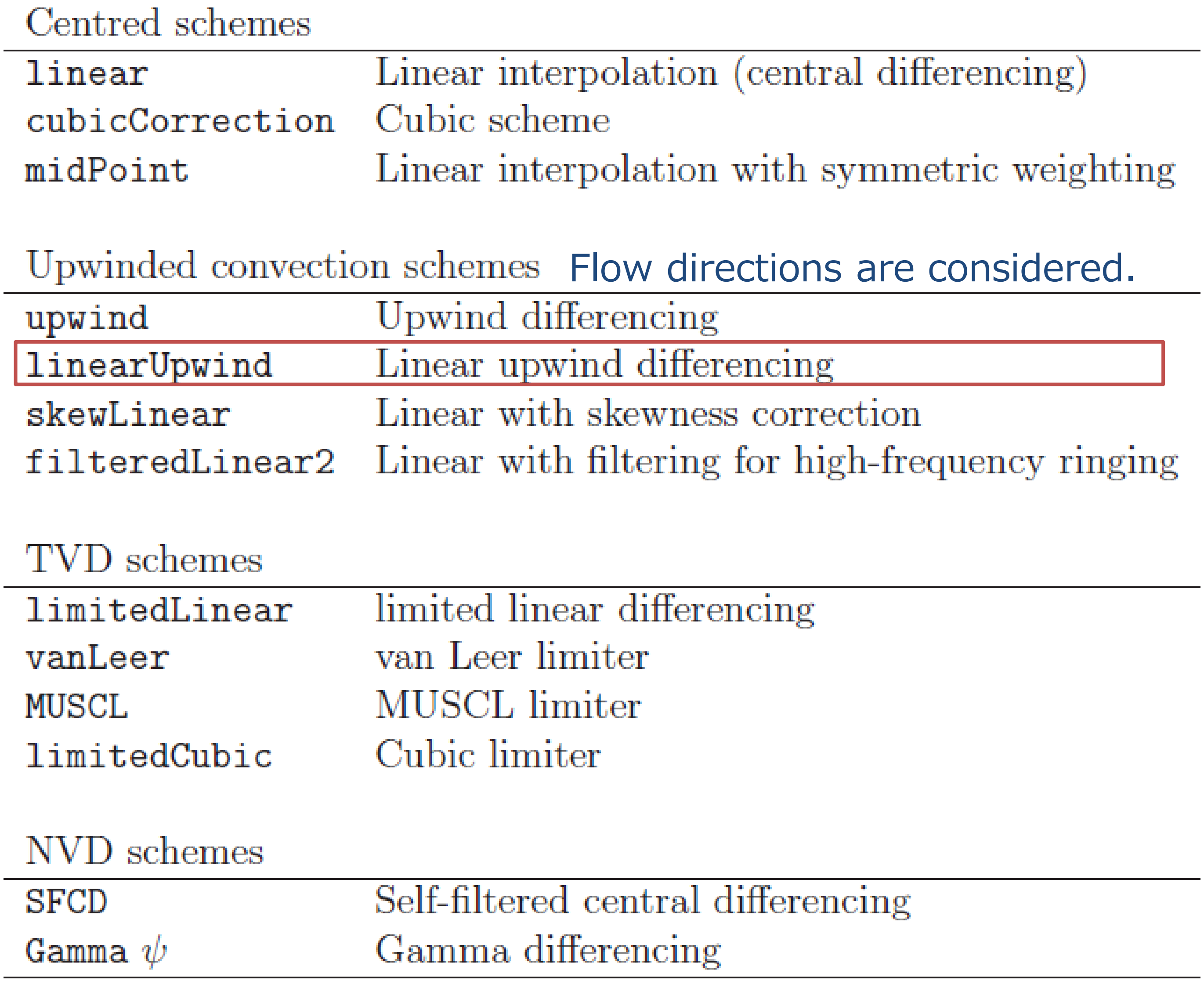
$$\psi_f\,[\mathbf{facei}] = \psi\,[\boldsymbol{owner\ cell\ of}\ \mathbf{facei}]$$

$$\psi_f\,[\mathbf{facei}] = \psi\,[\boldsymbol{neighbour\ cell\ of}\ \mathbf{facei}]$$

linearUpwind

Linear upwind differencing

Table 4.6: Interpolation schemes [1]



Centred schemes	
`linear`	Linear interpolation (central differencing)
`cubicCorrection`	Cubic scheme
`midPoint`	Linear interpolation with symmetric weighting

Upwinded convection schemes	
`upwind`	Upwind differencing
`linearUpwind`	Linear upwind differencing
`skewLinear`	Linear with skewness correction
`filteredLinear2`	Linear with filtering for high-frequency ringing

Flow directions are considered.

TVD schemes	
`limitedLinear`	limited linear differencing
`vanLeer`	van Leer limiter
`MUSCL`	MUSCL limiter
`limitedCubic`	Cubic limiter

NVD schemes	
`SFCD`	Self-filtered central differencing
`Gamma` ψ	Gamma differencing

Explicit correction

- “weights” are same as that of “upwind” scheme
- For “linearUpwind” scheme, corrected() returns “true”

```
//- Return true if this scheme uses an explicit correction
virtual bool corrected() const
{
    return true;
}
```

linearUpwind.H

- Calculation of explicit correction term

linearUpwind.C

```
forAll(faceFlux, facei)
{
    label celli = (faceFlux[facei] > 0) ? owner[facei] : neighbour[facei];
    sfCorr[facei] = (Cf[facei] - C[celli]) & gradVf[celli];
}
```

label of an upstream cell

- Upstream cell is judged by the sign of the face flux field “phi” (**faceFlux means phi in the above code**)

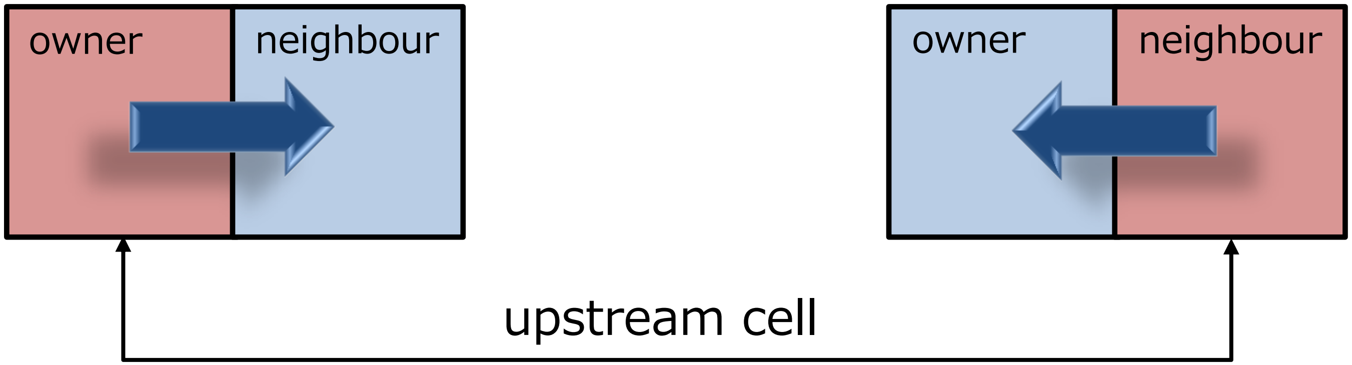
$$\boldsymbol{r} \cdot \nabla\psi\ [upstream\ cell]$$

linearUpwind

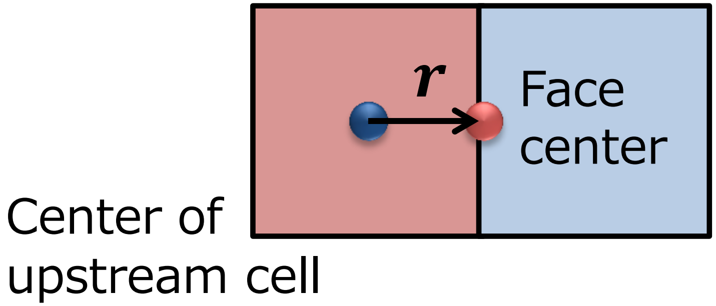
Interpolated value

phi[facei] > 0

phi[facei] ≦ 0



$$\psi_f\,[\mathbf{facei}] = \psi\,[\boldsymbol{upstream\ cell\ of}\ \mathbf{facei}] + \boldsymbol{r}\cdot\nabla\psi\,[\boldsymbol{upstream\ cell\ of}\ \mathbf{facei}]$$



Discretization of gradient

$$\begin{aligned}\boldsymbol{\psi_f}\,[\mathbf{facei}] &= \boldsymbol{\psi}\,[\boldsymbol{upstream\ cell\ of}\ \mathbf{facei}] \\ &\quad + \boldsymbol{r}\cdot\boxed{\boldsymbol{\nabla\psi}\,[\boldsymbol{upstream\ cell\ of}\ \mathbf{facei}]}\end{aligned}$$

- Specification of the discretization scheme of gradient term ∇p

"fvSchemes" file

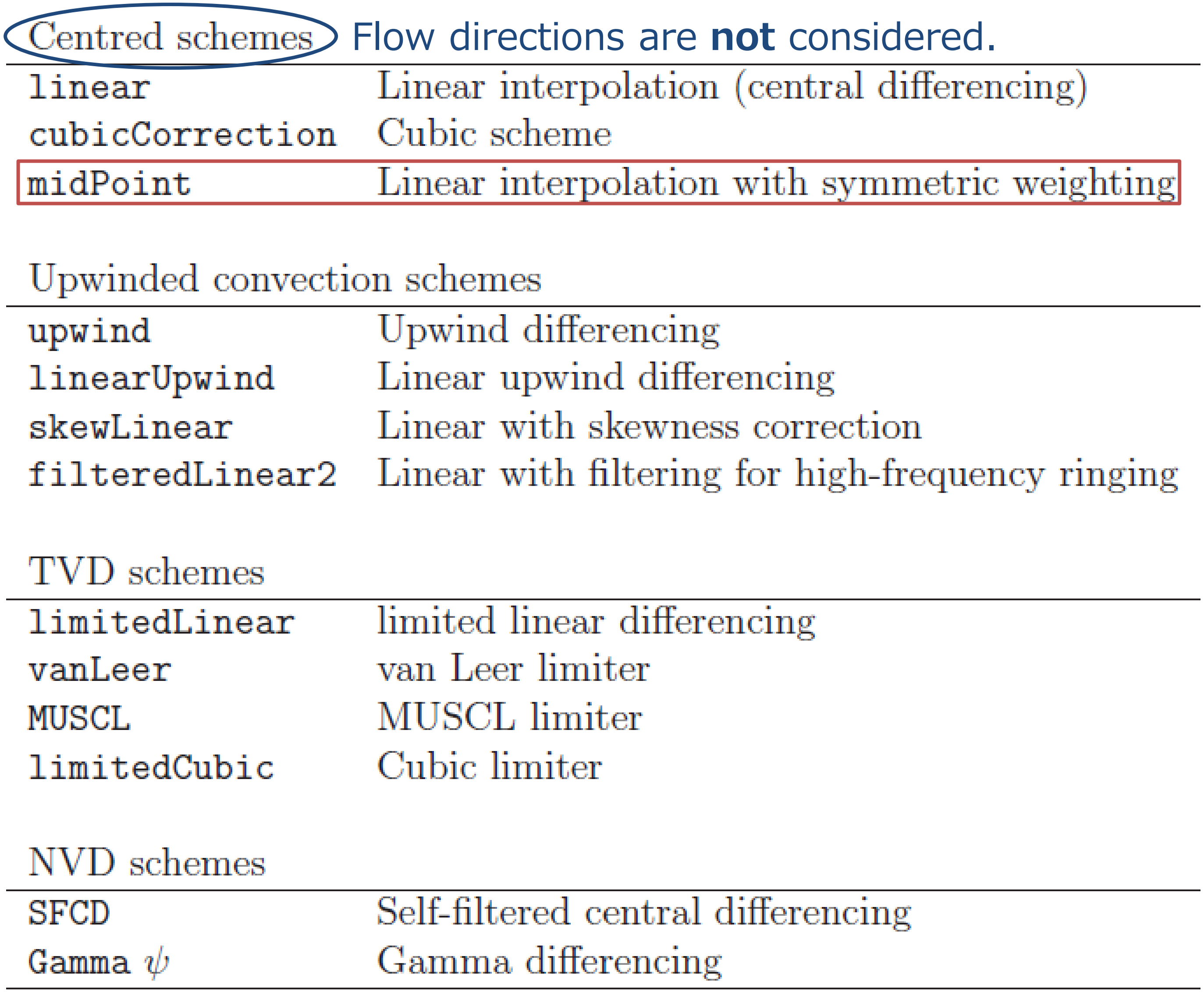
```
gradSchemes
{
    grad(psi)           Gauss linear;
}

interpolationSchemes
{
    interpolate(psi)  linearUpwind phi grad(psi);
}
```

The same strings have to be specified.

midPoint

Linear interpolation with symmetric weighting

Table 4.6: Interpolation schemes [1]



Flow directions are **not** considered.

Centred schemes	
`linear`	Linear interpolation (central differencing)
`cubicCorrection`	Cubic scheme
`midPoint`	Linear interpolation with symmetric weighting
Upwinded convection schemes	
`upwind`	Upwind differencing
`linearUpwind`	Linear upwind differencing
`skewLinear`	Linear with skewness correction
`filteredLinear2`	Linear with filtering for high-frequency ringing
TVD schemes	
`limitedLinear`	limited linear differencing
`vanLeer`	van Leer limiter
`MUSCL`	MUSCL limiter
`limitedCubic`	Cubic limiter
NVD schemes	
`SFCD`	Self-filtered central differencing
`Gamma` ψ	Gamma differencing

midPoint

Calculation of weights

midPoint.H

```
//- Return the interpolation weighting factors
tmp<surfaceScalarField> weights
(
    const GeometricField<Type, fvPatchField, volMesh>&
) const
{
    tmp<surfaceScalarField> taw
    (
        new surfaceScalarField
        (
            IOobject
            (
                "midPointWeights",
                this->mesh().time().timeName(),
                this->mesh()
            ),
            this->mesh(),
            dimensionedScalar("0.5", dimless, 0.5)
        )
    );

    surfaceScalarField::GeometricBoundaryField& awbf =
        taw().boundaryField();

    forAll(awbf, patchi)
    {
        if (!awbf[patchi].coupled())
        {
            awbf[patchi] = 1.0;
        }
    }

    return taw;
}
```

weights on the internal faces are 0.5

Explicit correction

- For “midPoint” scheme, corrected() returns “false”

```
//- Return true if this scheme uses an explicit correction
virtual bool corrected() const
{
    return false;
}
```

surfaceInterpolationScheme.H

So, “midPoint” interpolation has **no** explicit correction.

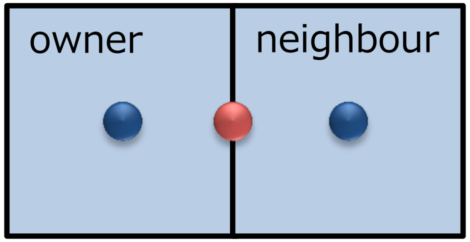
- Evaluation of the interpolated value

```
sfi[fi] = lambda[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = weights()[fi]*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = 0.5*(vfi[P[fi]] - vfi[N[fi]]) + vfi[N[fi]]
        = 0.5*(vfi[P[fi]] + vfi[N[fi]])
        = 0.5*(owner cell value + neighbour cell value)
```

Arithmetic mean

midPoint

Interpolated value



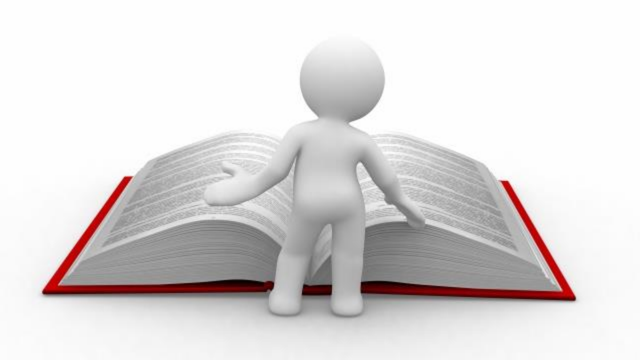
Arithmetic mean

$$\boldsymbol{\psi_f}\ [\mathbf{facei}] =$$

$$\frac{\mathbf{1}}{\mathbf{2}}(\boldsymbol{\psi}\ [\boldsymbol{owner\ cell\ of}\ \mathbf{facei}] + \ \boldsymbol{\psi}\ [\boldsymbol{neighbour\ cell\ of}\ \mathbf{facei}])$$

References

[1] User Guide http://foam.sourceforge.net/docs/Guides-a4/UserGuide.pdf
(accessed 06/15/2014)



Thank You!