IFEM - getting started

Knut Morten Okstad

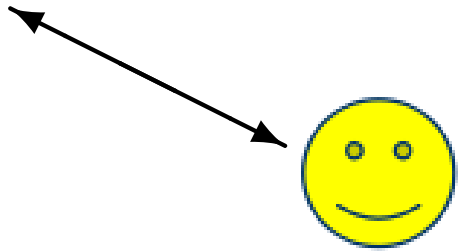
SINTEF ICT, Department of Applied Mathematics

January 7, 2016

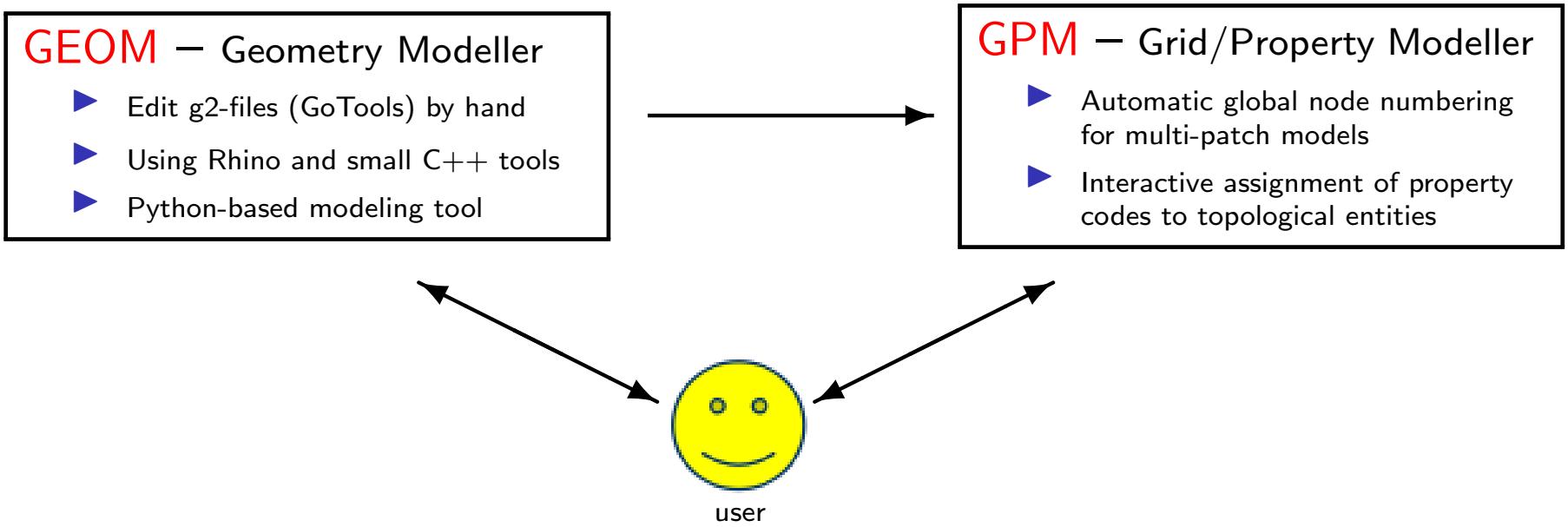
IFEM module overview

GEOM – Geometry Modeller

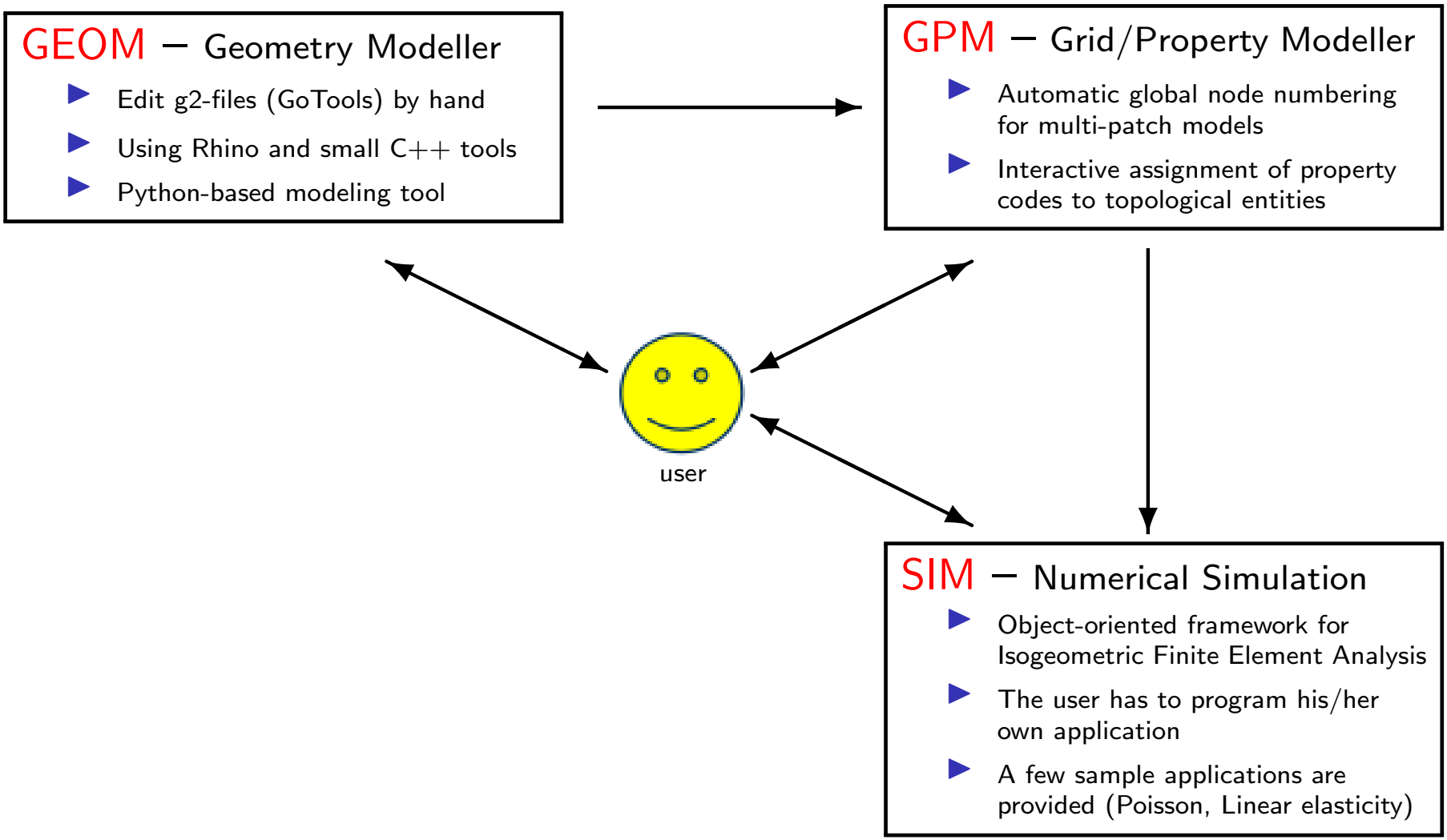
- Edit g2-files (GoTools) by hand
- Using Rhino and small C++ tools
- Python-based modeling tool



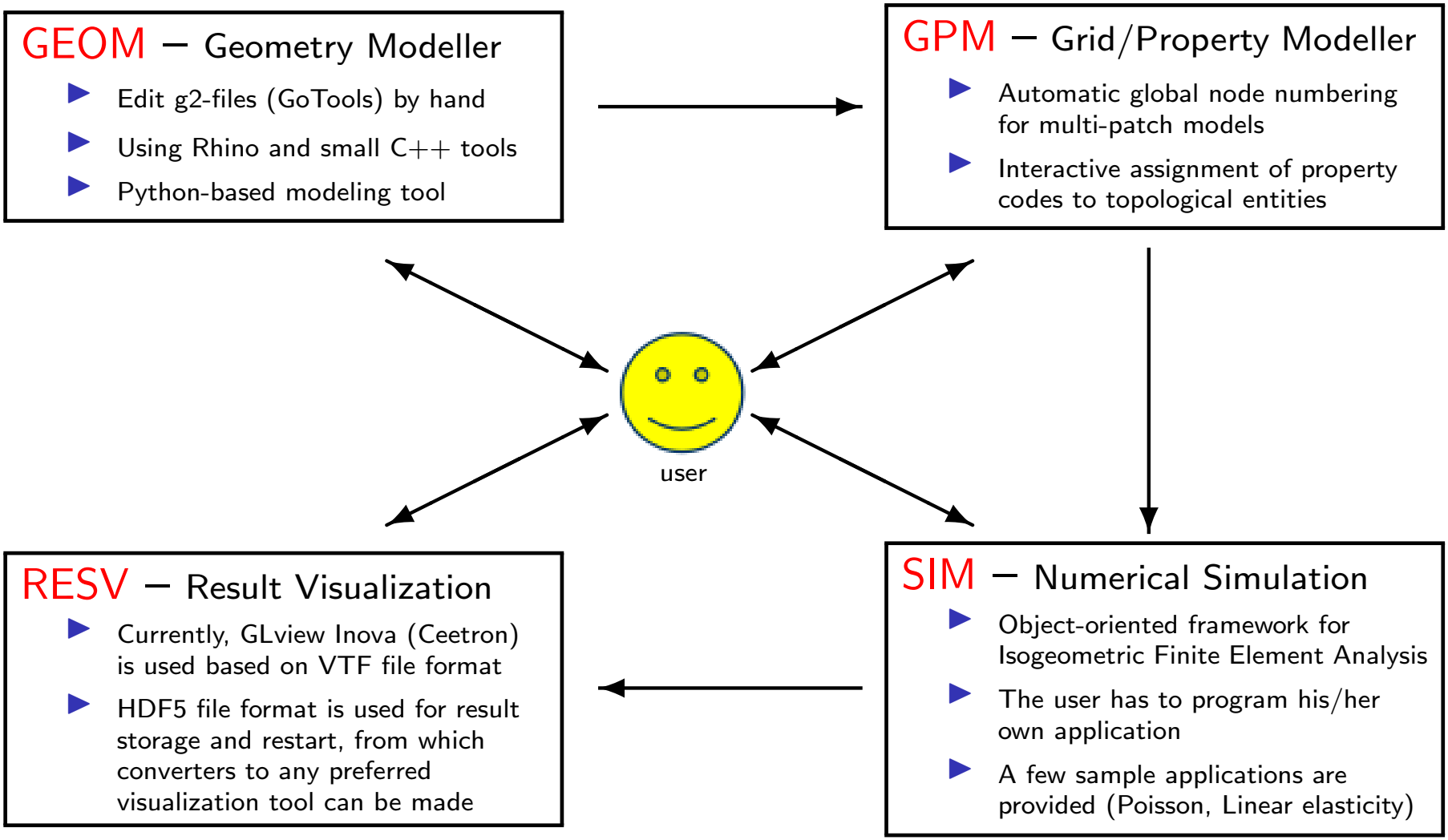
IFEM module overview



IFEM module overview



IFEM module overview



Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages
3. SIMbase - System level drivers
   - Administering an assembly of spline patches (blocks)
   - Sub-classes for problem-specific input and setup

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages
3. SIMbase - System level drivers
   - Administering an assembly of spline patches (blocks)
   - Sub-classes for problem-specific input and setup
4. ASMbase - Block/patch level drivers
   - Administers the element loop and numerical integration loop within a block (spline patch)
   - Sub-classes depending on discretization (Splines/NURBS, Lagrange, Spectral)
   - Uses *GoTools* to evaluate basis functions at integration points

     http://www.sintef.no/Projectweb/Geometry-Toolkits/GoTools

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages
3. SIMbase - System level drivers
   - Administering an assembly of spline patches (blocks)
   - Sub-classes for problem-specific input and setup
4. ASMbase - Block/patch level drivers
   - Administers the element loop and numerical integration loop within a block (spline patch)
   - Sub-classes depending on discretization (Splines/NURBS, Lagrange, Spectral)
   - Uses *GoTools* to evaluate basis functions at integration points

     http://www.sintef.no/Projectweb/Geometry-Toolkits/GoTools
5. Integrand - Integration point level
   - Administers the problem-dependent calculations at an integration point (interior and boundary integrals)
   - Problem-specific sub-classes

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages
3. SIMbase - System level drivers
   - Administering an assembly of spline patches (blocks)
   - Sub-classes for problem-specific input and setup
4. ASMbase - Block/patch level drivers
   - Administers the element loop and numerical integration loop wit
   - Su lines/NURBS, La
   - Uses *GoTools* to evaluate basis functions at integration points

     http://www.sintef.no/Projectweb/Geometry-Toolkits/GoTools
5. Integrand - Integration point level
   - Administers the problem-dependent calculations at an integration point (interior and boundary integrals)
   - Problem-specific sub-classes

Isogeometric level

Major class hierarchies of the SIMulation environment

1. NonLinSIM - Nonlinear simulation driver
   - Administers time/load step loop of the solution algorithm
   - Newton iteration loop, convergence check, configuration update
2. SystemMatrix/Vector - Linear algebra system level
   - Interface to equation solvers (direct/iterative, serial/parallel)
   - Sub-classes for various linear equation solver packages
3. SIMbase - System level drivers
   - Administering an assembly of spline patches (blocks)
   - Sub-classes for problem-specific input and setup
4. ASMbase - Block/patch level drivers
   - Administers the element loop and numerical integration loop
   - Su lines/NURBS, La
   - Uses *GoTools* to evaluate basis functions at integration points

     http://www.sintef.no/Projectweb/Geometry-Toolkits/GoTools
5. Integrand - Integration point level
   - Ad n int
   - Pr

Isogeometric level

User/Application level

ASM class hierarchy - *the Isogeometry level*

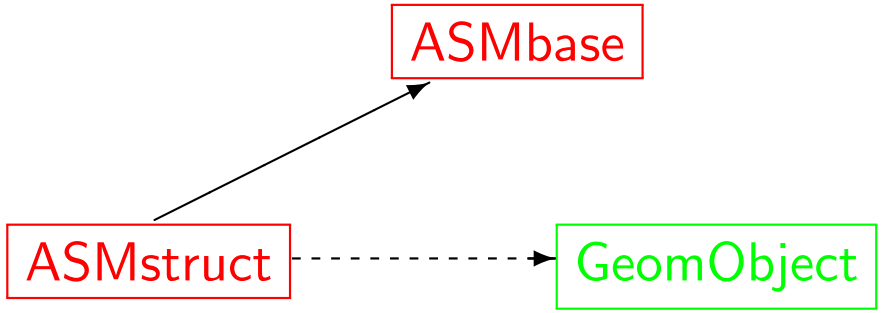
ASMbase

→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

ASM class hierarchy - *the Isogeometry level*



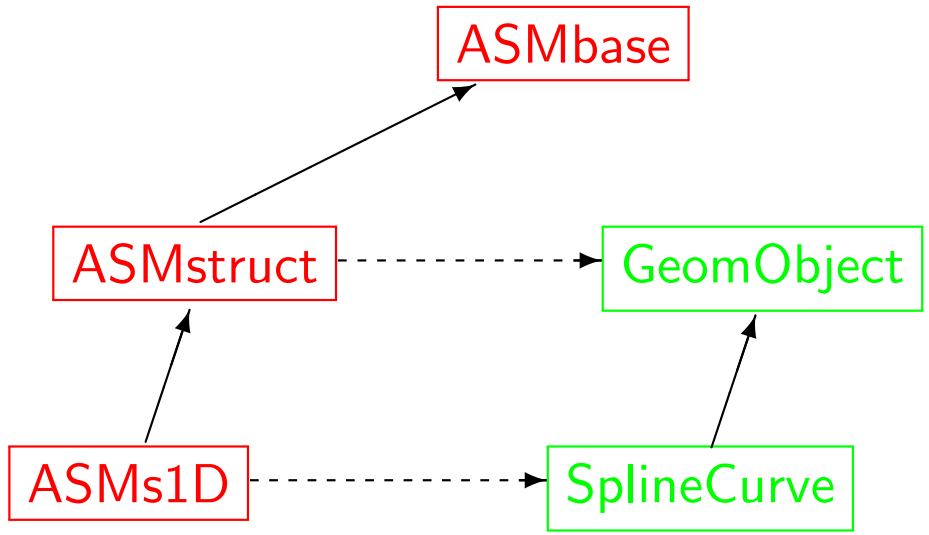
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



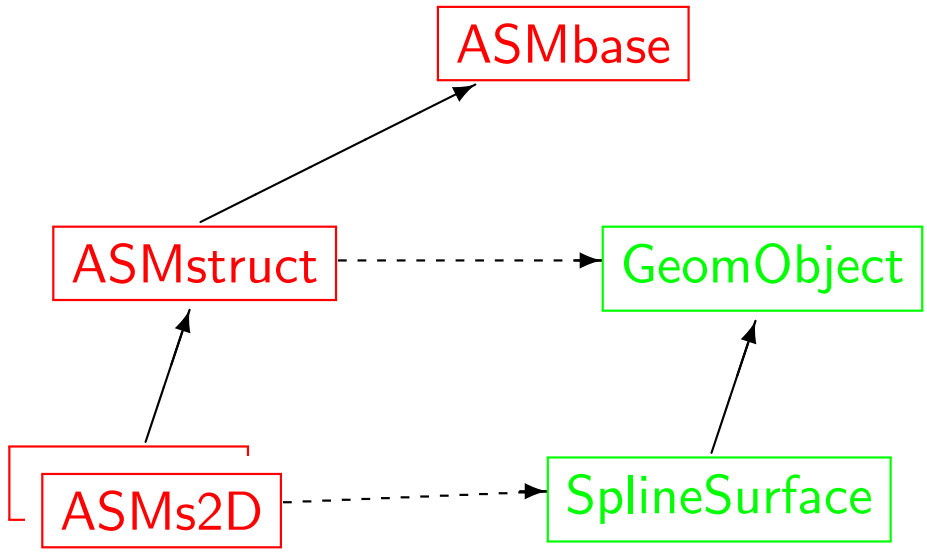
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



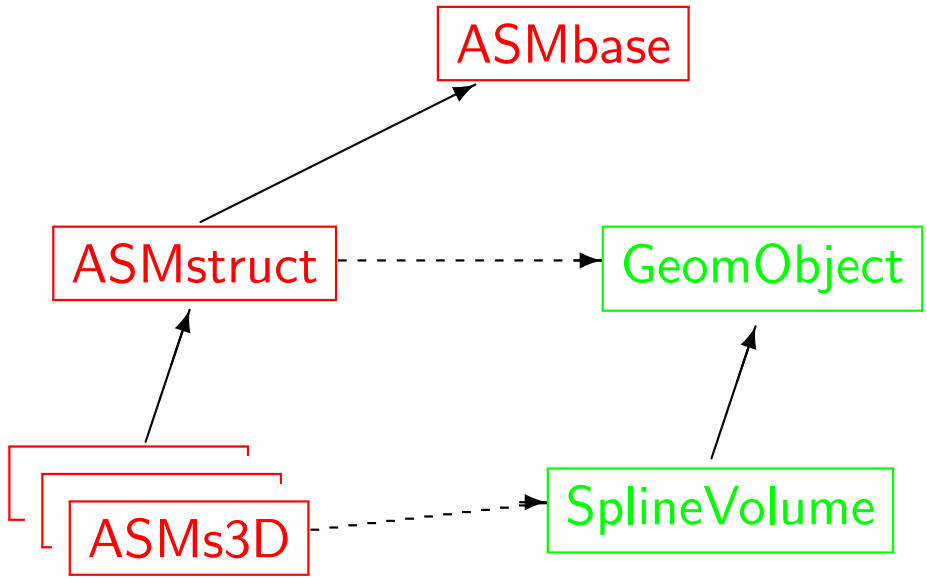
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



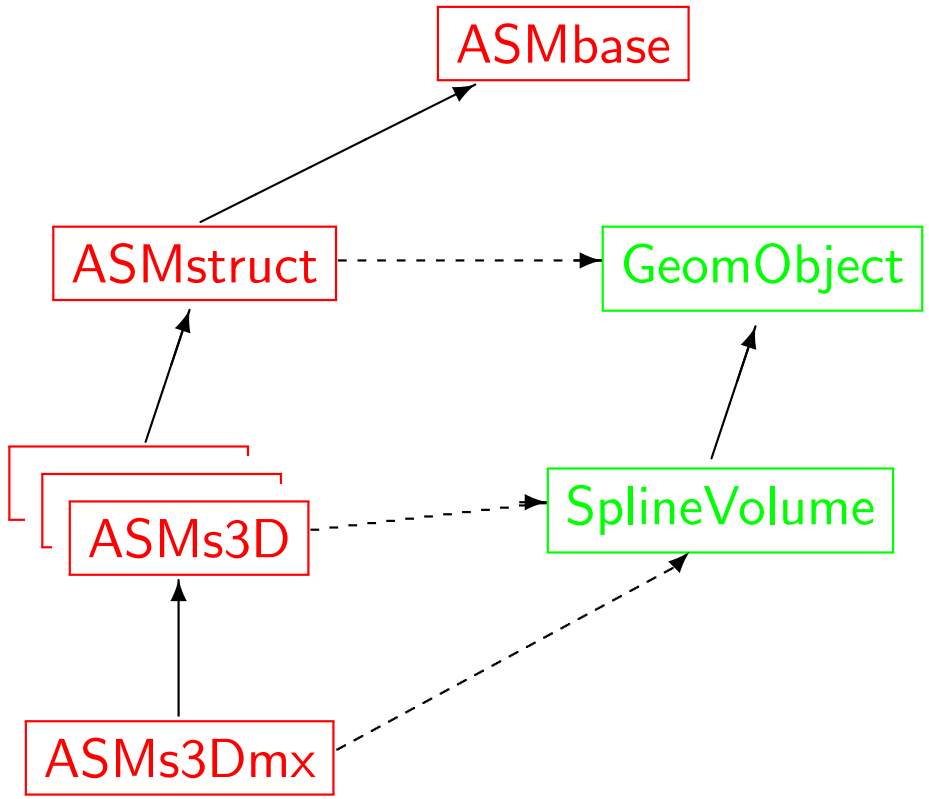
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



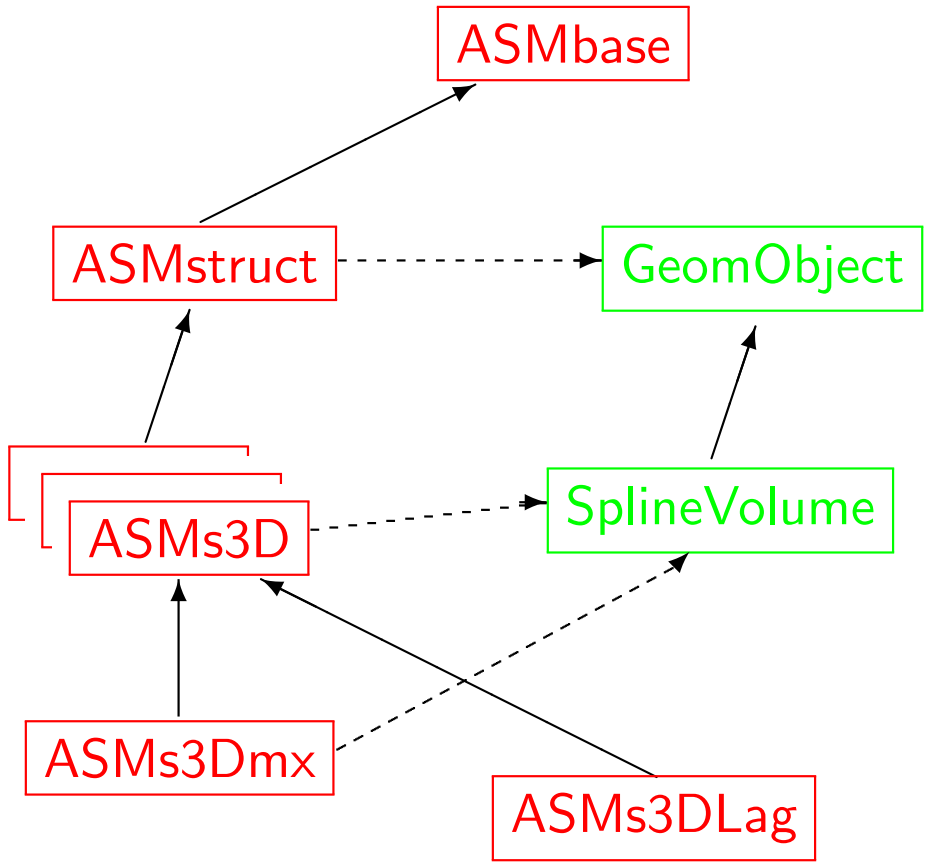
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



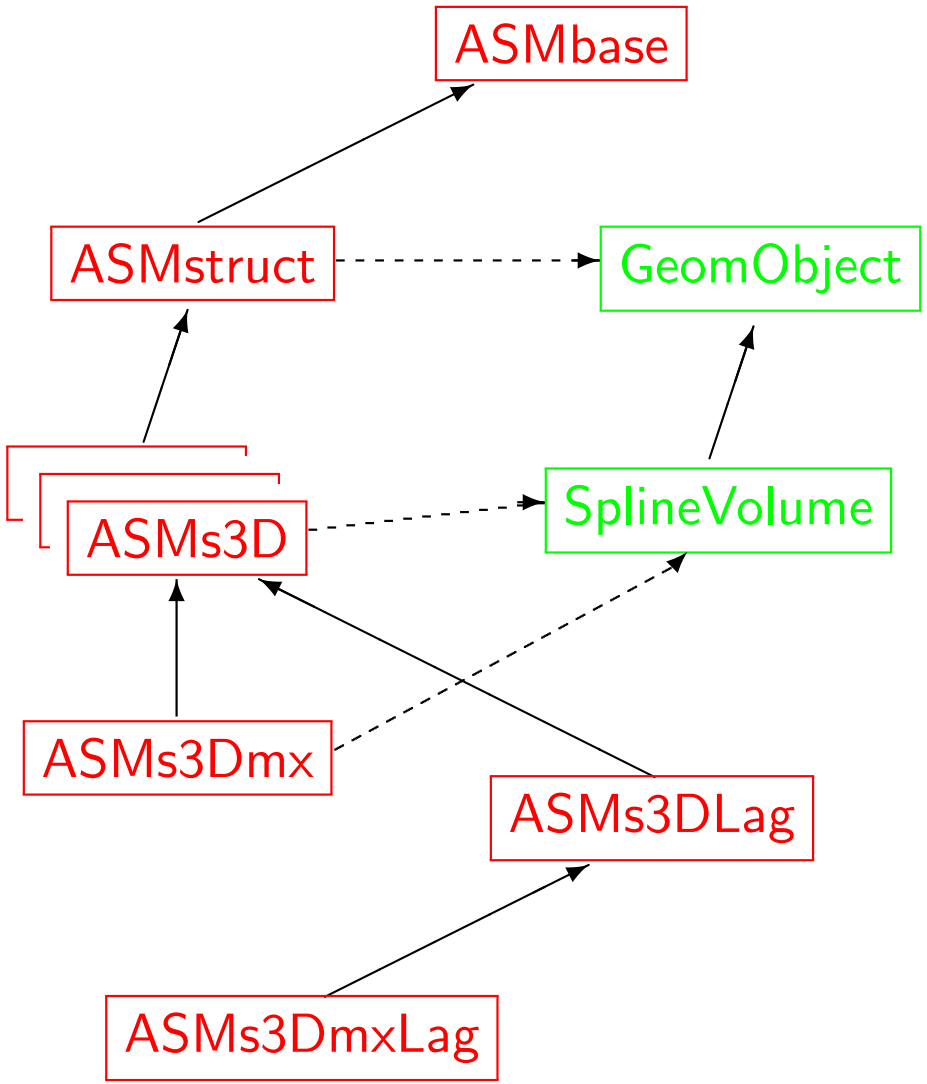
→ 'is-a' relationship

- - → 'has-a' relationship

IFEM classes

GoTools classes

ASM class hierarchy - *the Isogeometry level*



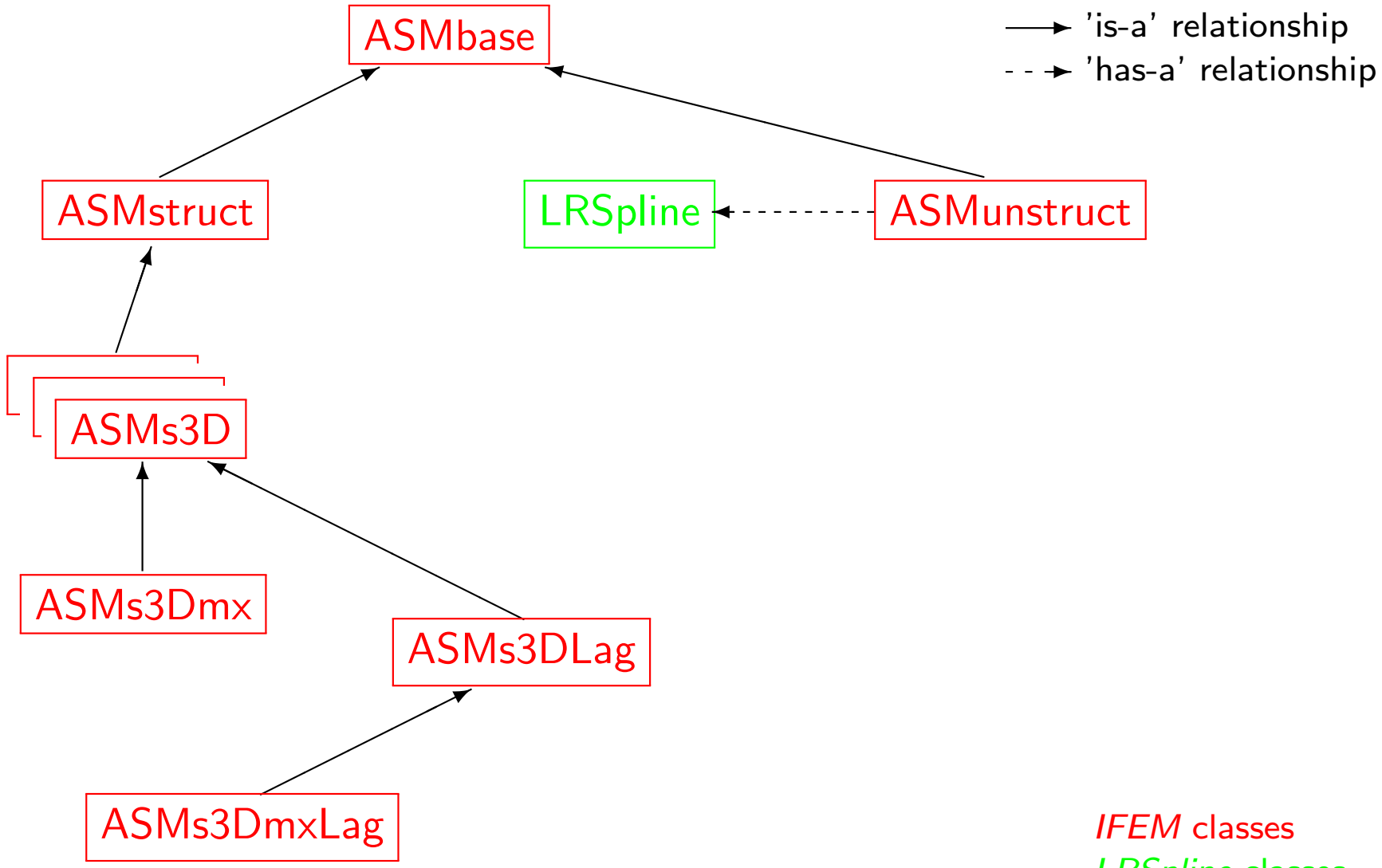
→ 'is-a' relationship

- - → 'has-a' relationship

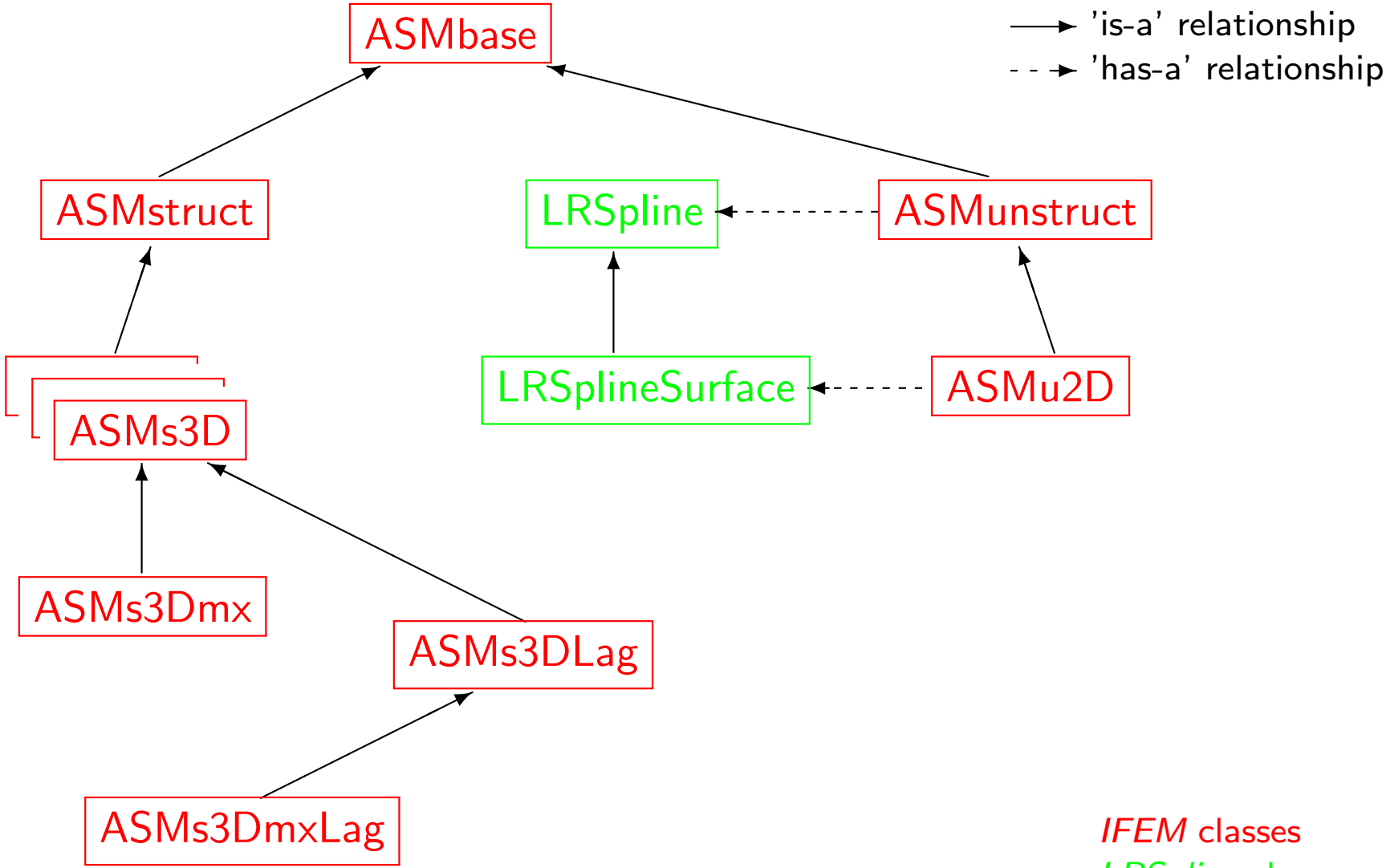
IFEM classes

GoTools classes

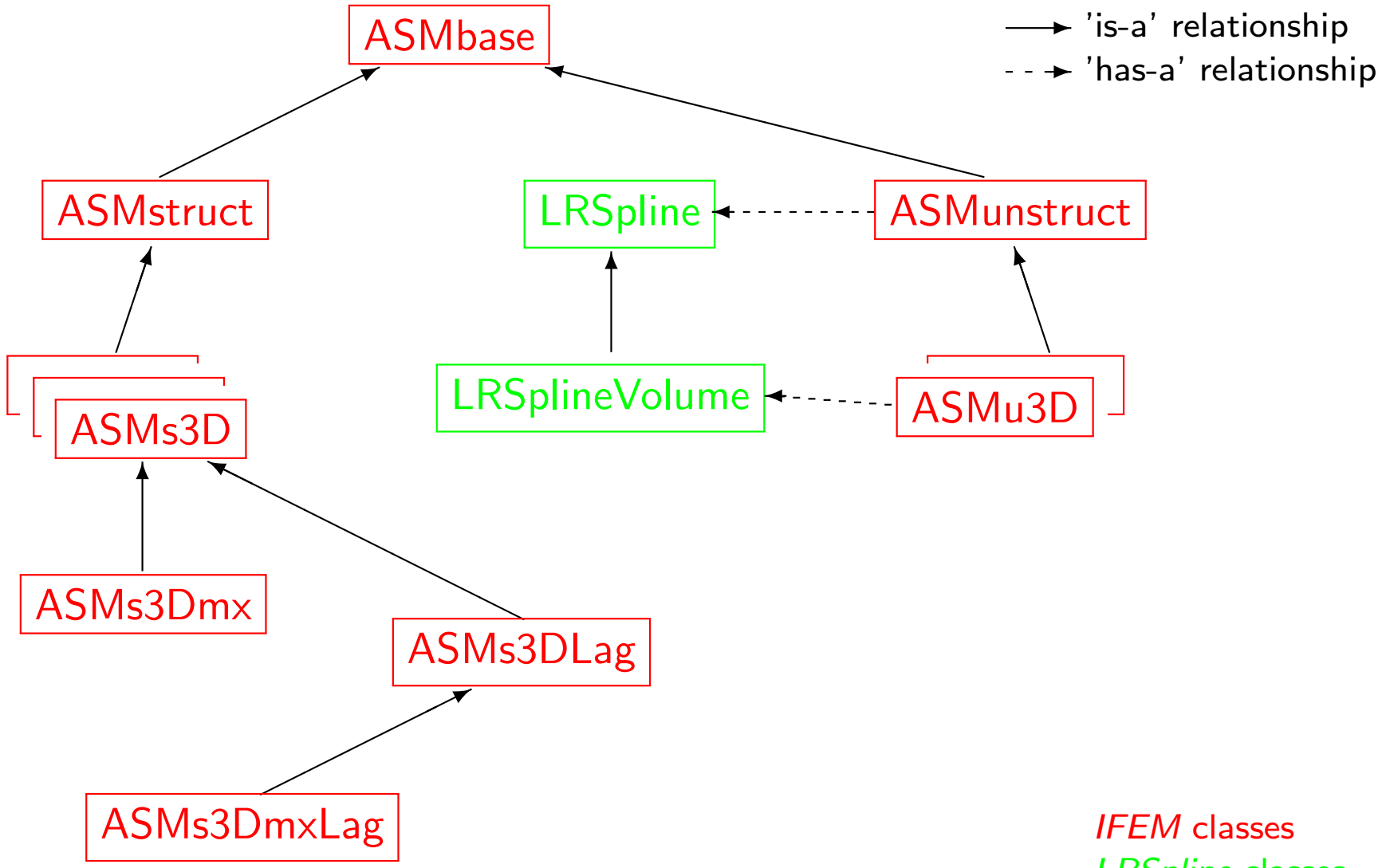
ASM class hierarchy - *the Isogeometry level*



ASM class hierarchy - *the Isogeometry level*

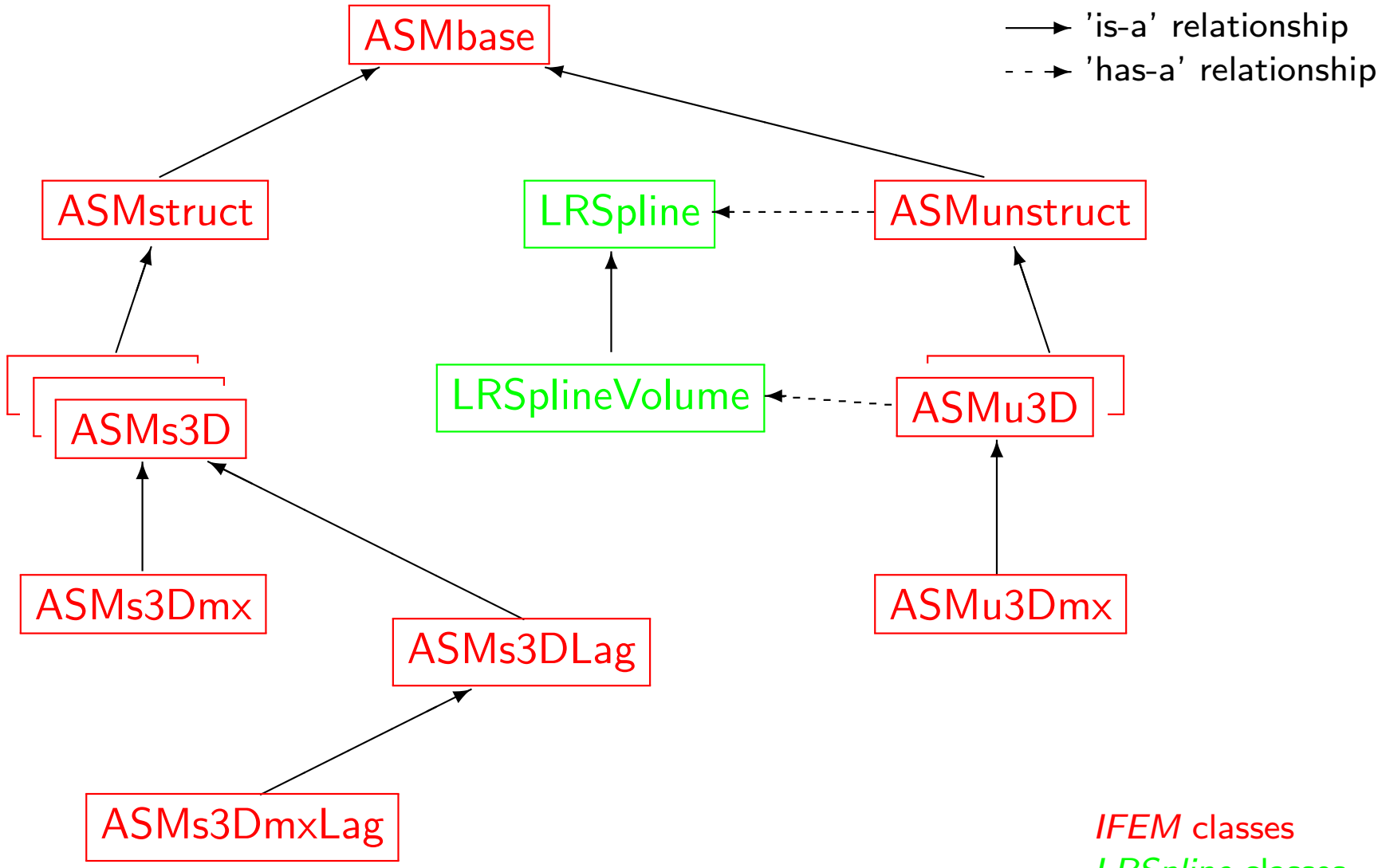


ASM class hierarchy - *the Isogeometry level*

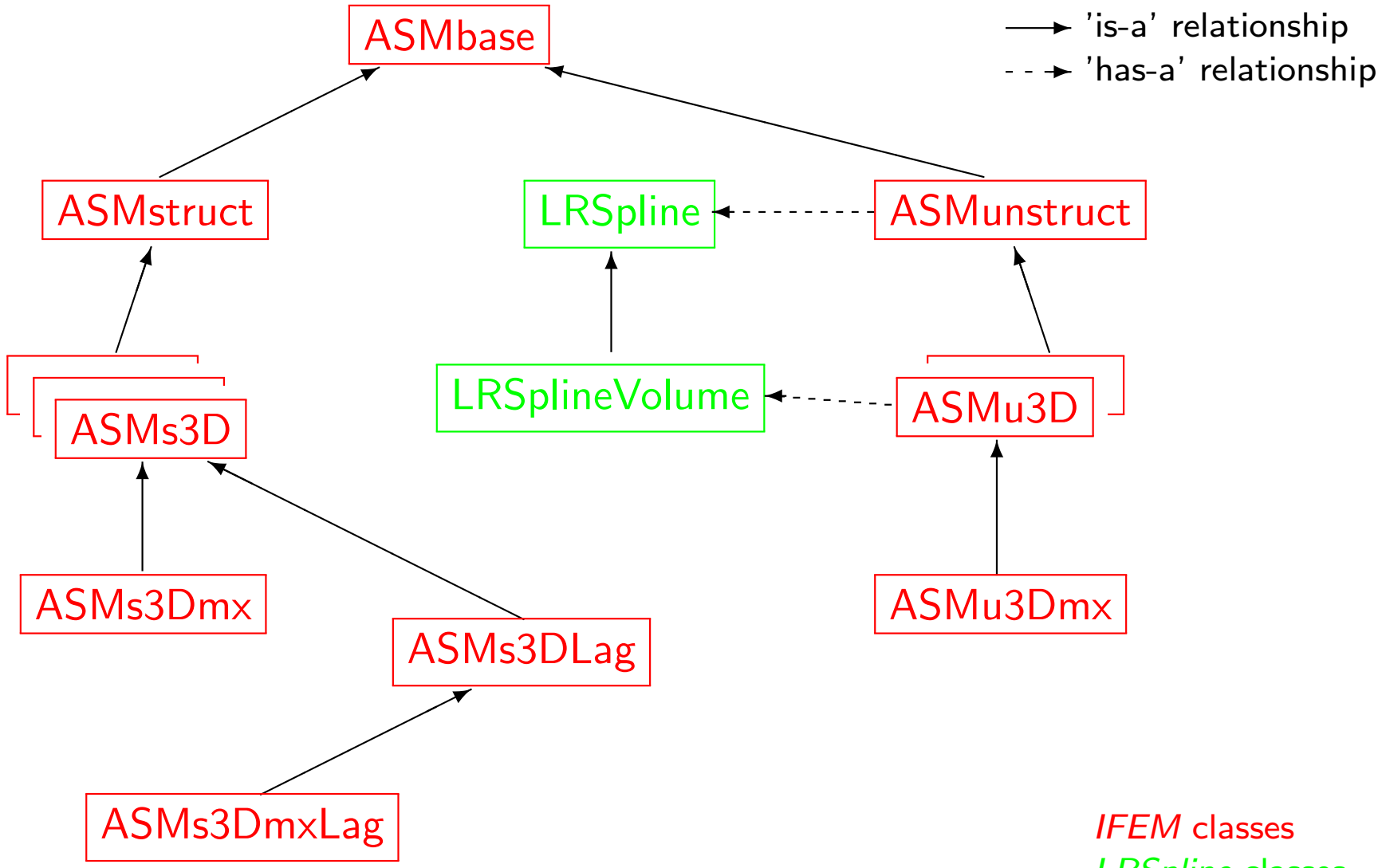


IFEM classes
LRSpline classes

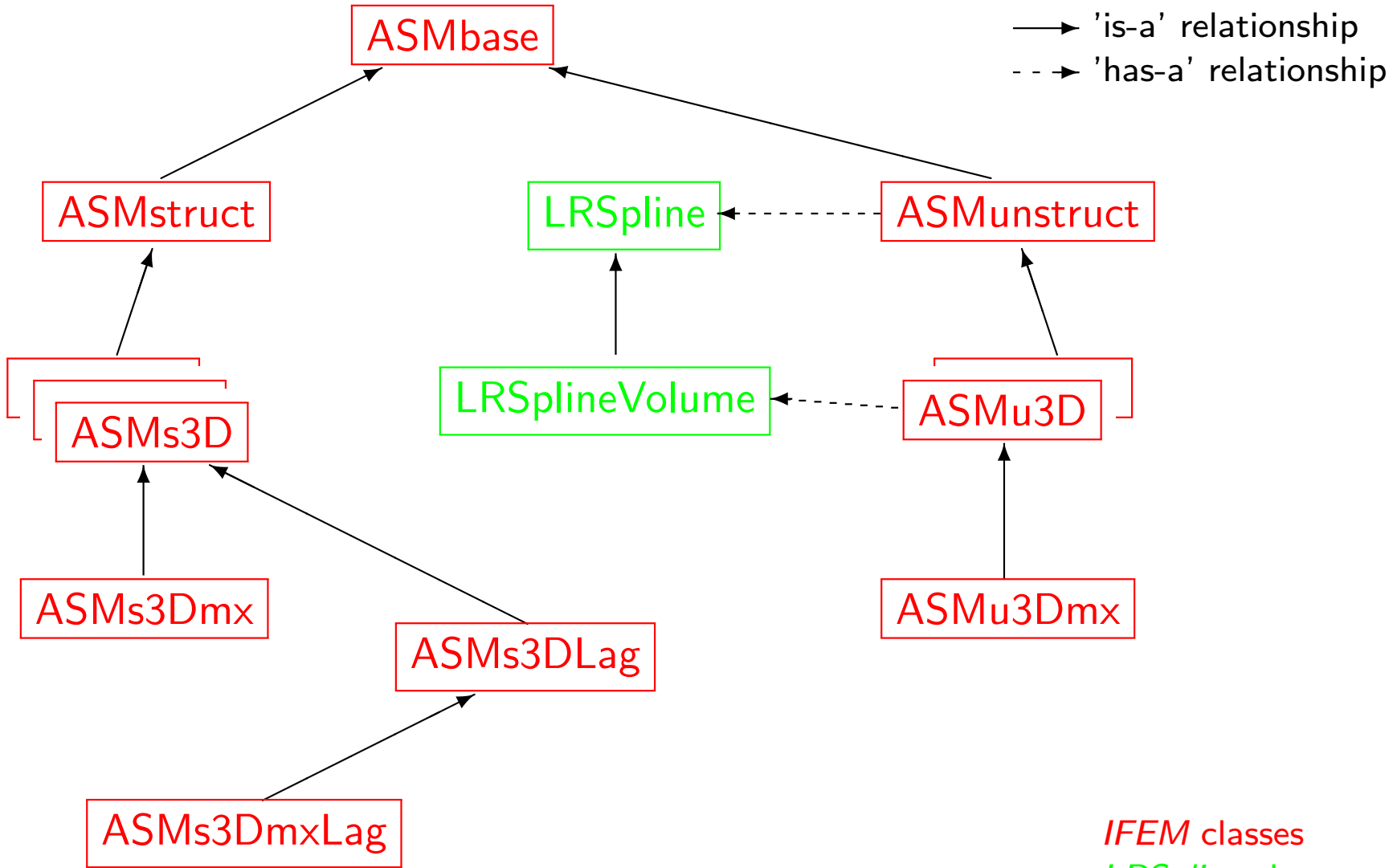
ASM class hierarchy - *the Isogeometry level*



ASM class hierarchy - *the Isogeometry level*



ASM class hierarchy - *the Isogeometry level*



ASMbase is the interface between the isogeometric FE procedures, and the solution algorithms (from above) and the physical problem to be solved (from below).

The main ASM methods

```
class ASMbase
{
public:
  //! \brief Evaluates an integral over the interior patch domain.
  //! \param integrand Object with problem-specific data and methods
  //! \param glbInt The integrated quantity
  //! \param[in] time Parameters for nonlinear/time-dependent simulations
  virtual bool integrate(Integrand& integrand,
                         GlobalIntegral& glbInt, const TimeDomain& time) = 0;

  //! \brief Evaluates a boundary integral over a patch face/edge.
  //! \param integrand Object with problem-specific data and methods
  //! \param[in] lIndex Local index of the boundary face/edge
  //! \param glbInt The integrated quantity
  //! \param[in] time Parameters for nonlinear/time-dependent simulations
  virtual bool integrate(Integrand& integrand, int lIndex,
                         GlobalIntegral& glbInt, const TimeDomain& time) = 0;
};
```

LocalIntegral and GlobalIntegral are interfaces to the element-level and system-level matrices of the FE problem. TimeDomain contains the integration parameters needed for nonlinear and/or time-dependent simulations.

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{




 }
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

```
  basis->SplineVolume::computeBasisGrid(u,v,w,splineData);
```

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

```
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

```
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

```
  basis->SplineVolume::computeBasisGrid(u,v,w,splineData);
```

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
    LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

```
    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

Loop over integration points; `do i=1,nGauss, j=1,nGauss, k=1,nGauss`

`end do i, j, k`

```
  end if
 end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

Loop over integration points; `do i=1,nGauss, j=1,nGauss, k=1,nGauss`

Fetch data from `splineData` belonging to current integration point; `N, dN/du`

```
    end do i, j, k

  end if
 end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

Loop over integration points; `do i=1,nGauss, j=1,nGauss, k=1,nGauss`

Fetch data from `splineData` belonging to current integration point; `N, dN/du`

Compute Cartesian coordinates and Jacobian; `X = N*Xnod, J = dN/du*Xnod`

and the gradient; `dN/dX = dN/du * J`$^{-1}$

`end do i, j, k`

`end if`

`end do iel`

```
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

`basis->SplineVolume::computeBasisGrid(u,v,w,splineData);`

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

Loop over integration points; `do i=1,nGauss, j=1,nGauss, k=1,nGauss`

Fetch data from `splineData` belonging to current integration point; `N, dN/du`

Compute Cartesian coordinates and Jacobian; `X = N*Xnod, J = dN/du*Xnod`

and the gradient; `dN/dX = dN/du * J`$^{-1}$

```
integrand.evalInt(*A, time, detJ*weight, N, dN/dX, X);
```

`end do i, j, k`

`end if`

`end do iel`

```
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis->SplineVolume::computeBasisGrid(u,v,w,splineData);
    splineData contains derivatives w.r.t. u,v,w of all basis functions
    at all integration points and the function values themselves
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates (control points) for current element, Xnod
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData belonging to current integration point; N, dN/du
        Compute Cartesian coordinates and Jacobian; X = N*Xnod, J = dN/du*Xnod
        and the gradient; dN/dX = dN/du * J^-1
        integrand.evalInt(*A, time, detJ*weight, N, dN/dX, X);
      end do i, j, k
      integrand.finalizeElement(*A);


    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis->SplineVolume::computeBasisGrid(u,v,w,splineData);
    splineData contains derivatives w.r.t. u,v,w of all basis functions
    at all integration points and the function values themselves
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates (control points) for current element, Xnod
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData belonging to current integration point; N, dN/du
        Compute Cartesian coordinates and Jacobian; X = N*Xnod, J = dN/du*Xnod
        and the gradient; dN/dX = dN/du * J^-1
        integrand.evalInt(*A, time, detJ*weight, N, dN/dX, X);
      end do i, j, k
      integrand.finalizeElement(*A);
      glInt.assemble(A->ref(), MGEL[iel]);
      A->destruct();
    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

```
basis->SplineVolume::computeBasisGrid(u,v,w,splineData);
```

`splineData` contains derivatives w.r.t. u,v,w of all basis functions at all integration points and the function values themselves

Loop over elements (knot-spans); `do iel=0,nel-1`

`If` current knot span is non-zero in all three directions `then`

```
LocalIntegral* A = integrand.getLocalIntegral(iel);
```

Initialize for numerical integration over the element

Fetch nodal coordinates (control points) for current element, `Xnod`

Loop over integration points; `do i=1,nGauss, j=1,nGauss, k=1,nGauss`

Fetch data from `splineData` belonging to current integration point; `N, dN/du`

Compute Cartesian coordinates and Jacobian; `X = N*Xnod, J = dN/du*Xnod`

and the gradient; `dN/dX = dN/du * J`$^{-1}$

```
integrand.evalInt(*A, time, detJ*weight, N, dN/dX, X);
```

```
end do i, j, k
integrand.finalizeElement(*A);
glInt.assemble(A->ref(), MGEL[iel]);
A->destruct();
```

```
end if
end do iel
}
```

The “user” interface ...

```
class Integrand
{
  //! \brief Evaluates the integrand at an interior point.
  //! \param elmInt The local integral object to receive the contributions
  //! \param[in] fe Finite element data of current integration point
  //! \param[in] time Parameters for nonlinear and time-dependent simulations
  //! \param[in] X Cartesian coordinates of current integration point
  //!
  //! \details The default implementation forwards to the stationary version.
  //! Reimplement this method for time-dependent or non-linear problems.
  virtual bool evalInt(LocalIntegral& elmInt, const FiniteElement& fe,
                       const TimeDomain& time, const Vec3& X) const;


  //! \brief Evaluates the integrand at a boundary point.
  //! \param elmInt The local integral object to receive the contributions
  //! \param[in] fe Finite element data of current integration point
  //! \param[in] time Parameters for nonlinear and time-dependent simulations
  //! \param[in] X Cartesian coordinates of current integration point
  //! \param[in] normal Boundary normal vector at current integration point
  //!
  //! \details The default implementation forwards to the stationary version.
  //! Reimplement this method for time-dependent or non-linear problems.
  virtual bool evalBou(LocalIntegral& elmInt, const FiniteElement& fe,
                       const TimeDomain& time,
                       const Vec3& X, const Vec3& normal) const;

  . . .
};
```

Overloaded versions of these method interfaces exist without the `TimeDomain` argument, for stationary/linear problems.

Finite element data at integration point level

```
class FiniteElement
{
public:
  int      iel;    //!< Element identifier
  size_t   iGP;    //!< Global integration point counter
  double   u;      //!< First parameter of current point
  double   v;      //!< Second parameter of current point
  double   w;      //!< Third parameter of current point
  double   xi;     //!< First local coordinate of current integration point
  double   eta;    //!< Second local coordinate of current integration point
  double   zeta;   //!< Third local coordinate of current integration point
  double   h;      //!< Characteristic element size
  Vector   N;      //!< Basis function values
  Vector   Navg;   //!< Volume-averaged basis function values
  Matrix   dNdX;   //!< First derivatives (gradient) of the basis functions
  Matrix3D d2NdX2; //!< Second derivatives of the basis functions
  Matrix   G;      //!< Matrix used for stabilized methods
  double   detJxW; //!< Weighted determinant of the coordinate mapping
};
```

An object of this class is used to transport all integration point quantities to the application-dependent integrands.

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
  - First approach: Establish the first basis by order-elevating the second basis (yields only $\mathcal{C}^{p-2}$ continuity for the first basis).
  - Second approach: Add one extra control point for the first basis in each parameter direction, and then reparameterize (both bases will posess $\mathcal{C}^{p-1}$ continuity but will have separate control point locations).

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
  - First approach: Establish the first basis by order-elevating the second basis (yields only $\mathcal{C}^{p-2}$ continuity for the first basis).
  - Second approach: Add one extra control point for the first basis in each parameter direction, and then reparameterize (both bases will posess $\mathcal{C}^{p-1}$ continuity but will have separate control point locations).
- The knot-span elements become the same for the two bases, $\Rightarrow$ simplifies the finite element topology management.

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
  - First approach: Establish the first basis by order-elevating the second basis (yields only $\mathcal{C}^{p-2}$ continuity for the first basis).
  - Second approach: Add one extra control point for the first basis in each parameter direction, and then reparameterize (both bases will posess $\mathcal{C}^{p-1}$ continuity but will have separate control point locations).
- The knot-span elements become the same for the two bases, $\Rightarrow$ simplifies the finite element topology management.
- Since the geometry represented by the two bases will be identical, it suffice to use the second (lowest order) basis only, when evaluating the Jacobian of the geometry mapping and the basis function gradients w.r.t. Cartesian coordinates.

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
  - First approach: Establish the first basis by order-elevating the second basis (yields only $\mathcal{C}^{p-2}$ continuity for the first basis).
  - Second approach: Add one extra control point for the first basis in each parameter direction, and then reparameterize (both bases will posess $\mathcal{C}^{p-1}$ continuity but will have separate control point locations).
- The knot-span elements become the same for the two bases, $\Rightarrow$ simplifies the finite element topology management.
- Since the geometry represented by the two bases will be identical, it suffice to use the second (lowest order) basis only, when evaluating the Jacobian of the geometry mapping and the basis function gradients w.r.t. Cartesian coordinates.
- The user only needs to relate to the lowest-order grid/basis, the higher order basis is established internally automatically.

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{

}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
```

Compute parameter values (u,v,w) of all integration points within the patch

```
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
```

```
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then

    end if
  end do iel
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
    end if
  end do iel
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss




      end do i, j, k



    end if
  end do iel
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData[12]; N1, dN1/du, N2, dN2/du

      end do i, j, k

    end if
  end do iel
 }
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData[12]; N1, dN1/du, N2, dN2/du
        Compute Cartesian coordinates and Jacobian; X = N2*Xnod, J = dN2/du*Xnod
        and the gradients; dN1/dX = dN1/du * J^-1, dN2/dX = dN2/du * J^-1,


      end do i, j, k



    end if
  end do iel
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData[12]; N1, dN1/du, N2, dN2/du
        Compute Cartesian coordinates and Jacobian; X = N2*Xnod, J = dN2/du*Xnod
        and the gradients; dN1/dX = dN1/du * J^-1, dN2/dX = dN2/du * J^-1,
        integrand.evalInt(*A, time, detJ*weight,
                          N1, dN1/dX, N2, dN2/dX, X);
      end do i, j, k


    end if
  end do iel
}
```

Numerical integration for the two-field mixed method

```
bool ASMs3Dmx::integrate (Integrand& integrand, GlobalIntegral& glInt,
                          const TimeDomain& time)
{
  Compute parameter values (u,v,w) of all integration points within the patch
  basis1->SplineVolume::computeBasisGrid(u,v,w,splineData1);
  basis2->SplineVolume::computeBasisGrid(u,v,w,splineData2);
  Loop over elements (knot-spans); do iel=0,nel-1
    If current knot span is non-zero in all three directions then
      LocalIntegral* A = integrand.getLocalIntegral(iel);
      Initialize for numerical integration over the element
      Fetch nodal coordinates for current element, Xnod (Note: for basis2 only)
      Loop over integration points; do i=1,nGauss, j=1,nGauss, k=1,nGauss
        Fetch data from splineData[12]; N1, dN1/du, N2, dN2/du
        Compute Cartesian coordinates and Jacobian; X = N2*Xnod, J = dN2/du*Xnod
        and the gradients; dN1/dX = dN1/du * J^-1, dN2/dX = dN2/du * J^-1,
        integrand.evalInt(*A, time, detJ*weight,
                          N1, dN1/dX, N2, dN2/dX, X);
      end do i, j, k
      integrand.finalizeElement(*A);
      glInt.assemble(A->ref(), MGEL[iel]);
      A->destruct();
    end if
  end do iel
}
```

Current Applications - `Integrand` sub-classes

- `Poisson` - Simple scalar equation

Current Applications - `Integrand` sub-classes

- `Poisson` - Simple scalar equation
- `Elasticity` - Solid mechanics problems
  - `LinearElasticity` - Linear elasticity, isotropic material
  - `NonlinearElasticityTL` - Finite deformation elasticity, Total Lagrangian formulation, linear elastic material
  - `NonlinearElasticityUL` - Finite deformation elasticity, Updated Lagrangian formulation, linear elastic, Neo-Hookean and plastic materials (linear elastic only in public version)

Current Applications - `Integrand` sub-classes

- `Poisson` - Simple scalar equation
- `Elasticity` - Solid mechanics problems
  - `LinearElasticity` - Linear elasticity, isotropic material
  - `NonlinearElasticityTL` - Finite deformation elasticity, Total Lagrangian formulation, linear elastic material
  - `NonlinearElasticityUL` - Finite deformation elasticity, Updated Lagrangian formulation, linear elastic, Neo-Hookean and plastic materials (linear elastic only in public version)
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes (not in public version)
    - `NonlinearElasticityULMixed` - ..., mixed formulation with continuous pressure and volumetric change fields (not public)
    - `NonlinearElasticityFbar` - ..., $\bar{F}$-formulation for the handling of nearly-incompressible materials (not public)

Current Applications - `Integrand` sub-classes

- `Poisson` - Simple scalar equation
- `Elasticity` - Solid mechanics problems
  - `LinearElasticity` - Linear elasticity, isotropic material
  - `NonlinearElasticityTL` - Finite deformation elasticity, Total Lagrangian formulation, linear elastic material
  - `NonlinearElasticityUL` - Finite deformation elasticity, Updated Lagrangian formulation, linear elastic, Neo-Hookean and plastic materials (linear elastic only in public version)
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes (not in public version)
    - `NonlinearElasticityULMixed` - ..., mixed formulation with continuous pressure and volumetric change fields (not public)
    - `NonlinearElasticityFbar` - ..., $\bar{F}$-formulation for the handling of nearly-incompressible materials (not public)
- Navier–Stokes CFD solvers (not part of the *ICADA* project)

Current Applications - `Integrand` sub-classes

- `Poisson` - Simple scalar equation
- `Elasticity` - Solid mechanics problems
  - `LinearElasticity` - Linear elasticity, isotropic material
  - `NonlinearElasticityTL` - Finite deformation elasticity, Total Lagrangian formulation, linear elastic material
  - `NonlinearElasticityUL` - Finite deformation elasticity, Updated Lagrangian formulation, linear elastic, Neo-Hookean and plastic materials (linear elastic only in public version)
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes (not in public version)
    - `NonlinearElasticityULMixed` - ..., mixed formulation with continuous pressure and volumetric change fields (not public)
    - `NonlinearElasticityFbar` - ..., $\bar{F}$-formulation for the handling of nearly-incompressible materials (not public)
- Navier–Stokes CFD solvers (not part of the *ICADA* project)
- Many others (coupled simulators, etc.)

Implementational issues (integration point level)

- Using splines as basis function, especially the higher-order ones, the "elements" become large (in terms of nodal connectivities) $\Rightarrow$ large, dense element matrices

Implementational issues (integration point level)

- Using splines as basis function, especially the higher-order ones, the "elements" become large (in terms of nodal connectivities) $\Rightarrow$ large, dense element matrices
- Element-level linear algebra: Use machine-optimized **BLAS** rather than inline C++ code
- Important to express the nonlinear FE formulation on *matrix* form (Voigt notation) — not *tensor* form

Implementational issues (integration point level)

- Using splines as basis function, especially the higher-order ones, the "elements" become large (in terms of nodal connectivities) $\Rightarrow$ large, dense element matrices
- Element-level linear algebra: Use machine-optimized **BLAS** rather than inline C++ code
- Important to express the nonlinear FE formulation on *matrix* form (Voigt notation) — not *tensor* form
- In addition: Integration and assembly of element-level matrices is done in parallel using multi-threading (OpenMP)

System-level linear algebra – equation solving

- Interfaced through classes `SystemMatrix` and `SystemVector` with sub-classes for particular solvers.
- Current available linear equation solvers:
  - LAPACK `DGESV` (dense matrices, small problems only)
  - SuperLU (direct methods) `http://crd.lbl.gov/~xiaoye/SuperLU`
  - PETSc (iterative methods) `http://www.mcs.anl.gov/petsc`
  - Parallelization on distributed memory based on PETSc/MPI

Detailed source code documentation

See the doxygen-generated html-pages ../html/index.html

Tutorial: Poisson equation in R^2

Given a heat source function $f(x, y)$ defined over a domain $\Omega \in R^2$, a flux function $h(x, y)$ defined over the boundary $\partial\Omega_h$, and a function $g(x, y)$ defined over the boundary $\partial\Omega_g = \partial\Omega \setminus \partial\Omega_h$, find the scalar function $u(x, y)$ satisfying

$$\left.\begin{aligned} q_{i,i} &= f \\ q_i &= -\kappa_{ij} u_{,j} \end{aligned}\right\} \quad \forall \quad \{x, y\} \in \overline{\Omega} \tag{1}$$

$$q_i n_i = h \quad \forall \quad \{x, y\} \in \partial\Omega_h \tag{2}$$

$$u = g \quad \forall \quad \{x, y\} \in \partial\Omega_g \tag{3}$$

where κ_{ij} is the conductivity tensor and n_i defines the outward-directed unit normal vector on $\partial\Omega_h$.

Tutorial: Poisson equation

```
class Poisson : public IntegrandBase
{
protected:
  // Physical properties
  double    kappa;   //!< Conductivity (constant)
  RealFunc* fluxFld; //!< Boundary normal flux field
  RealFunc* heatSrc; //!< Interior heat source field
```

Define the class `Poisson` as an `IntegrandBase` subclass (the class `IntegrandBase` inherits `Integrand`), containing data and methods specific to the 2D Poisson problem (assuming constant conductivity).

Tutorial: Poisson equation

```
class Poisson : public IntegrandBase
{
protected:
  // Physical properties
  double    kappa;   //!< Conductivity (constant)
  RealFunc* fluxFld; //!< Boundary normal flux field
  RealFunc* heatSrc; //!< Interior heat source field
```

```
public:
  Poisson() : kappa(1.0), fluxFld(0), heatSrc(0)
  {
    primsol.resize(1);
  }
  virtual ~Poisson() {}
```

Define the class `Poisson` as an `IntegrandBase` subclass (the class `IntegrandBase` inherits `Integrand`), containing data and methods specific to the 2D Poisson problem (assuming constant conductivity).

Class constructor and destructor. The constructor `Poisson()` initializes the data members.

Tutorial: Poisson equation

```
class Poisson : public IntegrandBase
{
protected:
  // Physical properties
  double    kappa;   //!< Conductivity (constant)
  RealFunc* fluxFld; //!< Boundary normal flux field
  RealFunc* heatSrc; //!< Interior heat source field
```

```
public:
  Poisson() : kappa(1.0), fluxFld(0), heatSrc(0)
  {
    primsol.resize(1);
  }
  virtual ~Poisson() {}
```

```
  void setMaterial(double K) { kappa = K; }
  void setFlux(RealFunc* ff) { fluxFld = ff; }
  void setSource(RealFunc* src) { heatSrc = src; }
```

Define the class `Poisson` as an `IntegrandBase` subclass (the class `IntegrandBase` inherits `Integrand`), containing data and methods specific to the 2D Poisson problem (assuming constant conductivity).

Class constructor and destructor. The constructor `Poisson()` initializes the data members.

Initialization of physical properties.

Tutorial: Poisson equation

```
class Poisson : public IntegrandBase
{
protected:
  // Physical properties
  double    kappa;   //!< Conductivity (constant)
  RealFunc* fluxFld; //!< Boundary normal flux field
  RealFunc* heatSrc; //!< Interior heat source field
```

Define the class `Poisson` as an `IntegrandBase` subclass (the class `IntegrandBase` inherits `Integrand`), containing data and methods specific to the 2D Poisson problem (assuming constant conductivity).

```
public:
  Poisson() : kappa(1.0), fluxFld(0), heatSrc(0)
  {
    primsol.resize(1);
  }
  virtual ~Poisson() {}
```

Class constructor and destructor. The constructor `Poisson()` initializes the data members.

```
  void setMaterial(double K) { kappa = K; }
  void setFlux(RealFunc* ff) { fluxFld = ff; }
  void setSource(RealFunc* src) { heatSrc = src; }
```

Initialization of physical properties.

```
  virtual LocalIntegral* getLocalIntegral(size_t nen, size_t,
                                          bool neumann) const;
```

Virtual method returning an element matrix object for the Poisson integrand.

Tutorial: Poisson equation

```
virtual bool evalInt(LocalIntegral& elmInt,
                     const FiniteElement& fe,
                     const Vec3& X) const;
virtual bool evalBou(LocalIntegral& elmInt,
                     const FiniteElement& fe,
                     const Vec3& X,
                     const Vec3& normal) const;
virtual bool evalSol(Vector& s,
                     const FiniteElement& fe,
                     const Vec3& X,
                     const std::vector<int>& MNPC) const;
virtual bool evalSol(Vector& s,
                     const VecFunc& asol,
                     const Vec3& X) const;
```

Virtual methods for integrand and solution field evaluation.

Tutorial: Poisson equation

```
  virtual bool evalInt(LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X) const;
  virtual bool evalBou(LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X,
                       const Vec3& normal) const;
  virtual bool evalSol(Vector& s,
                       const FiniteElement& fe,
                       const Vec3& X,
                       const std::vector<int>& MNPC) const;
  virtual bool evalSol(Vector& s,
                       const VecFunc& asol,
                       const Vec3& X) const;

  virtual NormBase* getNormIntegrand(AnaSol* asol = 0) const;

  bool evalSol(Vector& s,
               const Vector& eV,
               const Matrix& dNdX,
               const Vec3& X) const;
  bool formCmatrix(Matrix& C, const Vec3& X,
                   bool invers = false) const;
};
```

Virtual methods for integrand and solution field evaluation.

Methods for solution norm integration.

Tutorial: Poisson equation

```
LocalIntegral* Poisson::getLocalIntegral (size_t nen, size_t,
                                          bool neumann) const
{
  ElmMats* result = new ElmMats();
  result->rhsOnly = neumann;
  result->withLHS = !neumann;
  result->resize(neumann ? 0 : 1, 1);
  result->redim(nen);
  return result;
}
```

Element initialization:
Allocate an element matrix object and set the size of the matrices based on the number of element nodes.
Indicate whether the left-hand-side matrices are to be integrated or not.

Tutorial: Poisson equation

```
bool Poisson::evalInt (LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X) const
{
  ElmMats& elMat = static_cast<ElmMats&>(elmInt);

  if (!elMat.A.empty())
  {
    Matrix C; C.diag(kappa,2); // Diagonal constitutive matrix

    Matrix CB;
    CB.multiply(C,fe.dNdX,false,true).multiply(fe.detJxW);
    elMat.A.front().multiply(fe.dNdX,CB,false,false,true);
  }

  if (heatSrc && !elMat.b.empty())
    elMat.b.front().add(fe.N,(*heatSrc)(X)*fe.detJxW);

  return true;
}

bool Poisson::evalBou (LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X, const Vec3& normal) const
{
  ElmMats& elMat = static_cast<ElmMats&>(elmInt);
  if (!fluxFld || elMat.b.empty()) return false;

  double h = -(*fluxFld)(X); // normal flux at point X

  elMat.b.front().add(fe.N,h*fe.detJxW);

  return true;
}
```

Integrand evaluations:
Assuming here $\kappa_{ij} = \kappa\delta_{ij}$

$$[CB] = [C] \cdot [\partial N/\partial \mathbf{X}]^T |J| w$$

$$[eM] = \sum ([\partial N/\partial \mathbf{X}] \cdot [CB])$$

$$\{eS\} = \sum (f\{N\}|J|w)$$

$$\{eS\} = \sum (-h\{N\}|J|w)$$

Tutorial: Poisson equation

```
bool Poisson::evalSol (Vector& q,
                       const FiniteElement& fe, const Vec3& X,
                       const std::vector<int>& MNPC) const
{
  if (primsol.front().empty()) return false;

  Vector eV;
  int ierr = utl::gather(MNPC,1,primsol.front(),eV);
  if (ierr > 0) return false;

  Matrix C; C.diag(kappa,2); // Diagonal constitutive matrix

  // Evaluate the heat flux vector
  Matrix CB;
  CB.multiply(C,fe.dNdX,false,true).multiply(eV,q);
  q *= -1.0;

  return true;
}
```

Secondary solution evaluation:

$$\mathbf{q} = [C] \cdot [\partial N/\partial \mathbf{X}]^T \cdot \{eV\}$$

```
bool Poisson::evalSol (Vector& s, const VecFunc& asol,
                       const Vec3& X) const
{
  s = Vector(asol(X).ptr(),2);
  return true;
}
```

Analytical solution

Tutorial: Poisson equation

```
class PoissonNorm : public NormBase
{
  VecFunc* anasol;  //!< Analytical heat flux

public:
  PoissonNorm(Poisson& p, VecFunc* a = 0)
  : NormBase(p), anasol(a) {}
  virtual ~PoissonNorm() {}

  virtual bool hasBoundaryTerms() const { return true; }

  virtual bool evalInt(LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X) const;
  virtual bool evalBou(LocalIntegral& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X,
                       const Vec3& normal) const;
};
```

```
NormBase* Poisson::getNormIntegrand (AnaSol* asol) const
{
  if (asol)
    return new PoissonNorm(*const_cast<Poisson*>(this),
                           asol->getScalarSecSol());
  else
    return new PoissonNorm(*const_cast<Poisson*>(this));
}
```

Accompanying class for solution norm integration

`NormBase` is a sub-class of `Integrand` with a couple of added methods common to all norm classes.

Tutorial: Poisson equation

```
bool PoissonNorm::evalInt (LocalIntegral& elmInt,
                           const FiniteElement& fe,
                           const Vec3& X) const
{
  Poisson& problem = static_cast<Poisson&>(myProblem);
  ElmNorm& pnorm = static_cast<ElmNorm&>(elmInt);

  // Evaluate the inverse constitutive matrix at this point
  Matrix Cinv;
  problem.formCmatrix(Cinv,X,true);

  // Evaluate the finite element heat flux field
  Vector sigmah;
  problem.evalSol(sigmah,pnorm.vec.front(),fe.dNdX,X);

  // Integrate the energy norm a(u^h,u^h)
  pnorm[0] += sigmah.dot(Cinv*sigmah)*fe.detJxW;
  // Integrate the external energy (h,u^h)
  double u = pnorm.vec.front().dot(fe.N);
  pnorm[1] += problem.getHeat(X)*u*fe.detJxW;

  if (anasol) {
    // Evaluate the analytical heat flux
    Vector sigma((*anasol)(X).ptr(),sigmah.size());
    // Integrate the energy norm a(u,u)
    pnorm[2] += sigma.dot(Cinv*sigma)*fe.detJxW;
    // Integrate the error in energy norm a(u-u^h,u-u^h)
    sigma -= sigmah;
    pnorm[3] += sigma.dot(Cinv*sigma)*fe.detJxW;
  }

  return true;
}
```

Norm integrand evaluation

Tutorial: Poisson equation

```
bool PoissonNorm::evalBou (LocalIntegral& elmInt,
                           const FiniteElement& fe,
                           const Vec3& X,
                           const Vec3& normal) const
{
  Poisson& problem = static_cast<Poisson&>(myProblem);
  ElmNorm& pnorm = static_cast<ElmNorm&>(elmInt);

  // Evaluate the surface heat flux
  double t = problem.getTraction(X,normal);
  // Evaluate the temperature field
  double u = pnorm.vec.front().dot(fe.N);

  // Integrate the external energy (t,u^h)
  pnorm[1] += t*u*fe.detJxW;

  return true;
}
```

Integration of external energy due to boundary flux.

Tutorial: Poisson equation

```
class SIMPoisson2D : public SIM2D
{
  Poisson   prob; //!< Poisson data and methods
  RealArray mVec; //!< Material data

public:
  SIMPoisson2D() : SIM2D(1) { myProblem = &prob; }
  virtual ~SIMPoisson2D() { myProblem = 0; }

protected:
  virtual bool parse(const TiXmlElement* elem);
  virtual bool initMaterial(size_t propInd);
  virtual bool initNeumann(size_t propInd);
};


bool SIMPoisson2D::initMaterial (size_t propInd)
{
  if (propInd >= mVec.size()) return false;
  prob.setMaterial(mVec[propInd]);
  return true;
}

bool SIMPoisson2D::initNeumann (size_t propInd)
{
  SclFuncMap::const_iterator sit = myScalars.find(propInd);
  if (sit == myVectors.end()) return false;
  prob.setFlux(tit->second);
  return true;
}
```

Simulation driver for 2D problems.

Tutorial: Poisson equation

```
class SIMPoisson2D : public SIM2D
{
  Poisson   prob; //!< Poisson data and methods
  RealArray mVec; //!< Material data

public:
  SIMPoisson2D() : SIM2D(1) { myProblem = &prob; }
  virtual ~SIMPoisson2D() { myProblem = 0; }

protected:
  virtual bool parse(const TiXmlElement* elem);
  virtual bool initMaterial(size_t propInd);
  virtual bool initNeumann(size_t propInd);
};


bool SIMPoisson2D::initMaterial (size_t propInd)
{
  if (propInd >= mVec.size()) return false;
  prob.setMaterial(mVec[propInd]);
  return true;
}

bool SIMPoisson2D::initNeumann (size_t propInd)
{
  SclFuncMap::const_iterator sit = myScalars.find(propInd);
  if (sit == myVectors.end()) return false;
  prob.setFlux(tit->second);
  return true;
}
```

Simulation driver for 2D problems.

Alternative, use templates to support multiple dimensions:

```
template<class Dim>
class SIMPoisson : public Dim
{
 ...
};
```

where `Dim` can be either `SIM1D`, `SIM2D` or `SIM3D`.

Tutorial: Poisson equation

```
bool SIMPoisson2D::parse (const TiXmlElement* elem)
{
  if (strcasecmp(elem->Value(),"poisson"))
    return this->SIM2D::parse(elem);

  const TiXmlElement* child = elem->FirstChildElement();
  for (; child; child = child->NextSiblingElement())

    if (!strcasecmp(child->Value(),"isotropic")) {
      int code = this->parseMaterialSet(child,mVec.size());
      double kappa = 1000.0;
      utl::getAttribute(child,"kappa",kappa);
      if (code == 0)
        prob.setMaterial(kappa);
      mVec.push_back(kappa);
    }
```

Tutorial: Poisson equation

```
    else if (!strcasecmp(child->Value(),"source")) {
      int code = -1; // Reserve negative code(s) for the source term function
      while (myScalars.find(code) != myScalars.end()) --code;
      std::string type;
      utl::getAttribute(child,"type",type,true);
      if (type == "expression" && child->FirstChild()) {
        std::cout <<"\tHeat source function: "
                  << child->FirstChild()->Value() << std::endl;
        myScalars[code] = new EvalFunction(child->FirstChild()->Value());
      }
      else {
        std::cerr <<"  ** SIMPoisson2D::parse: Invalid source function "<< type << std::endl;
        continue;
      }
      prob.setSource(myScalars[code]);
    }

    else if (!strcasecmp(child->Value(),"anasol")) {
      std::string type;
      utl::getAttribute(child,"type",type,true);
      if (type == "expression") {
        std::cout <<"\tAnalytical solution: Expression"<< std::endl;
        if (!mySol) mySol = new AnaSol(child);
      }
      else
        std::cerr <<"  ** SIMPoisson2D::parse: Invalid analytical solution "<< type << std::endl;
  }

  return true;
}
```

Tutorial: Poisson equation

```
int main (int argc, char** argv)
{
  // (Lots of initialisations skipped here...)

  // Read in model definitions and establish the FE data structures
  SIMbase* model = new SIMPoisson2D(); // (or new SIMPoisson<SIM2D>;)
  if (!model->read(infile))
    return 1;
  if (!model->preprocess(ignoredPatches,fixDup))
    return 1;

  model->setQuadratureRule(nGauss);

  Matrix eNorm;
  Vector gNorm, sol;

  model->initSystem(solver,1,1);
  model->setAssociatedRHS(0,0);
  if (!model->assembleSystem())
    return 2;

  // Solve the linear system of equations
  if (!model->solveSystem(sol,1))
    return 3;

  // Evaluate solution norms
  if (!model->solutionNorms(Vectors(1,sol),eNorm,gNorm))
    return 4;

  // Print output to terminal and VTF, etc.
}
```

Core parts of the main program

Tutorial: Poisson equation, sample input files

```
<simulation>

  <!-- General - geometry definitions !-->
  <geometry>
    <patchfile>square2D.g2</patchfile>
    <raiseorder patch="1" u="2" v="2"/>
    <refine type="uniform" patch="1" u="7" v="7"/>
    <topologysets>
      <set name="Dirichlet" type="edge">
        <item patch="1">4</item>
      </set>
      <set name="Neumann" type="edge">
        <item patch="1">3</item>
      </set>
    </topologysets>
  </geometry>

  <!-- General - boundary conditions !-->
  <boundaryconditions>
    <dirichlet set="Dirichlet" comp="1"/>
    <neumann type="anasol" set="Neumann" comp="1"/>
  </boundaryconditions>

  <!-- Problem-specific block !-->
  <poisson>
    <source type="expression">PI*PI*cos(PI*x)*(2-y)</source>
    <anasol type="expression">
      <primary>cos(PI*x)*(2-y)</primary>
      <secondary>PI*sin(PI*x)*(2-y)|cos(PI*x)</secondary>
    </anasol>
  </poisson>

</simulation>
```

square2d.g2:

```
200 1 0 0
3 0
2 2
0 0 1 1
2 2
0 0 1 1
0.0 0.0 0.0
2.0 0.0 0.0
0.0 2.0 0.0
2.0 2.0 0.0
```

Tutorial: Poisson equation, sample input files

```
<simulation>

  <!-- General - geometry definitions !-->
  <geometry>
    <patchfile>square2D.g2</patchfile>
    <raiseorder patch="1" u="2" v="2"/>
    <refine type="uniform" patch="1" u="7" v="7"/>
    <topologysets>
      <set name="Dirichlet" type="edge">
        <item patch="1">4</item>
      </set>
      <set name="Neumann" type="edge">
        <item patch="1">3</item>
      </set>
    </topologysets>
  </geometry>

  <!-- General - boundary conditions !-->
  <boundaryconditions>
    <dirichlet set="Dirichlet" comp="1"/>
    <neumann type="anasol" set="Neumann" comp="1"/>
  </boundaryconditions>

  <!-- Problem-specific block !-->
  <poisson>
    <source type="expression">PI*PI*cos(PI*x)*(2-y)</source>
    <anasol type="expression">
      <primary>cos(PI*x)*(2-y)</primary>
      <secondary>PI*sin(PI*x)*(2-y)|cos(PI*x)</secondary>
    </anasol>
  </poisson>

</simulation>
```

square2d.g2:

```
200 1 0 0
3 0
2 2
0 0 1 1
2 2
0 0 1 1
0.0 0.0 0.0
2.0 0.0 0.0
0.0 2.0 0.0
2.0 2.0 0.0
```

This is equivalent to both of the following:

```
<geometry scale="2.0"/>
<geometry Lx="2.0" Ly="2.0"/>
```

and then `<patchfile>` is not needed.