IFEM - getting started

Knut Morten Okstad

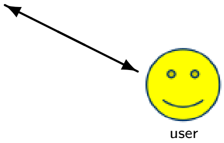
SINTEF ICT, Department of Applied Mathematics

March 1, 2012

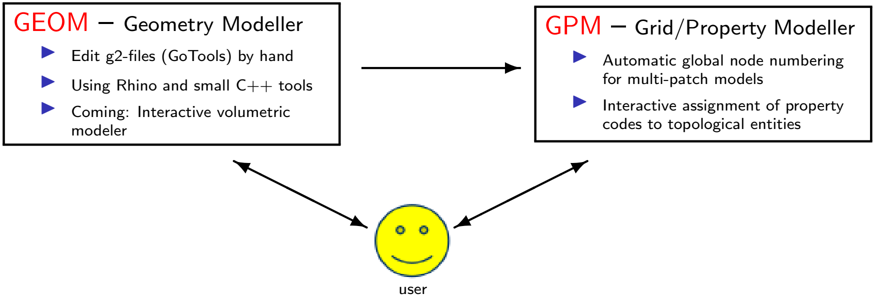
IFEM module overview

GEOM – Geometry Modeller

- Edit g2-files (GoTools) by hand
- Using Rhino and small C++ tools
- Coming: Interactive volumetric modeler



IFEM module overview



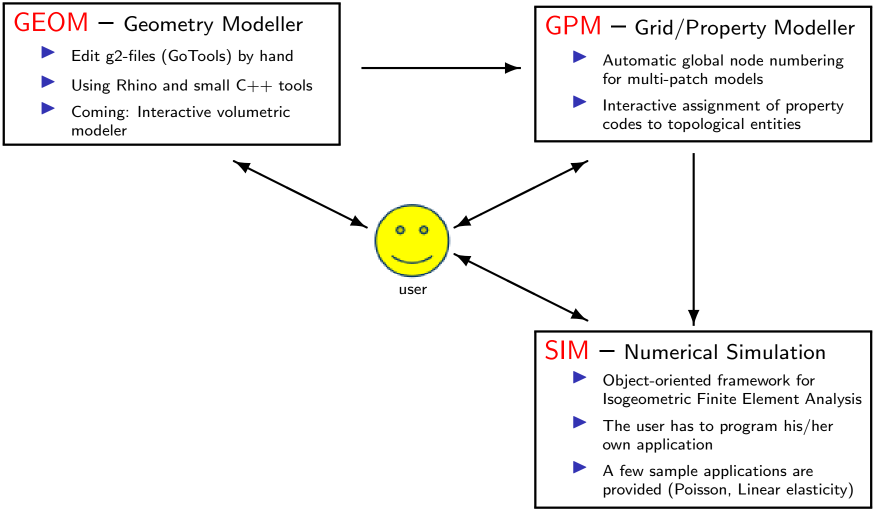
GEOM – Geometry Modeller

- Edit g2-files (GoTools) by hand
- Using Rhino and small C++ tools
- Coming: Interactive volumetric modeler

GPM – Grid/Property Modeller

- Automatic global node numbering for multi-patch models
- Interactive assignment of property codes to topological entities

IFEM module overview



GEOM – Geometry Modeller

- Edit g2-files (GoTools) by hand
- Using Rhino and small C++ tools
- Coming: Interactive volumetric modeler

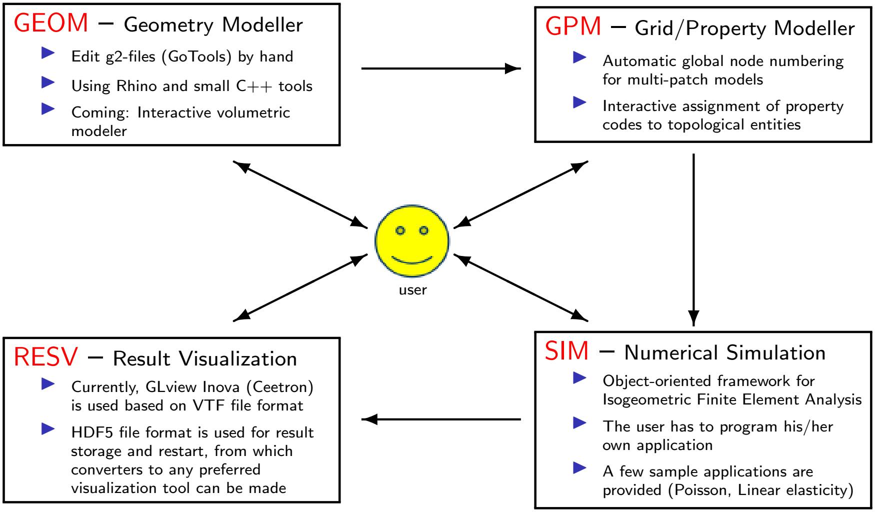
GPM – Grid/Property Modeller

- Automatic global node numbering for multi-patch models
- Interactive assignment of property codes to topological entities

SIM – Numerical Simulation

- Object-oriented framework for Isogeometric Finite Element Analysis
- The user has to program his/her own application
- A few sample applications are provided (Poisson, Linear elasticity)

IFEM module overview



GEOM – Geometry Modeller

- Edit g2-files (GoTools) by hand
- Using Rhino and small C++ tools
- Coming: Interactive volumetric modeler

GPM – Grid/Property Modeller

- Automatic global node numbering for multi-patch models
- Interactive assignment of property codes to topological entities

RESV – Result Visualization

- Currently, GLview Inova (Ceetron) is used based on VTF file format
- HDF5 file format is used for result storage and restart, from which converters to any preferred visualization tool can be made

SIM – Numerical Simulation

- Object-oriented framework for Isogeometric Finite Element Analysis
- The user has to program his/her own application
- A few sample applications are provided (Poisson, Linear elasticity)

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

     Isogeometric level
   - 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 **Isogeometric level** 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 **User/Application level** n int
   - Pr

ASM class hierarchy - *the Isogeometry level*

ASMbase

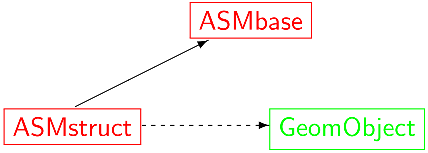
$\longrightarrow$ 'is-a' relationship

$\dashrightarrow$ 'has-a' relationship

IFEM classes

GoTools classes

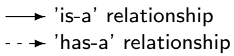
ASM class hierarchy - *the Isogeometry level*

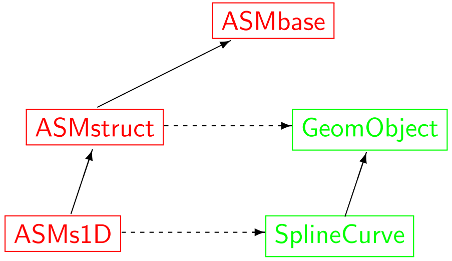


$\longrightarrow$ 'is-a' relationship
$\dashrightarrow$ 'has-a' relationship

IFEM classes
GoTools classes

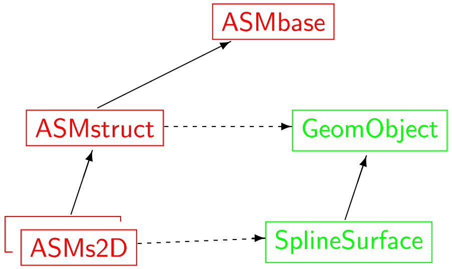
ASM class hierarchy - *the Isogeometry level*




$$\begin{tikzcd}
 & \boxed{\text{ASMbase}} & \\
\boxed{\text{ASMstruct}} \arrow[ur] \arrow[rr, dashed] & & \boxed{\text{GeomObject}} \\
\boxed{\text{ASMs1D}} \arrow[u] \arrow[rr, dashed] & & \boxed{\text{SplineCurve}} \arrow[u]
\end{tikzcd}$$

IFEM classes
GoTools classes

ASM class hierarchy - *the Isogeometry level*



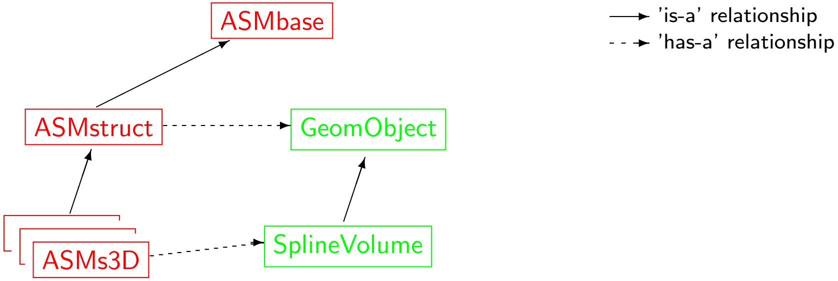
→ 'is-a' relationship

⇢ 'has-a' relationship

IFEM classes

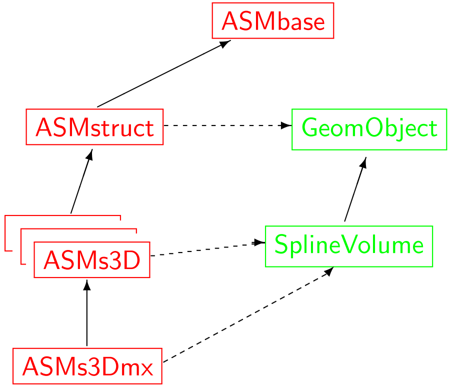
GoTools classes

ASM class hierarchy - *the Isogeometry level*



IFEM classes
GoTools classes

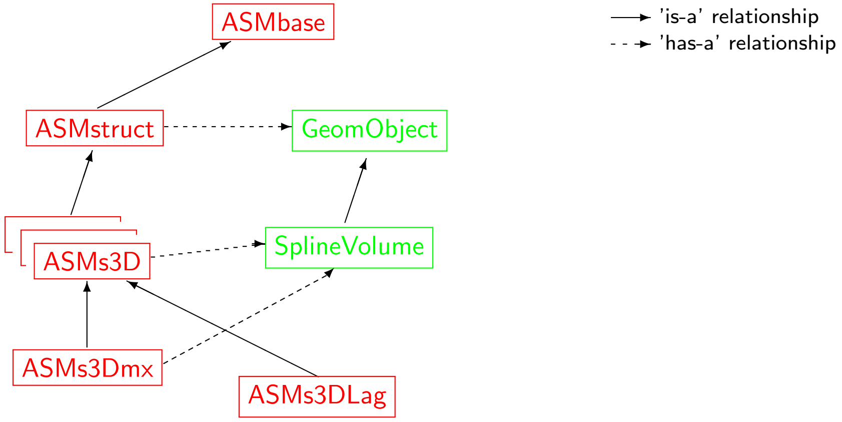
ASM class hierarchy - *the Isogeometry level*



⟶ 'is-a' relationship
⇢ 'has-a' relationship

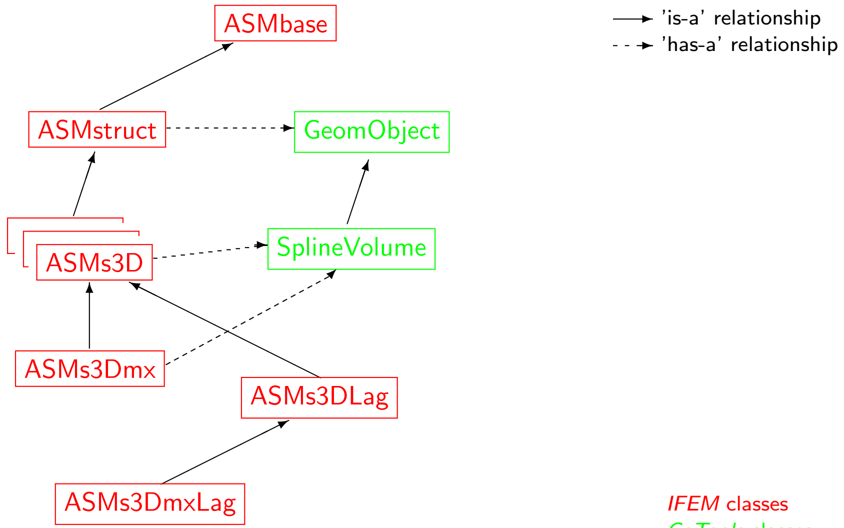
IFEM classes
GoTools classes

ASM class hierarchy - *the Isogeometry level*



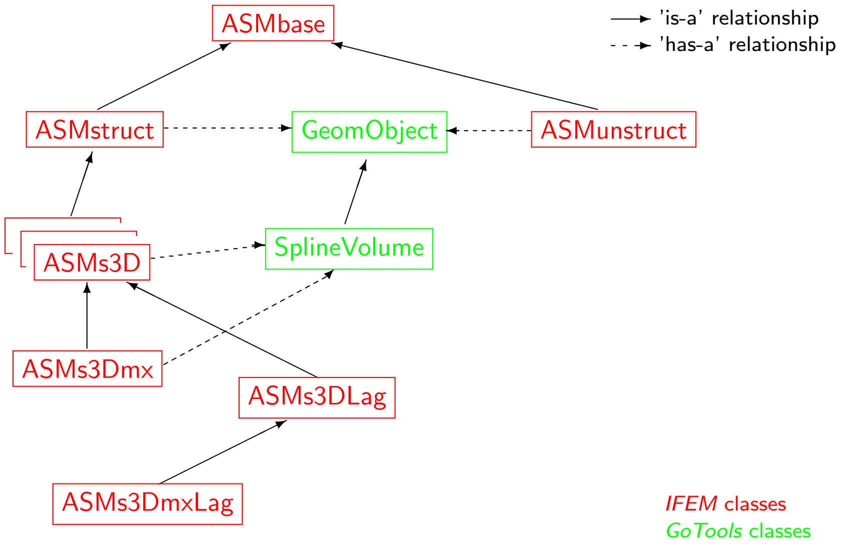
IFEM classes
GoTools classes

ASM class hierarchy - *the Isogeometry level*

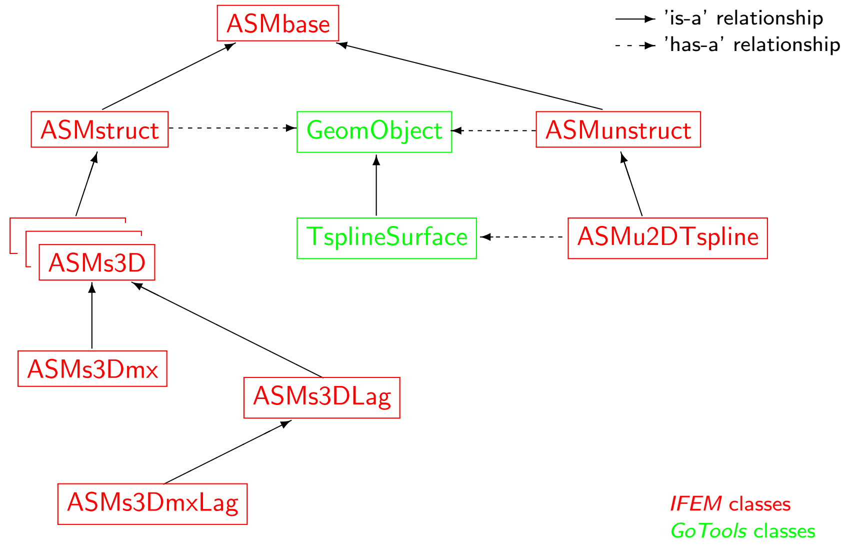


IFEM classes
GoTools classes

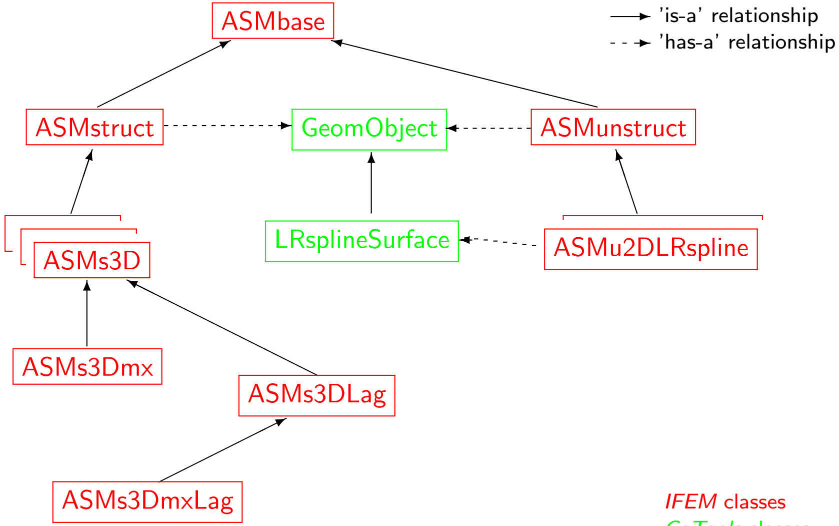
ASM class hierarchy - *the Isogeometry level*



ASM class hierarchy - *the Isogeometry level*

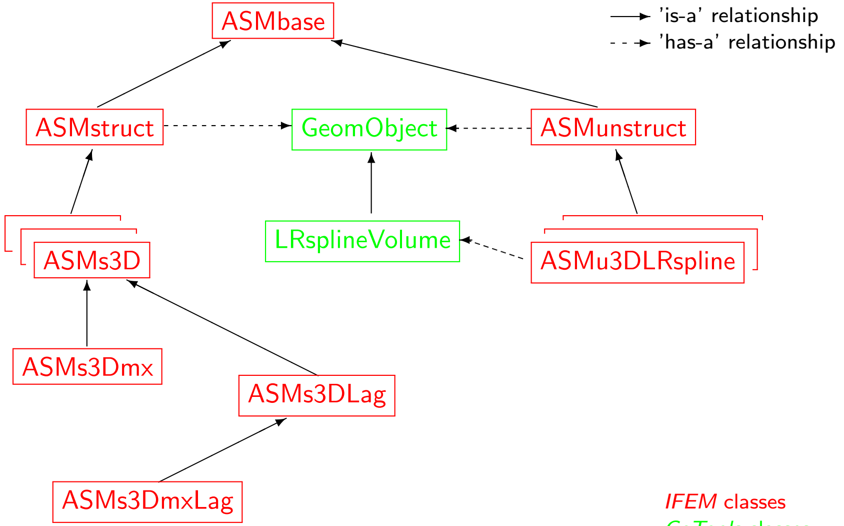


ASM class hierarchy - *the Isogeometry level*



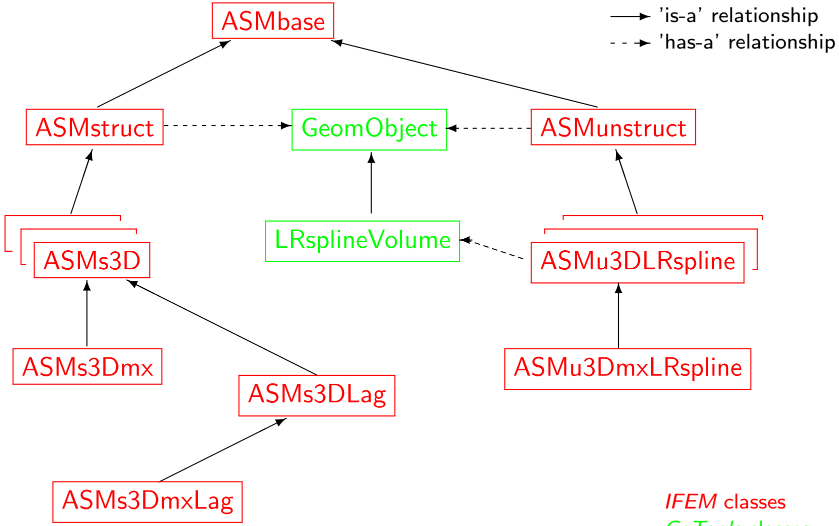
IFEM classes
GoTools classes

ASM class hierarchy - *the Isogeometry level*



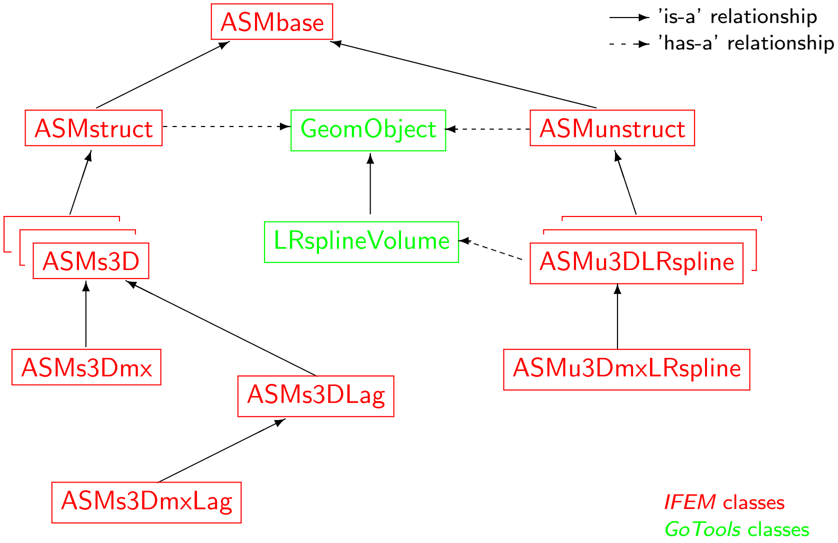
IFEM classes
GoTools classes

ASM class hierarchy - *the Isogeometry level*



IFEM classes
GoTools classes

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

```
typedef std::vector<LocalIntegral*> LintegralVec; //!< Local integral container

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
  //! \param locInt Vector of element-wise contributions to \a glbInt
  virtual bool integrate(Integrand& integrand,
                         GlobalIntegral& glbInt, const TimeDomain& time,
                         const LintegralVec& locInt = LintegralVec()) = 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
  //! \param locInt Vector of element-wise contributions to \a glbInt
  virtual bool integrate(Integrand& integrand, int lIndex,
                         GlobalIntegral& glbInt, const TimeDomain& time,
                         const LintegralVec& locInt = LintegralVec()) = 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, const LintegralVec& locInt)
{
























 }
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time, const LintegralVec& locInt)
{
  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, const LintegralVec& locInt)
{
  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, const LintegralVec& locInt)
{
  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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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(locInt[iel], 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, const LintegralVec& locInt)
{
  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
      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(locInt[iel], time, detJ*weight, N, dN/dX, X);
      end do i, j, k
      integrand.finalizeElement(locInt[iel]);

    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time, const LintegralVec& locInt)
{
  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
      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(locInt[iel], time, detJ*weight, N, dN/dX, X);
      end do i, j, k
      integrand.finalizeElement(locInt[iel]);
      glInt.assemble(locInt[iel], MGEL[iel]);
    end if
  end do iel
}
```

Numerical integration method for a 3D spline patch

```
bool ASMs3D::integrate (Integrand& integrand, GlobalIntegral& glInt,
                        const TimeDomain& time, const LintegralVec& locInt)
{
  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
      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(locInt[iel], time, detJ*weight, N, dN/dX, X);
      end do i, j, k
      integrand.finalizeElement(locInt[iel]);
      glInt.assemble(locInt[iel], MGEL[iel]);
    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
  double    u;      //!< First parameter of current point
  double    v;      //!< Second parameter of current point
  double    w;      //!< Third parameter of current 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
  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
- Current solution: Establish the first basis by order-elevating the second basis once

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
- Current solution: Establish the first basis by order-elevating the second basis once
  - 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
- Current solution: Establish the first basis by order-elevating the second basis once
  - 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
- Current solution: Establish the first basis by order-elevating the second basis once
  - 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.

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
- Current solution: Establish the first basis by order-elevating the second basis once
  - 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.
  - But, we only get $\mathcal{C}^{p-2}$ continuity in the highest-order solution field (p being the polynomial order of the first basis), and $\mathcal{C}^{p-1}$ continuity in the other field.

Framework for two-field mixed formulations

- Two sets of basis functions – the first basis should be of one order higher than the second
- Current solution: Establish the first basis by order-elevating the second basis once
  - 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.
  - But, we only get $\mathcal{C}^{p-2}$ continuity in the highest-order solution field (p being the polynomial order of the first basis), and $\mathcal{C}^{p-1}$ continuity in the other field.
- TODO: Manage bases with $\mathcal{C}^{p-1}$ continuity in both fields.

Numerical integration for the two-field mixed method

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



 }
```

Numerical integration for the two-field mixed method

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

  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, const LintegralVec& locInt)
{
  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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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, const LintegralVec& locInt)
{
  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
      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]; $N_1$, $dN_1/du$, $N_2$, $dN_2/du$





      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, const LintegralVec& locInt)
{
  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
      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]; $N_1$, $dN_1/du$, $N_2$, $dN_2/du$
        Compute Cartesian coordinates and Jacobian; $X = N_2 * Xnod$, $J = dN_2/du * Xnod$
        and the gradients; $dN_1/dX = dN_1/du * J^{-1}$, $dN_2/dX = dN_2/du * J^{-1}$,


      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, const LintegralVec& locInt)
{
  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
      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]; $N_1$, $dN_1/du$, $N_2$, $dN_2/du$
        Compute Cartesian coordinates and Jacobian; $X = N_2*Xnod$, $J = dN_2/du*Xnod$
        and the gradients; $dN_1/dX = dN_1/du * J^{-1}$, $dN_2/dX = dN_2/du * J^{-1}$,
        integrand.evalInt(locInt[iel], time, detJ*weight,
                          $N_1$, $dN_1/dX$, $N_2$, $dN_2/dX$, 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, const LintegralVec& locInt)
{
  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
      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]; $N_1$, $dN_1/du$, $N_2$, $dN_2/du$
        Compute Cartesian coordinates and Jacobian; $X = N_2*Xnod$, $J = dN_2/du*Xnod$
        and the gradients; $dN_1/dX = dN_1/du * J^{-1}$, $dN_2/dX = dN_2/du * J^{-1}$,
        integrand.evalInt(locInt[iel], time, detJ*weight,
                          $N_1$, $dN_1/dX$, $N_2$, $dN_2/dX$, X);
      end do i, j, k
      integrand.finalizeElement(locInt[iel]);
      glInt.assemble(locInt[iel], MGEL[iel]);
    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, Neo-Hookean materials

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, Neo-Hookean materials
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes
    - `NonlinearElasticityULMixed` - ..., mixed formulation with with continuous pressure and volumetric change fields (work in progress)

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, Neo-Hookean materials
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes
    - `NonlinearElasticityULMixed` - ..., mixed formulation with with continuous pressure and volumetric change fields (work in progress)
- `StabilizedStokes` - Pressure-stabilized Stokes, fully coupled
  - `NavierStokesG2` - G2-stabilized Navier–Stokes solver, Euler time integration
  - `NavierStokesG2M2` - ..., Mid-point rule time integration

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, Neo-Hookean materials
    - `NonlinearElasticityULMX` - Incompressible and nearly incompressible materials, mixed formulation with internal pressure and volumetric change modes
    - `NonlinearElasticityULMixed` - ..., mixed formulation with with continuous pressure and volumetric change fields (work in progress)
- `StabilizedStokes` - Pressure-stabilized Stokes, fully coupled
  - `NavierStokesG2` - G2-stabilized Navier–Stokes solver, Euler time integration
  - `NavierStokesG2M2` - ..., Mid-point rule time integration
- Projection-based, decoupled Navier–Stokes solvers (Chorin method)
  - Mixed formulation (work in progress)

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

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 in progress (based on PETSc and 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 Integrand
{
protected:
  // Physical properties
  double    kappa;   //!< Conductivity (constant)
  VecFunc*  fluxFld; //!< Boundary heat flux field
  RealFunc* heatSrc; //!< Interior heat source field

  // Finite element quantities
  Matrix* eM; //!< Element coefficient matrix
  Vector* eS; //!< Element right-hand-side vector
  Vector* eV; //!< Element solution vector

  mutable ElmMats myMats; //!< Local element matrices
```

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

Tutorial: Poisson equation

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

  // Finite element quantities
  Matrix* eM; //!< Element coefficient matrix
  Vector* eS; //!< Element right-hand-side vector
  Vector* eV; //!< Element solution vector

  mutable ElmMats myMats; //!< Local element matrices
```

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

```
public:
  Poisson() : kappa(1.0), fluxFld(0), heatSrc(0)
  {
    primsol.resize(1);
    myMats.A.resize(1);
    myMats.b.resize(2);
    eM = &myMats.A[0];
    eS = &myMats.b[0];
    eV = &myMats.b[1];
  }
  virtual ~Poisson() {}
```

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

Tutorial: Poisson equation

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

Initialization of physical properties.

Tutorial: Poisson equation

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

Initialization of physical properties.

```
virtual bool initElement(const std::vector<int>& MNPC);
virtual bool initElementBou(const std::vector<int>& MNPC);
```

Virtual methods for element initialization during the numerical integration.

Tutorial: Poisson equation

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

Initialization of physical properties.

```
virtual bool initElement(const std::vector<int>& MNPC);
virtual bool initElementBou(const std::vector<int>& MNPC);
```

Virtual methods for element initialization during the numerical integration.

```
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 Vector& N,
                     const Matrix& dNdX,
                     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

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

Initialization of physical properties.

```
    virtual bool initElement(const std::vector<int>& MNPC);
    virtual bool initElementBou(const std::vector<int>& MNPC);
```

Virtual methods for element initialization during the numerical integration.

```
    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 Vector& N,
                         const Matrix& dNdX,
                         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.

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

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

Methods for solution norm integration.

Tutorial: Poisson equation

```
bool Poisson::initElement (const std::vector<int>& MNPC)
{
  const size_t nen = MNPC.size();

  eM->resize(nen,nen,true);
  eS->resize(nen,true);

  int ierr = 0;
  if (!primsol.front().empty())
    if ((ierr = utl::gather(MNPC,1,primsol.front(),*eV)))
      std::cerr <<" *** Poisson::initElement: Detected "
                << ierr <<" node numbers out of range."
                << std::endl;

  myMats.withLHS = true;
  return ierr == 0;
}


bool Poisson::initElementBou (const std::vector<int>& MNPC)
{
  eS->resize(MNPC.size(),true);

  myMats.withLHS = false;
  return true;
}
```

Element initialization:
Set the size of the element matrices based on the number of element nodes.
Extract the element solution vector from the global (patch-level) vector.
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
{
  elmInt = &myMats;

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

  Matrix CB;
  CB.multiply(C,fe.dNdX,false,true).multiply(fe.detJxW);
  eM->multiply(fe.dNdX,CB,false,false,true);

  if (heatSrc)
    eS->add(fe.N,(*heatSrc)(X)*fe.detJxW);

  return true;
}
```

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

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

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

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

```
bool Poisson::evalBou (LocalIntegral*& elmInt,
                       const FiniteElement& fe,
                       const Vec3& X, const Vec3& normal) const
{
  elmInt = &myMats;
  if (!fluxFld) return false;

  Vec3 q = (*fluxFld)(X); // heat flux at point X

  double flux = -(q*normal);
  eS->add(fe.N,flux*fe.detJxW);

  return true;
}
```

$$\{eS\} = \sum(-(\mathbf{q}\cdot\mathbf{n})\{N\}|J|w)$$

Tutorial: Poisson equation

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

  Matrix C;
  this->formCmatrix(C,X);

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

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

  return true;
}
```

Secondary solution evaluation:

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

```
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

```
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));
}


class PoissonNorm : public NormBase
{
  Poisson& problem; //!< The problem-specific data
  VecFunc* anasol;  //!< Analytical heat flux

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

  virtual bool initElement(const std::vector<int>& MNPC)
  {
    return problem.initElement(MNPC);
  }

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

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
{
  ElmNorm* eNorm = dynamic_cast<ElmNorm*>(elmInt);
  if (!eNorm) return false;

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

  // Evaluate the finite element heat flux field
  Vector sigmah;
  if (!problem.evalSol(sigmah,fe.dNdX,X)) return false;

  // Integrate the energy norm a(u^h,u^h)
  ElmNorm& pnorm = *eNorm;
  pnorm[0] += sigmah.dot(Cinv*sigmah)*fe.detJxW;
  if (anasol)
  {
    // Evaluate the analytical heat flux
    Vector sigma((*anasol)(X).ptr(),sigmah.size());
    // Integrate the energy norm a(u,u)
    pnorm[1] += sigma.dot(Cinv*sigma)*fe.detJxW;
    // Integrate the error in energy norm a(u-u^h,u-u^h)
    sigma -= sigmah;
    pnorm[2] += sigma.dot(Cinv*sigma)*fe.detJxW;
  }

  return true;
}
```

Norm integrand evaluation

Tutorial: Poisson equation

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

public:
  SIMPoisson2D() : SIM2D(1), prob(2)
  { myProblem = &prob; }

  virtual ~SIMPoisson2D()
  { myProblem = 0; }

protected:
  virtual bool parse(char* keyWord, std::istream& is);
  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)
{
  VecFuncMap::const_iterator tit = myVectors.find(propInd);
  if (tit == myVectors.end()) return false;
  prob.setTraction(tit->second);
  return true;
}
```

Simulation driver class

Tutorial: Poisson equation

```
bool SIMPoisson2D::parse (char* keyWord, std::istream& is)
{
  char* cline = 0;
  if (!strncasecmp(keyWord,"ISOTROPIC",9))
  {
    int nmat = atoi(keyWord+10);
    std::cout <<"\nNumber of isotropic materials: "<< nmat << std::endl;
    for (int i = 0; i < nmat && (cline = utl::readLine(is)); i++)
    {
      int    code  = atoi(strtok(cline," "));
      double kappa = atof(strtok(NULL," "));
      std::cout <<"\tMaterial code "<< code <<": "<< kappa << std::endl;
      if (code == 0)
        prob.setMaterial(kappa);
      else if (this->setPropertyType(code,Property::MATERIAL,mVec.size()))
        mVec.push_back(kappa);
    }
  }
  else if (!strncasecmp(keyWord,"SOURCE",6))
  {
    cline = strtok(keyWord+6," ");
    if (!strncasecmp(cline,"SQUARE",6))
    {
      double L = atof(strtok(NULL," "));
      std::cout <<"\nHeat source function: Square L="<< L << std::endl;
      prob.setSource(new Square2DHeat(L));
    }
    else
      std::cerr <<"  ** SIMPoisson2D::parse: Unknown source function "
                << cline << std::endl;
  }
```

Tutorial: Poisson equation

```
    else if (!strncasecmp(keyWord,"ANASOL",6))
    {
      cline = strtok(keyWord+6," ");
      if (!strncasecmp(cline,"SQUARE",6))
      {
        double L = atof(strtok(NULL," "));
        std::cout <<"\nAnalytical solution: Square L="<< L << std::endl;
        mySol = new AnaSol(NULL,new Square2D(L));
      }
      else if (!strncasecmp(cline,"LSHAPE",6))
      {
        mySol = new AnaSol(NULL,new LshapePoisson());
        std::cout <<"\nAnalytical solution: Lshape"<< std::endl;
      }
      else
      {
        std::cerr <<"  ** SIMPoisson2D::parse: Unknown analytical solution "
                  << cline <<" (ignored)"<< std::endl;
        return true;
      }

      // Define the analytical boundary traction field
      int code = (cline = strtok(NULL," ")) ? atoi(cline) : 0;
      if (code > 0 && mySol->getScalarSecSol())
      {
        this->setPropertyType(code,Property::NEUMANN);
        myVectors[code] = mySol->getScalarSecSol();
      }
    }
    else
      return this->SIM2D::parse(keyWord,is);

    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();
  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

```
PATCHFILE    lshape2d.g2
PROPERTYFILE lshape2d.prc

RAISEORDER 1
# patch ru rv
  1     2  2

REFINE 1
# patch ru rv
  1     7  7

DIRICHLET 1
# code
  1

# Analytical solution
# Specifier   code
ANASOL Lshape 2
```

lshape2d.prc:

```
1 0 1 2
2 0 1 0
2 0 1 1
2 0 1 3
```

lshape2d.g2:

```
200 1 0 0
3 0
7 4
0 0 0 0 1 1 1 2 2 2 2
4 4
0 0 0 0 1 1 1 1

0.000000 -1.000000 0
0.000000 -0.666667 0
0.000000 -0.333333 0
0.000000 -0.000000 0
0.333333 -0.000000 0
0.666667 -0.000000 0
1.000000 -0.000000 0
-0.333333 -1.000000 0
-0.333333 -0.555556 0
-0.333333 -0.111111 0
-0.333333 0.333333 0
0.111111 0.333333 0
0.555556 0.333333 0
1.000000 0.333333 0
-0.666667 -1.000000 0
-0.666667 -0.444444 0
-0.666667 0.111111 0
-0.666667 0.666667 0
```

+ 10 more lines