ICADA: Class hierarchies of the **SIM**ulation module

Knut Morten Okstad

January 21, 2010

1 Introduction

This document intends to give a brief overview of the class hierarchies of the simulation module of the isogeometric analysis toolbox, developed for the ICADA project. A prototype code for 3D linear elasticity problems has been developed in order to gain experience in using splines or NURBS as basis functions in place of standard Lagrange polynomials. A framework for general FE analysis based on splines is then developed, where the main point is that the user should be able to create his/her own simulators of a PDE-governed problem, by creating only a few classes representing the weak form of the mathematical problem at hand.

2 The main class hierarchies

The prototype code consists mainly of two classes (in addition to some utility classes and linear algebra methods). These are **LinearEl** on the system level, and **VolumePatch** on the local (patch) level. In the new version, these two classes are replaced by several class hierarchies for increased flexibility, and with the ability for code reuse on a broader range of applications. The main hierarchies are the following:

SIM (base class: `SIMbase`) Simulation drivers administrating an assembly of spline patches. Solution algorithms for time stepping and path-following methods, etc., will also be implemented withing this hierarchy.

ASM (base class: `ASMbase`) Administers numerical integration and FE assembly over a patch. Each patch has a pointer to a GoTools `GeomObject` instance representing the actual geometry and basis functions of that patch.

Integrand (base class: `Integrand`) Represents the actual problem to be solved with physical property parameters and methods for evaluating the integrand of the governing weak form problem. An instance of this class is passed as argument to the integration methods of the ASM-classes.

Integral (base classes: `LocalIntegral, GlobalIntegral`) Represents the integrated quantities on element and system level, respectively. An instance of `GlobalIntegral` is passed as argument to the integration methods of the ASM-classes, which then updates it with contributions from that patch.

SystemMatrix (base classes: `SystemMatrix, SystemVector`) Interface to different matrix and vector representations required by the linear equation solvers to be used.

2.1 SIM

SIMbase ⟵ **SIM3D** ⟵ **SIMLinEl3D**

The **SIMbase** class replaces the **LinearEl** class of the prototype. Its main methods and data are indicated in the class definition below (refer to the source code or the doxygen-generated documentation for complete definition):

```
class SIMbase
{
public:
  //! \brief Reads model data from the specified input file \a *fileName.
  bool read(const char* fileName);

  //! \brief Administers assembly of the linear equation system.
  //! \param[in] prevSol Previous primary solution vector in DOF-order
  bool assembleSystem(const Vector* prevSol = 0);

  //! \brief Solves the assembled linear system of equations for a given load.
  //! \param[out] solution Global primary solution vector
  //! \param[in] printSol Print solution if its size is less than \a printSol
  bool solveSystem(Vector& solution, int printSol = 0);

  //! \brief Integrates some solution norm quantities.
  //! \details If an analytical solution is provided, norms of the exact
  //! error in the solution are computed as well.
  //! \param[in] psol Global primary solution vector
  //! \param[out] eNorm Element-wise norm quantities
  //! \param[out] gNorm Global norm quantities
  bool solutionNorms(const Vector& psol, Matrix& eNorm, Vector& gNorm);

  //! \brief Performs a generalized eigenvalue analysis of the assembled system.
  //! \param[in] iop Which eigensolver method to use
  //! \param[in] nev Number of eigenvalues/vector (see ARPack documentation)
  //! \param[in] ncv Number of Arnoldi vectors (see ARPack documentation)
  //! \param[in] shift Eigenvalue shift
  //! \param[out] solution Computed eigenvalues and associated eigenvectors
  //! \param[in] iA Index of system matrix \b A in \a myEqSys->A
  //! \param[in] iB Index of system matrix \b B in \a myEqSys->A
  bool systemModes(std::vector<Mode>& solution,
                   int nev, int ncv, int iop, double shift,
                   size_t iA = 0, size_t iB = 1);

protected:
  //! \brief Parses a data section from an input stream.
  //! \param[in] keyWord Keyword of current data section to read
  //! \param is The file stream to read from
  virtual bool parse(char* keyWord, std::istream& is) = 0;

  // Model attributes
  std::vector<ASMbase*> myModel;   //!< The actual NURBS/spline model
  PropertyVec           myProps;   //!< Physical property mapping
  Integrand*            myProblem; //!< Problem-specific data and methods

private:
  // Solver attributes
  SAMpatch*    mySam;   //!< Data for FE assembly management
  AlgEqSystem* myEqSys; //!< The linear equation system
};
```

The **SIM3D** class is a common base for drivers of 3D continuum problems, whereas **SIMLinEl3D** is a sample driver for 3D linear elasticity problems. Similar drivers might be needed for 2D problems and shell models. Basically, the subclasses only need to (re)implement the `parse` method, which parses the problem-specific data from an input file.

2.2 ASM

ASMbase ⟵ **ASMstruct** ⟵ **ASMs3D**

The **ASMbase** class replaces the **VolumePatch** class of the prototype. However, it is more generic and contains nothing specific to the actual problem to be solved, nor does it pay attention to the spatial discretization method (structured or unstructured), or the spatial dimensions of the problem.

The subclass **ASMstruct** is a common base for patches with structured discretizations, that is, the "nodal points" are organized in a topological pattern such that the local node number can be computed knowing only its index along each parameter direction along with the number of nodes in each direction.

Later, an accompanying class **ASMunstruct** is to be made, serving as a base for adaptive meshes based on T-splines and similar techniques.

The sub-class **ASMs3D** is for trivariate structured patches. It has a pointer to an underlying **SplineVolume** object from the *GoTools* library, containing all information needed to represent the trivariate spline basis for the geometry and solution fields Later, **ASMs2D** and **ASMs1D** for two-parametric and single-parametric patches will be added.

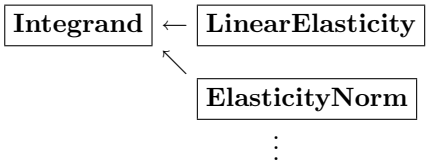
The computational core of the ASM-hierarchy is in the data and methods reproduced in the class definition below (see source code or doxygen documentation for complete class definition):

```
class ASMbase
{
public:
  // Methods for integration of finite element quantities.

  //! \brief Evaluates an integral over the interior patch domain.
  //! \param integrand Object with problem-specific data and methods
  //! \param glbInt The integrated quantity
  //! \param locInt Vector of element-wise contributions to \a glbInt
  virtual bool integrate(Integrand& integrand,
                         GlobalIntegral& glbInt,
                         const LintegralVec& locInt = LintegralVec()) = 0;

  //! \brief Evaluates a boundary integral over a patch face.
  //! \param integrand Object with problem-specific data and methods
  //! \param[in] face Local index of the boundary face
  //! \param glbInt The integrated quantity
  //! \param locInt Vector of element-wise contributions to \a glbInt
  virtual bool integrate(Integrand& integrand, short int face,
                         GlobalIntegral& glbInt,
                         const LintegralVec& locInt = LintegralVec()) = 0;
protected:
  // Standard finite element data structures
  unsigned char ndim; //!< Number of parametric dimensions (1, 2 or 3)
  unsigned char nsd;  //!< Number of space dimensions (ndim <= nsd <= 3)

  IntVec MLGE;  //!< Matrix of Local to Global Element numbers
  IntVec MLGN;  //!< Matrix of Local to Global Node numbers
  IntMat MNPC;  //!< Matrix of Nodal Point Correspondence
  BCVec  BCode; //!< Vector of Boundary condition codes
  MPCSet mpcs;  //!< All multi-point constraints with slave in this patch
};
```

The two `integrate` methods perform numerical integration over the interior of the patch, and over one of its boundary faces, respectively. The integrand and the resulting integrated quantity are provided as arguments to the methods through virtual interface classes, such that the integration methods themselves can be kept purely problem-independent.

The `integrate` methods are implemented in the **ASMs3D** sub-class where they can be optimized for 3D structured grids. Other integration methods and/or unstructured grid representations can be incorporated by deriving another subclasses of **ASMbase** where `integrate` is re-implemented.

2.3 Integrand



The purpose of the **Integrand** class is to serve as a generic representation of the mathematical problem to be solved. It mainly contains virtual methods for evaluation of the weak from of the problem, one method for interior terms, and another method for boundary terms. It also has a method for evaluating secondary solution variables when the primary solution is known. All methods take FE quantities at current integration point as arguments. The methods are invoked from the `integrate` methods of the **ASM**-classes.

```
class Integrand
{
public:
  //! \brief Evaluates the integrand at an interior point.
  //! \param elmInt The local integral object to receive the contributions
  //! \param[in] detJW Jacobian determinant times integration point weight
  //! \param[in] N Basis function values
  //! \param[in] dNdX Basis function gradients
  //! \param[in] X Cartesian coordinates of current integration point
  virtual bool evalInt(LocalIntegral*& elmInt, double detJW,
                       const Vector& N, const Matrix& dNdX,
                       const Vec3& X) const { return false; }

  //! \brief Evaluates the integrand at a boundary point.
  //! \param elmInt The local integral object to receive the contributions
  //! \param[in] detJW Jacobian determinant times integration point weight
  //! \param[in] N Basis function values
  //! \param[in] dNdX Basis function gradients
  //! \param[in] X Cartesian coordinates of current integration point
  //! \param[in] normal Boundary normal vector at current integration point
  virtual bool evalBou(LocalIntegral*& elmInt, double detJW,
                       const Vector& N, const Matrix& dNdX,
                       const Vec3& X, const Vec3& normal) const { return false; }

  //! \brief Evaluates the secondary solution at current integration point.
  //! \param[out] s The solution field values
  //! \param[in] N Basis function values
  //! \param[in] dNdX Basis function gradients
  //! \param[in] X Cartesian coordinates of current integration point
  //! \param[in] MNPC Matrix of nodal point correspondence
  virtual bool evalSol(Vector& s,
                       const Vector& N, const Matrix& dNdX, const Vec3& X,
                       const std::vector<int>& MNPC) const { return false; }

  //! \brief Returns a pointer to an Integrand for solution norm evaluation.
  //! \param[in] asol Pointer to the analytical solution field (optional)
  virtual Integrand* getNormIntegrand(TensorFunc* asol = 0) const { return 0; }

protected:
  Vector primsol; //!< Current primary solution vector for this patch
};
```

A sample class **LinearElasticy** is provided, which can be used as a template for other, similar problem classes, like Poisson problem, Navier-Stokes, etc. **ElasticityNorm** is an accompanying class, used for integrating solution norms which depend on problem parameters. It requires a reference to a **LinearElasticity** object for access of problem parameters, but by implementing the norm integrand as a separate **Integrand** class, it can be evaluated using the same methods in the **ASM** and **SIM** hierarchies, as when assembling the finite element matrices.

2.4 LocalIntegral and GlobalIntegral

The classes **LocalIntegral** and **GlobalIntegral** are generic interfaces to integrated quantities over a single element, and over the global domain, respectively. We need these interfaces in order to make the `integrate` implementations of the **ASM**-classes independent on the type of quantity that is integrated. The **LocalIntegral** class has no methods or data itself, whereas **GlobalIntegral** has one virtual method or assembling a local quantity into its corresponding global quantity:

```
class LocalIntegral
{
protected:
  //! \brief The default constructor is protected to allow sub-classes only.
  LocalIntegral() {}
public:
  //! \brief Empty destructor.
  virtual ~LocalIntegral() {}
};

class GlobalIntegral
{
protected:
  //! \brief The default constructor is protected to allow sub-classes only.
  GlobalIntegral() {}

public:
  //! \brief Empty destructor.
  virtual ~GlobalIntegral() {}

  //! \brief Adds a LocalIntegral object into a corresponding global object.
  //! \param[in] elmObj The local integral object to add into \a *this.
  //! \param[in] elmId Global number of the element associated with elmObj
  virtual bool assemble(const LocalIntegral* elmObj, int elmId) = 0;
};
```

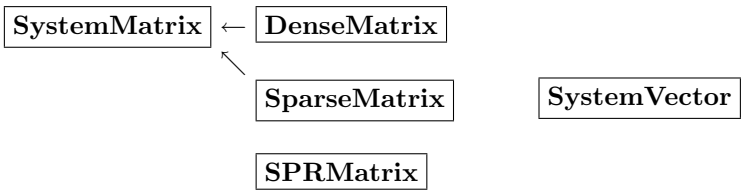
The **ElmMats** sub-class contains just a set of element matrices and vectors, and has an accompanying global equivalent in the **AlgEqSystem** class. These classes are used when assembling linear system of equations resulting from the FE discretization.

Similarly, the class **ElmNorm** is just a vector of local norm quantities and has its global equivalent in the **GlbBNorm** class. Their class definitions are particularly simple and are reproduced below:

```
class ElmNorm : public LocalIntegral
{
public:
  //! \brief The constructor assigns the internal pointer.
  ElmNorm(double* p) : ptr(p) {}
  //! \brief Empty destructor.
  virtual ~ElmNorm() {}

  //! \brief Indexing operator for assignment.
  double& operator[](size_t i) { return ptr[i]; }
  //! \brief Indexing operator for referencing.
  const double& operator[](size_t i) const { return ptr[i]; }

private:
  double* ptr; //!< Pointer to the actual norm values
};
```

```
class GlbNorm : public GlobalIntegral
{
public:
  //! \brief The constructor initializes a reference to the global norm vector.
  GlbNorm(std::vector<double>& vec) : myVals(vec) {}
  //! \brief Empty destructor.
  virtual ~GlbNorm() {}

  //! \brief Adds element norm quantities into the global norm object.
  //! \param[in] elmObj Pointer to the element norms to add into \a *this
  //! \param[in] elmId Global number of the element associated with \a *elmObj
  virtual bool assemble(const LocalIntegral* elmObj, int elmId);

private:
  std::vector<double>& myVals; //!< Reference to a vector of global norm values
};
```

2.5 SystemMatrix (and SystemVector)



The **SystemMatrix** are not changed since the prototype version. However, a new **SystemVector** class is introduced, and used in place of **Vector** when it represents the right-hand-side vector of a global equation system. This class may then be a base class for later parallel version.