How to use BaseMatrix and BaseVector.

A way to store, centralize and manipulate mechanical data in solvers

At that time, we have chosen to store the mechanical global matrices and vectors in OdeSolvers. The interface required to use a specific matrix library as a SOFA mechanical matrix is basic. You just need to overload:

- a resize method
- a dimension accessor
- a way to read / write floating point data corresponding to indices.

This interface is described in the following UML diagram. This diagram represents a way to use the NewMat library dense Matrix and Vector. Tthe resolution methods are implemented in the Matrix implementation class, where we know precisely the kind of data we are manipulating.

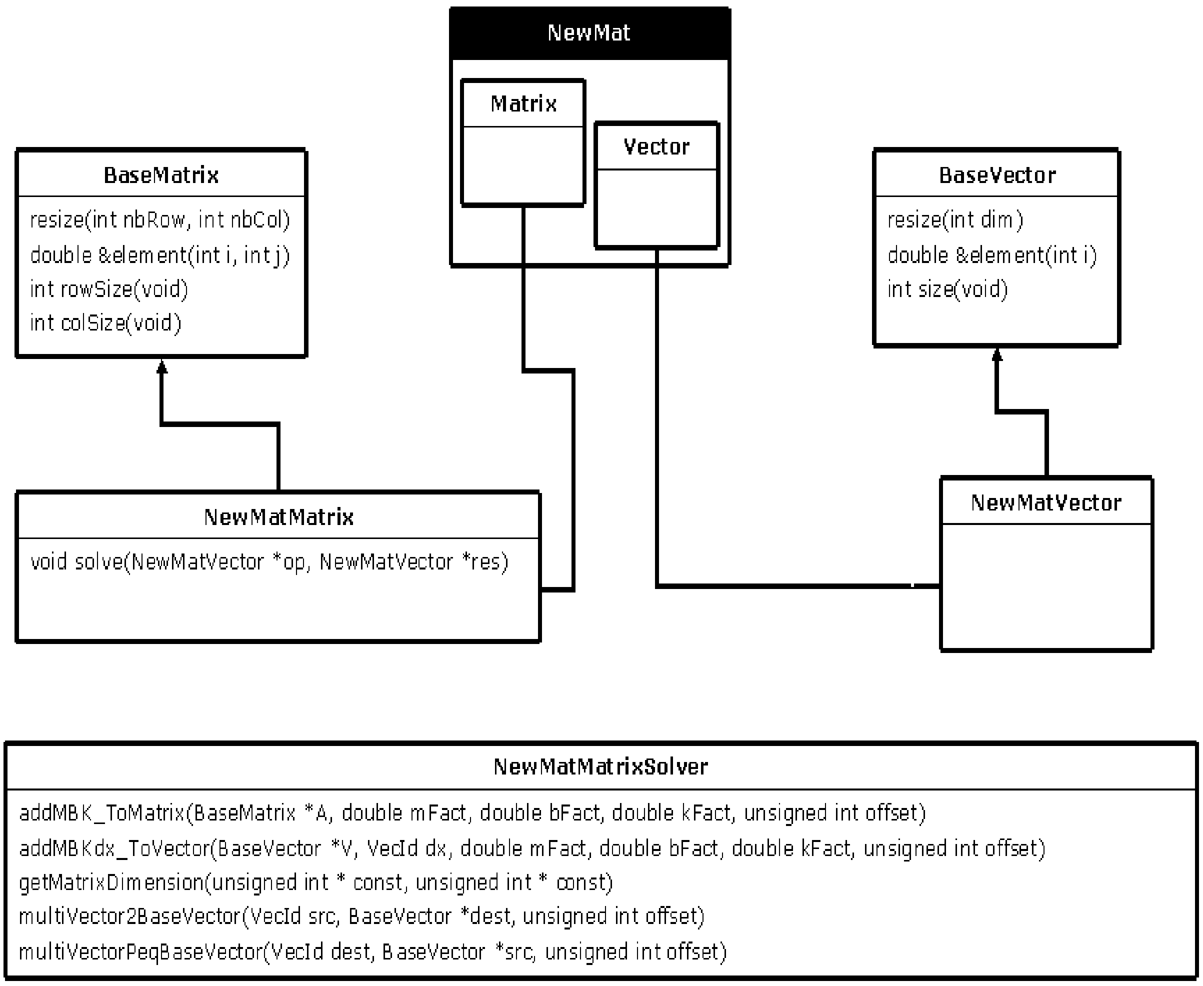
NewMat
Matrix
Vector
BaseMatrix
resize(int nbRow, int nbCol)
double &element(int i, int j)
int rowSize(void)
int colSize(void)
BaseVector
resize(int dim)
double &element(int i)
int size(void)
NewMatMatrix
void solve(NewMatVector *op, NewMatVector *res)
NewMatVector
NewMatMatrixSolver
addMBK_ToMatrix(BaseMatrix *A, double mFact, double bFact, double kFact, unsigned int offset)
addMBKdx_ToVector(BaseVector *V, VecId dx, double mFact, double bFact, double kFact, unsigned int offset)
getMatrixDimension(unsigned int * const, unsigned int * const)
multiVector2BaseVector(VecId src, BaseVector *dest, unsigned int offset)
multiVectorPeqBaseVector(VecId dest, BaseVector *src, unsigned int offset)

Illustration 1: UML Diagram of the NewMat Library SOFA Interface for Dense Matrices

As shown in illustration 1, BaseMatrix and BaseVector are currently managed by 5 methods in the solver:

`getMatrixDimension` returns the global matrix dimension of the mechanical system described in the OdeSolver sub-tree.

`addMBK_ToMatrix` and **`addMBKdx_ToVector`** allow the “mechanical components” to bring their contribution respectively to the BaseMatrix and BaseVector. A scalar factor for the mass, the damping and the stiffness is available to parametrize the components contribution.

`multiVector2BaseVector` is used to create a BaseVector corresponding to an abstract multiVector.

`multiVectorPeqBaseVector` is used to add a BaseVector to a multiVector. Basically when you are computing dx = K-1 * f, it allows you to apply easily the solution to the position multiVector.

The corresponding interfaces in the mechanical components are the following:

```
BaseMass
addMToMatrix(BaseMatrix * mat, double mFact, unsigned int offset)
addMDxToVector(BaseVector * resVect, const VecDeriv& dx, double mFact, unsigned
int offset)

BaseForceField
addKToMatrix(BaseMatrix * mat, double kFact, unsigned int offset)
addKDxToVector(BaseVector * resVect, const VecDeriv& dx, double kFact, unsigned
int offset)

BaseConstraint
applyConstraint(BaseMatrix *mat, unsigned int offset)
applyConstraint(defaulttype::BaseVector *vect, unsigned int offset)
```

The offset parameter gives the beginning of the current block in the matrix. The offset update is made automatically by the MechanicalMatrixVisitor.

A concrete example :
The Static Resolution of a tetrahedron FEM using MKL dense Matrices

Currently, we have 2 static solvers using dense matrices in SOFA. An MKL Sparse Matrix Solver will come very soon. These 2 solvers are implementing the same integration and resolution method. One is using a BaseMatrix implementation based on the NewMat Library, the other on the MKL.

For the MKL, here is the code for the Matrix. Data is stored in a Dynamic_Matrix<double> (an homemade Class chosen for its efficiency). The element method is specific for the MKL resolution method **`dgesv`**. Indeed, this method needs the transposed Matrix to solve the system Ax=b, so here the **`element()`** method access the data in a “transposing way”.

```
class MKLMatrix : public BaseMatrix
{
public:

	MKLMatrix() { impl = new Dynamic_Matrix<double>; }

	virtual ~MKLMatrix() { delete impl; }

	virtual void resize(int nbRow, int nbCol) { impl->resize(nbRow, nbCol); }

	virtual int rowSize(void) { return impl->rows; }

	virtual int colSize(void) { return impl->columns; }

	virtual double &element(int i, int j) { return *(impl->operator[](j) + i); }

	virtual void solve(MKLVector *rHTerm)
	{
		int n=impl->rows;
		int nrhs=1;
		int lda = n;
		int ldb = n;
```

```
                int info;
                int *ipiv = new int[n];

                // solve Ax=b
                // b is overwritten by the linear system solution
                dgesv(&n,&nrhs,impl->m,&lda,ipiv,rHTerm->impl->v,&ldb,&info);
        }


private:
        Dynamic_Matrix<double> *impl;
};
```

Then the solver code. In this example, we wanted to have exclusively the stiffness in the global matrix and the external forces in the global vector (basically the Weight). You can see in the TetrahedronFemForceField the implementation of addKToMatrix and in UniformMass, the addMDxToVector method.

```
MKLSolver::MKLSolver()
{
        globalStiffnessMatrix = new MKLMatrix();
        externalForcesGlobalVector = new MKLVector();

        updateMatrix = true;

        nbRow = 0;
        nbCol = 0;
}

MKLSolver::~MKLSolver()
{
        delete globalStiffnessMatrix;
        delete externalForcesGlobalVector;
}

void MKLSolver::solve(double dt)
{
        OdeSolver* group = this;
        MultiVector f(group, VecId::force());
        MultiVector pos(group, VecId::position());
        MultiVector dx(group, VecId::null());

        if (this->updateMatrix)
        {
                getMatrixDimension(&nbRow, &nbCol);
                globalStiffnessMatrix->resize(nbRow, nbCol);
                externalForcesGlobalVector->resize(nbRow);
                /* Compute and assemble the global "Mechanical" Matrix threw forcefield,
masses... Then applies baseConstraints such as fixed point constraints */
                addMBK_ToMatrix(globalStiffnessMatrix,0,0,1);

                /* Compute and assemble the global "Mechanical" Vector threw forcefield,
masses... Then applies baseConstraints such as fixed point constraints */
                addMBKdx_ToVector(externalForcesGlobalVector,dx,1,0,0);

                /* Solve dx = Fext / K */
                globalStiffnessMatrix->solve(externalForcesGlobalVector);

                /* Update position using the solver result */
                multiVectorPeqBaseVector(pos, externalForcesGlobalVector);

                /* This solver is static and should be solved only one time */
                updateMatrix = false;
        }
}
```

Hope this helps...