# TNL - Template Numerical Library

**Tomáš Oberhuber**, Vítězslav Žabka, Vladimír Klement

tomas.oberhuber@fjfi.cvut.cz

Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering,
Czech Technical University in Prague

## Overview

We present new numerical library designed to profit from C++ templates. It supports computations on GPU via Nvidia CUDA. In near future, we plan to release version 0.1 which targets to solution of **non-linear partial differential equations**. To solve the non-linear problems efficiently, fast assembling of the linear systems in each time step is necessary. Therefore our library provides **uniform interface for several formats for sparse matrices** which is accessible even from the CUDA kernels. It significantly simplifies development of the non-linear solvers.

## Aims of TNL project

We want our library to be **F.R.E.E.** – **F**lexible, **R**eliable, **E**fficient and **E**asy to use.

1. **F**lexible -researchers in numerical analysis and computer simulations often must try many different approaches and methods before they find the successful one. A library, which meets these requirements, must be flexible and it must allow to do large changes with small effort. C++ templates are appropriate tool for this task.
2. **R**eliable - numerical simulations must be, of course, correct. To eliminate as many bugs in our code as possible, we write number of tests and we compare our implementation with other libraries.
3. **E**fficient - computer simulations can be very time consuming. To speed things up, TNL supports computations on GPUs. C++ templates help to generate highly optimized code.
4. **E**asy to use - even though TNL combines technologies like C++ templates and GPU computations, it should be available even to people without deep knowledge of C++ and GPU.

## Vectors

Vectors are represented by a template class:
**tnlVector< Real = double, Device = tnlHost, Index = int >**
where

- `Real` is precision of the floating point arithmetics
- `Device` is device where the vector is allocated ( `tnlHost` or `tnlCuda`)
- `Index` is type for indexing the vector elements

Example:

```
/* ***
 * Allocate and setup vectors
 */
tnlVector< double, tnlHost > xHost;
tnlVector< double, tnlCuda > xDevice, yDevice;
xHost.setSize( 1000 );
xHost.setValue( 1.0 ) ;
xDevice = xHost;
yDevice.setLike( xDevice );
yDevice.setValue( 2.0 );

/* ***
 * Compute y = 0.6 * y + 2.0 * x
 */
yDevice.addVector( xDevice, 2.0, 0.6 );

/* ***
 * Compute scalar product of yDevice and xDevice
 */
double s = yDevice.scalarProduct( xDevice );

/* ***
 * Compute L2 norm of yDevice
 */
double l2_norm = yDevice.lpNorm( 2.0 );

/* ***
 * Save yDevice to file
 */
yDevice.save( "y.tnl");
```

## Matrices

Matrices are represented as:
**tnl*Format*Matrix<Real = double, Device = tnlHost, Index = int>**
The following formats are supported on both CPU and GPU:

- Dense
- Tridiagonal
- Multidiagonal
- Ellpack
- Sliced Ellpack [1] (published as Row-grouped CSR format)
- Chunked Ellpack [2] (published as Improved Row-grouped CSR format)
- CSR

To initialize the sparse matrix formats like (Sliced/Chunked) Ellpack or CSR, one must compute the number of the nonzero elements in each row. We refer it as row length:

```
typedef tnlCSRMatrix< double, tnlCuda, int > Matrix;
Matrix matrix;
matrix.setDimensions( 1000, 1000 );
typename Matrix::RowLengthsVector rowLengths;
rowLengths.setSize( 1000 );
computeRowLengths( rowLengths );
matrix.setRowLengths( rowLengths );
```

To set the nonzero elements of the matrix, we first transfer the matrix object instance to the GPU (we transfer only metadata not the matrix elements – they are already on the GPU):

```
Matrix* deviceImage = passToDevice( matrix );
dim3 blockSize( 256 ), gridSize( 4 );
int sharedMemory = ...
assemblyMatrix<<< gridSize,
                  blockSize,
                  sharedMemory >>>(deviceImage,...);
freeFromDevice( matrix );
```

The kernel `assemblyMatrix` may look like:

```
template< typename Matrix >
__global__ assemblyMatrix( Matrix* matrix, .... )
{
   int* columns[];
   double* values[];
   int rowIdx = blockIdx.x*blockDim.x+threadIdx.x;
   if( rowIdx < matrix->getRows() )
   {
      int elements =
         computeRow(rowIdx,columns,values);
      matrix->setRowFast(rowIdx,columns,
                         values,elements );
   }
}
```

## Solvers

The following **solvers of the linear systems** are supported:

- SOR,
- CG
- BiCGStab
- GMRES [4]
- TFQMR

Preconditioners are not implemented yet.
The following **explicit solvers for PDEs** are supported:

- Euler (first order)
- Runge-Kutta-Merson (fourth order) with time step adaptivity [3,4]

## Time-dependent PDE solver

TNL contains `tnlSolver` class which prepares the necessary framework for the user. The `main` function is very simple:

```
int main( int argc, char* argv[] )
{
   tnlSolver< simpleProblem > solver;
   if( ! solver. run( CONFIG_FILE, argc, argv ) )
      return EXIT_FAILURE;
   return EXIT_SUCCESS;
}
```

The `simpleProblem` is a template implementing the problem being solved:

```
template< typename Mesh >
bool simpleProblemSolver< Mesh >::
   init( const tnlParameterContainer& config )
{
   /* ***
    * Set-up your solver here:
    * 1. Read input parameters and model
    *    coefficients like these
    */
   const tnlString& problemName =
      config.getParameter<tnlString>("problem");

   /* ***
    * 2. Set-up the mesh.
    */
   const tnlString& meshFile =
      config.getParameter<tnlString>("mesh");
   if( ! this->mesh.load( meshFile ) )
      return false;

   /* ***
    * 3. Set-up DOFs
    */
   const IndexType& dofs = this->mesh.getDofs();
   this->dofVector.setSize(dofs);
   return true;
}

template< typename Mesh >
bool simpleProblemSolver<Mesh>::setInitialCondition(
   const tnlParameterContainer& config )
{
   if( ! this->dofVector.load("initial-cond"))
      return false;
   return true;
}
```

```
template< typename Mesh >
bool simpleProblemSolver<Mesh>::makeSnapshot(
    const RealType& time, const IndexType& step )
{
   cout << "Writing output at time "
        << time << " step " << step;
   tnlString fileName( ... );
   if( ! this->dofsVector.save(fileName))
      return false;
   return true;
}

template< typename Mesh >
typename simpleProblemSolver<Mesh>::DofVectorType&
    simpleProblemSolver<Mesh>::getDofVector()
{
   return dofVector;
}

template< typename Mesh >
void simpleProblemSolver< Mesh >::
   getExplicitRHS( const RealType& time,
                   const RealType& tau,
                   DofVectorType& u,
                   DofVectorType& fu )
{
   /* ***
    * Compute the right-hand side of
    *
    *   d/dt u(x) = fu( x, u ).
    */

   if( DeviceType::getDevice() == tnlHostDevice )
   {
      ...
   }
   if( DeviceType::getDevice() == tnlCudaDevice )
   {
      ...
   }
}

template< typename Mesh >
void simpleProblemSolver< Mesh >::
   assemblyMatrix( RealType& time,
                   const RealType& tau,
                   DofVectorType& u,
                   MatrixType& matrix )
{
   if( DeviceType::getDevice() == tnlHostDevice )
   {
      ...
   }
   if( DeviceType::getDevice() == tnlCudaDevice )
   {
      ...
   }
}
```

## Future plans

TNL experimentaly supports:

- high-precision arithmetics
- structured and unstructured numerical meshes

We aim to implement efficient solvers for the Navier-Stokes equations. We plan the following milestones:

- unstructured meshes and finite element methods (version 0.2)
- multigrid methods (version 0.3)
- computations on multi-GPU systems and clusters of GPUs (version ???)

## References

[1] Oberhuber, T.; Suzuki, A. and Vacata, J. *New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA Acta Technica*, 2011, 56, pp. 447-466.

[2] Heller M., Oberhuber T., *Improved Row-grouped CSR Format for Storing of Sparse Matrices on GPU*, Proceedings of Algoritmy 2012, 2012, Handlovičová A., Minarechová Z. and Ševčovič D. (ed.), pp. 282-290.

[3] Oberhuber, T.; Suzuki, A. and Žabka, V., *The CUDA implementation of the method of lines for the curvature dependent flows*, Kybernetika, 2011, 47, pp. 251-272.

[4] Oberhuber T., Suzuki A., Vacata J., Žabka V., *Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods*, Journal of Math-for-Industry, 2011, vol. 3, pp. 73–79 .

TNL will be available soon at

**https://code.google.com/p/tnl**