Advanced Programming Techniques
ExAdvPT
Winter Term 2015/2016

Assignment sheet B

Assignments that are marked with *StudOn submission* are **mandatory** and must be submitted via StudOn in time – please see the StudOn page for deadlines.

Fun with matrices

Your successful implementation of last assignment's matrix class has attracted some attention of students from other departments. In detail, some guys from the math division seem to have a use for it. They claim, that they have a solver for Partial Differential Equations (PDEs) that is just missing a well-written matrix class. (... and a vector class, but that is merely a generalization of the matrix class.) In their current project, they solve a 1D finite differences discretization of Poisson's equation

$$\Delta u = b,$$

$$b = sin(2\pi x).$$

Fortunately, you don't need to go into the math too deep as there is already a setup routine and a solver implementation. Your task is finishing what they started.

1 Coding: Setting up the solver

Start by downloading the solver implementation (Solver.cpp) which will serve as test case for your implementations. As you can see, the initialization of the utilized matrix and vectors is already present and doesn't have not be adapted. The solver is also fully functional, that is if suitable Matrix and Vector classes are provided.

mandatory Set up a vector class (hint: this should be quite similar to a (Nx1) matrix)
mandatory Check that you vector and matrix implementations include all required functionalities. This will most likely require the implementation of three additional functions:

- Matrix::inverseDiagonal which calculates the inverse of the diagonal of itself. It is sufficient to implement this function for square matrices (remember suitable asserts). Please also refrain from implementing a general matrix inversion!
- Multiplication of a matrix and a vector. The function signature should look like this:

```
Vector<T> operator* (const Vector<T> & o) const;
```

- Vector::l2Norm, which calculates the L2 norm of itself. An initial implementation of Vector::l2Norm could look like this:

```
double l2Norm ( ) const {
  double norm = 0.;
  for (int i = 0; i < size_; ++i)
    norm += data_[i] * data_[i];
  return sqrt(norm);
}
```

optional Think about possible enhancements and use std::accumulate to realize a much more concise implementation.

2 Checking results

StudOn submission

To roughly verify the results of your implementation run the solver for varying problem sizes (17, 33, 65, 129). Afterwards, submit the required number of iterations for each test case via StudOn.

3 Coding: Checking performance

Your colleagues are happy with your implementation, but somehow you are not satisfied. The main reason is that you suspect untapped performance possibilities. Have a look at std::chrono and use its functionality to time the solver in order to assess its performance.

4 Coding: Improving performance

As you expect, you find out that the solver is quite slow and, even worse, scales horribly. After a quick analysis, you identify the matrix operations as a possible cause. Moreover, the matrix in the sample problem has a very special structure[1] but this is not exploited at all! As a possible counter measure, you devise a way of improving performance by introducing a stencil class. However, you don't want to break the general applicability of the solver which is why you decide to add a common interface in the form of an abstract base class.

mandatory Download the initial interface implementation (MatrixLike.h) from StudOn and extend your Matrix class to implement the interface (i.e. derive from MatrixLike). Also change the solve function to accept a reference to a MatrixLike instance:

```
void solve (const MatrixLike<T, MatrixImpl>& A,
            const Vector<T>& b, Vector<T>& u)
```

If you measure your solver again, you will most likely observe a severe performance penalty. Think about possible reasons and adapt the MatrixLike interface accordingly. (hint: there are two ways of doing this and one of them is simply removing the () operator from the interface.)

After fixing the interface, it is time to implement the Stencil class.

mandatory Download the skeleton (Stencil.h) from StudOn and implement the prepared functions. Implement the testStencil function in Solver.cpp and verify that the results of the solver (i.e. printed

[1] en.wikipedia.org/wiki/Tridiagonal_matrix

residuals and iteration counts) do not change when using the Stencil class instead of the Matrix class.

optional Make use of std::find_if when implementing Stencil::inverseDiagonal to identify the entry with zero offset.

5 Coding: Checking performance ... again

After successfully implementing the stencil class it is time to compare execution times. Think about possible reasons for observed differences.

6 Coding: Improving usability

As a last step, you decide to extend the Vector class for improved usability.

optional Implement a constructor for the Vector class taking a std::function used to initialize its contents. This eliminates the old initialization routine

```
Vector<double> b(numGridPoints, 0.);
for (int x = 0; x < numGridPoints; ++x) {
  b(x) = sin(2. * PI * (x / (double)(numGridPoints - 1)));
}
```

and replaces it with a new expression based on a lambda expression, e.g.

```
Vector<double> b(numGridPoints, /*insert lambda here*/ );
```

7 Wrapping up

StudOn submission

Finally, add your solutions for Matrix.h, Vector.h and Stencil.h as well as your adaptations of Solver.cpp and MatrixLike.h to an archive and upload it to StudOn.