```cpp
#include "benchmarks.h"
#include "vdop.h"
#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>    // assert()


#ifdef __INTEL_CLANG_COMPILER
#pragma message(" ##########  Use of MKL  ###############")
#include <mkl.h>
#else
#pragma message(" ##########  Use of CBLAS  ###############")

#include <cblas.h>                // cBLAS Library
#include <lapacke.h>              // Lapack

#endif



// (A) Inner product of two vectors (from skalar_stl)
double scalar(vector<double> const &x, vector<double> const &y)
{
    assert(x.size() == y.size());
    size_t const N = x.size();
    double sum = 0.0;
    for (size_t i = 0; i < N; ++i)
    {
        sum += x[i] * y[i];
    }
    return sum;
}

// (A) 5.(b) Kahan scalar product
double Kahan_skalar(vector<double> const &x, vector<double> const &y)
{
    double sum = 0.0;
    double c = 0.0;
    size_t n = x.size();
    for (size_t i = 0; i < n; ++i)
    {
        double z = x[i]*y[i] - c;    // c is the part that got lost in the last iter
ation
        double t = sum + z;          // when adding sum + z, the lower digits are lo
st if sum is large
        c = (t - sum) - z;           // now we recover the lower digits to add in th
e next iteration
        sum = t;
    }
    return sum;
}

// (A) 6. cBLAS scalar product
double scalar_cBLAS(vector<double> const &x, vector<double> const &y)
{
    return cblas_ddot(x.size(), x.data(), 1, y.data(), 1); // x.data() = &x[0]
}


// (B) Matrix-vector product (from intro_vector_densematrix)
vector<double> MatVec(vector<double> const &A, vector<double> const &x)
{
    size_t const nelem = A.size();
    size_t const N = x.size();
    assert(nelem % N == 0); // make sure multiplication is possible
    size_t const M = nelem/N;

```

```
   66:
   67:     vector<double> b(M);
   68:
   69:     for (size_t i = 0; i < M; ++i)
   70:     {
   71:         double tmp = 0.0;
   72:         for (size_t j = 0; j < N; ++j)
   73:             tmp += A[N*i + j] * x[j];
   74:         b[i] = tmp;
   75:     }
   76:
   77:     return b;
   78: }
   79:
   80: // (B) cBLAS Matrix-vector product
   81: vector<double> MatVec_cBLAS(vector<double> const &A, vector<double> const &x)
   82: {
   83:     size_t const nelem = A.size();
   84:     size_t const N = x.size();
   85:     assert(nelem % N == 0); // make sure multiplication is possible
   86:     size_t const M = nelem/N;
   87:
   88:
   89:     vector<double> b(M);
   90:
   91:     cblas_dgemv(CblasRowMajor, CblasNoTrans, M, N, 1.0, A.data(), N, x.data(), 1, 0.
0, b.data(), 1);
   92:
   93:     return b;
   94: }
   95:
   96:
   97: // (C) Matrix-matrix product
   98: vector<double> MatMat(vector<double> const &A, vector<double> const &B, size_t const
 &L)
   99: {
  100:     size_t const nelem_A = A.size();
  101:     size_t const nelem_B = B.size();
  102:
  103:     assert(nelem_A % L == 0 && nelem_B % L == 0);
  104:
  105:     size_t const M = nelem_A/L;
  106:     size_t const N = nelem_B/L;
  107:
  108:
  109:     vector<double> C(M*N);
  110:
  111:
  112:
  113:     for (size_t i = 0; i < M; ++i)
  114:     {
  115:         for (size_t j = 0; j < N; ++j)
  116:         {
  117:             double C_temp = 0;
  118:             for (size_t k = 0; k < L; ++k)
  119:             {
  120:                 C_temp += A[L*i + k]*B[N*k + j];
  121:             }
  122:             C[N*i + j] = C_temp;
  123:         }
  124:     }
  125:
  126:     return C;
  127: }
  128:
  129: // (C) cBLAS matrix-matrix product
  130: vector<double> MatMat_cBLAS(vector<double> const &A, vector<double> const &B, size_t
 const &L)
```

swap loops. should be faster

```
131: {
132:     size_t const nelem_A = A.size();
133:     size_t const nelem_B = B.size();
134:
135:     assert(nelem_A % L == 0 && nelem_B % L == 0);
136:
137:     size_t const M = nelem_A/L;
138:     size_t const N = nelem_B/L;
139:
140:
141:     vector<double> C(M*N);
142:
143:     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, N, L, 1.0, A.data(), L
, B.data(), N, 0.0, C.data(), N);
144:
145:     return C;
146: }
147:
148:
149: // (D) Evaluation of a polynomial function
150: vector<double> poly(vector<double> const &a, vector<double> const &x)
151: {
152:     size_t const N = x.size();
153:     size_t const p = a.size() - 1;
154:     vector<double> y(N, 0);
155:
156:     for (size_t i = 0; i < N; ++i)
157:     {
158:         double x_temp = x[i];
159:         double y_temp = 0;
160:         for (size_t k = 0; k < p + 1; ++k)
161:         {
162:             y_temp += x_temp*y_temp + a[p - k];
163:         }
164:         y[i] = y_temp;
165:     }
166:
167:     return y;
168: }
169:
170:
171: // (E) Solves linear system of equations
172: void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
173: {
174:     const double omega   = 1.0;
175:     const int    maxiter = 1000;
176:     const double tol  = 1e-5,                // tolerance
177:                  tol2 = tol * tol;           // tolerance^2
178:
179:     int nrows = SK.Nrows();                  // number of rows == number of columns
180:     assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );
181:
182:     cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s"  << endl;
183:     //  Choose initial guess
184:     for (int k = 0; k < nrows; ++k)
185:     {
186:         u[k] = 0.0;                          //  u := 0
187:     }
188:
189:     vector<double> dd(nrows);                // matrix diagonal
190:     vector<double>  r(nrows);                // residual
191:     vector<double>  w(nrows);                // correction
192:
193:     SK.GetDiag(dd);                          //  dd := diag(K)
194:     ////DebugVector(dd);{int ijk; cin >> ijk;}
195:
196:     //  Initial sweep
197:     SK.Defect(r, f, u);                      //  r := f - K*u
```

```
  198:
  199:     vddiv(w, r, dd);                          //  w := D^{-1}*r
  200:     double sigma0 = dscapr(w, r);             // s0 := <w,r>
  201:
  202:     // Iteration sweeps
  203:     int iter  = 0;
  204:     double sigma = sigma0;
  205:     while ( sigma > tol2 * sigma0 && maxiter > iter)
  206:     {
  207:         ++iter;
  208:         vdaxpy(u, u, omega, w );              //  u := u + om*w
  209:         SK.Defect(r, f, u);                   //  r := f - K*u
  210:         vddiv(w, r, dd);                      //  w := D^{-1}*r
  211:         sigma = dscapr(w, r);                 // s0 := <w,r>
  212: //              cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
  213:     }
  214:     cout << "aver. Jacobi rate :  " << exp(log(sqrt(sigma / sigma0)) / iter) << "  (
" << iter << " iter)" << endl;
  215:     cout << "final error: " << sqrt(sigma / sigma0) << " (rel)   " << sqrt(sigma) <<
 " (abs)\n";
  216:
  217:     return;
  218: }
  219:
  220:
  221:
  222:
  223:
  224:
  225:
  226:
  227:
  228:
  229:
  230:
  231:
  232:
  233:
  234:
  235:
  236:
  237:
  238:
  239:
  240:
  241:
  242:
  243:
  244:
  245:
  246:
```

```
    1: #pragma once
    2: #include "getmatrix.h"
    3: #include <vector>
    4:
    5: using namespace std;
    6:
    7: /**     (A) Inner product of two vectors (from skalar_stl)
    8:         @param[in] x    vector
    9:         @param[in] y    vector
   10:         @return             resulting Euclidian inner product <x,y>
   11: */
   12: double scalar(vector<double> const &x, vector<double> const &y);
   13:
   14: /**     (A) 5.(b) Inner product of two vectors using the Kahan scalar product
   15:         @param[in] x    vector
   16:         @param[in] y    vector
   17:         @return             resulting Euclidian inner product <x,y>
   18: */
   19: double Kahan_skalar(vector<double> const &x,  vector<double> const &y);
   20:
   21: /**     (A) 6. cBLAS scalar product of two vectors
   22:         @param[in] x    vector
   23:         @param[in] y    vector
   24:         @return             resulting Euclidian inner product <x,y>
   25: */
   26: double scalar_cBLAS(vector<double> const &x, vector<double> const &y);
   27:
   28:
   29: /**     (B) Matrix-vector product (from intro_vector_densematrix)
   30:  *      @param[in] A    dense matrix (1D access)
   31:  *  @param[in] u        vector
   32:  *
   33:  *      @return    resulting vector
   34: */
   35: vector<double> MatVec(vector<double> const &A, vector<double> const &x);
   36:
   37: /**     (B) 6. cBLAS Matrix-vector product
   38:  *      @param[in] A    dense matrix (1D access)
   39:  *  @param[in] u        vector
   40:  *
   41:  *      @return    resulting vector
   42: */
   43: vector<double> MatVec_cBLAS(vector<double> const &A, vector<double> const &x);
   44:
   45:
   46: /**     (C) Matrix-matrix product
   47:  *      @param[in] A                    MxL dense matrix (1D access)
   48:  *  @param[in] B                        LxN dense matrix (1D access)
   49:  *  @param[in] shared_dim       shared dimension L
   50:  *
   51:  *      @return    resulting MxN matrix
   52: */
   53: vector<double> MatMat(vector<double> const &A, vector<double> const &B, size_t const
 &shared_dim);
   54:
   55: /**     (C) 6. cBLAS Matrix-matrix product
   56:  *      @param[in] A                    MxL dense matrix (1D access)
   57:  *  @param[in] B                        LxN dense matrix (1D access)
   58:  *  @param[in] shared_dim       shared dimension L
   59:  *
   60:  *      @return    resulting MxN matrix
   61: */
   62: vector<double> MatMat_cBLAS(vector<double> const &A, vector<double> const &B, size_t
 const &shared_dim);
   63:
   64:
   65: /**     (D) Evaluation of a polynomial function using Horner's scheme
   66:  *      @param[in] a    coefficient vector
```

```
   67:  *  @param[in] x        vector with input values
   68:  *
   69:  *      @return    vector with output values
   70: */
   71: vector<double> poly(vector<double> const &a, vector<double> const &x);
   72:
   73:
   74: /**     (E) Solves linear system of equations  K @p u = @p f  via the Jacobi iterati
on (from jaboci_oo_stl)
   75:  * We use a distributed symmetric  CSR matrix @p SK and initial guess of the
   76:  * solution is set to 0.
   77:  * @param[in] SK        CSR matrix
   78:  * @param[in] f         distributed local vector storing the right hand side
   79:  * @param[out] u        accumulated local vector storing the solution.
   80: */
   81: void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u);
   82:
   83:
   84:
   85:
   86:
   87:
   88:
   89:
```

```cpp
#include "benchmark_tests.h"
#include "benchmarks.h"
#include <chrono>
#include <iostream>
#include <math.h>
using namespace std::chrono;

vector<double> test_A(const size_t &NLOOPS, const size_t &N, const function<double(const vector<double>&, const vector<double>&)>& scalar_function)
{
    cout << "#################### (A) ####################" << endl;
    cout << "\nLOOPS = " << NLOOPS << endl;
    cout << "\nN = " << N << endl;


//  Memory allocation
    cout << "Memory allocation\n";

    vector<double> x(N), y(N);

    cout.precision(2);
    cout << 2.0*N *sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory allocated\n";
    cout.precision(6);


//  Data initialization
//  Special:  x_i = i+1;  y_i = 1/x_i  ==> <x,y> == N

    for (size_t i = 0; i < N; ++i)
    {
        x[i] = i % 219 + 1;
        y[i] = 1.0/x[i];
    }


    cout << "\nStart Benchmarking scalar\n";

    auto t1 = system_clock::now(); // start timer
// Do calculation
    double check(0.0),ss(0.0);
    for (size_t i = 0; i < NLOOPS; ++i)
    {
        check = scalar_function(x, y);
        ss += check;                    // prevents the optimizer from removing unused calculation results.
    }

    auto t2 = system_clock::now();  // stop timer
    auto duration = duration_cast<microseconds>(t2 - t1);        // duration in microseconds
    double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration in seconds
    t_diff = t_diff/NLOOPS;                                      // duration per loop seconds


// Check the correct result
    cout << "\n <x,y> = " << check << endl;
    if (static_cast<unsigned int>(check) != N)
        cout << "  !!   W R O N G  result   !!\n";
    cout << endl;


// Timings  and Performance
    cout << endl;
    cout.precision(2);

```

```
    64:
    65:     double Gflops = 2.0*N / t_diff / 1024 / 1024 / 1024;
    66:     double MemBandwidth = 2.0*N / t_diff / 1024 / 1024 / 1024 * sizeof(x[0]);
    67:
    68:     cout << "Total duration : " << t_diff*NLOOPS << endl;
    69:     cout << "Timing in sec. : " << t_diff << endl;
    70:     cout << "GFLOPS         : " << Gflops << endl;
    71:     cout << "GiByte/s       : " << MemBandwidth << endl;
    72:
    73: //#########################################################################
    74:     cout << "\nStart Benchmarking norm\n";
    75:
    76:     auto t3 = system_clock::now(); // start timer
    77: // Do calculation
    78:     double ss2(0.0);
    79:     for (size_t i = 0; i < NLOOPS; ++i)
    80:     {
    81:         auto sk1 = sqrt(scalar(x, x));
    82:         ss2 += sk1;                    // prevents the optimizer from removing unused
 calculation results.
    83:     }
    84:
    85:     auto t4 = system_clock::now();  // stop timer
    86:     auto duration2 = duration_cast<microseconds>(t4 - t3);        // duration in mic
roseconds
    87:     double t_diff2 = static_cast<double>(duration2.count()) / 1e6; // overall durati
on in seconds
    88:     t_diff2 = t_diff2/NLOOPS;                                      // duration per l
oop seconds
    89:
    90:
    91:     cout << "ss(norm): " << ss2 << endl;
    92:     cout << "Timing in sec. : " << t_diff2 << endl;
    93:
    94:
    95:
    96:
    97:
    98:
    99:     return vector<double>{t_diff, Gflops, MemBandwidth};
   100: }
   101:
   102:
   103: vector<double> test_B(const size_t &NLOOPS, const size_t &N, const size_t &M, const
function<vector<double>(const vector<double>&, const vector<double>&)>& MatVec_function)
   104: {
   105:     cout << "#################### (B) ####################" << endl;
   106:
   107:     cout << "\nLOOPS = " << NLOOPS << endl;
   108:     cout << "\nN = " << N << endl;
   109:     cout << "\nM = " << M << endl;
   110:
   111: //  Memory allocation
   112:     cout << "Memory allocation\n";
   113:
   114:     vector<double> A(M*N);
   115:     vector<double> x(N);
   116:
   117:     cout.precision(2);
   118:     cout << (1.0*M*N + N) * sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory allo
cated\n";
   119:     cout.precision(6);
   120:
   121: //  Data initialization
   122:
   123:     for (size_t i = 0; i < M; ++i)
   124:         for (size_t j = 0; j < N; ++j)
   125:             A[N*i + j] = (i + j) % 219 + 1;
```

```
  126:
  127:
  128:     for (size_t j = 0; j < N; ++j)
  129:     {
  130:         x[j] = 1.0/A[N*17 + j];
  131:     }
  132:
  133:     cout << "\nStart Benchmarking MatVec\n";
  134:
  135:     auto t1 = system_clock::now(); // start timer
  136: // Do calculation
  137:     vector<double> b(M);
  138:
  139:     for (size_t i = 0; i < NLOOPS; ++i)
  140:     {
  141:         b = MatVec_function(A, x);
  142:     }
  143:
  144:     auto t2 = system_clock::now();  // stop timer
  145:     auto duration = duration_cast<microseconds>(t2 - t1);        // duration in micr
oseconds
  146:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  147:     t_diff = t_diff/NLOOPS;                                      // duration per loo
p seconds
  148:
  149:
  150: // Check the correct result
  151:     cout << "\n <A[17,*],x> = " << b[17] << endl;
  152:     if (static_cast<size_t>(b[17]) != N)
  153:     {
  154:         cout << "  !!   W R O N G  result   !!\n";
  155:     }
  156:     cout << endl;
  157:
  158:
  159: // Timings  and Performance
  160:     cout << endl;
  161:     cout.precision(2);
  162:
  163:     double Gflops = (2.0*N*M) / t_diff / 1024 / 1024 / 1024;
  164:     double MemBandwidth = (2.0*N*M + M)/ t_diff / 1024 / 1024 / 1024 * sizeof(x[0]);
  165:
  166:     cout << "Total duration : " << t_diff*NLOOPS << endl;
  167:     cout << "Timing in sec. : " << t_diff << endl;
  168:     cout << "GFLOPS         : " << Gflops << endl;
  169:     cout << "GiByte/s       : " << MemBandwidth << endl;
  170:
  171:
  172:
  173:     return vector<double>{t_diff, Gflops, MemBandwidth};
  174: }
  175:
  176:
  177: vector<double> test_C(const size_t &NLOOPS, const size_t &L, const size_t &M, const
size_t &N, const function<vector<double>(const vector<double>&, const vector<double>&, size
_t const &shared_dim)>& MatMat_function)
  178: {
  179:     cout << "#################### (C) ####################" << endl;
  180:     cout << "\nLOOPS = " << NLOOPS << endl;
  181:     cout << "\nL = " << L << endl;
  182:     cout << "\nM = " << M << endl;
  183:     cout << "\nN = " << N << endl;
  184:
  185:
  186: //  Memory allocation
  187:     cout << "Memory allocation\n";
  188:
```

```
  189:     vector<double> A(M*L);
  190:     vector<double> B(L*N);
  191:
  192:     cout.precision(2);
  193:     cout << (1.0*M*L + L*N) *sizeof(A[0]) / 1024 / 1024 / 1024 << " GByte Memory all
ocated\n";
  194:     cout.precision(6);
  195:
  196:
  197: //  Data initialization
  198:
  199:     for (size_t i = 0; i < M; ++i)
  200:         for (size_t k = 0; k < L; ++k)
  201:             A[L*i + k] = (i + k) % 219 + 1;
  202:
  203:     for (size_t k = 0; k < L; ++k)
  204:         for (size_t j = 0; j < N; ++j)
  205:             B[N*k + j] = 1.0/A[L*17 + k];
  206:
  207:
  208:     cout << "\nStart Benchmarking MatMat\n";
  209:
  210:     auto t1 = system_clock::now(); // start timer
  211: // Do calculation
  212:     vector<double> C(M*N);
  213:     double check;
  214:     double check_sum;= 0.0;
  215:
  216:     for (size_t i = 0; i < NLOOPS; ++i)
  217:     {
  218:         C = MatMat_function(A, B, L);
  219:
  220:         check = C[N*17];
  221:         check_sum += check; // prevents the optimizer from removing unused calculati
on results.
  222:     }
  223:     cout << check_sum;
  224:
  225:     auto t2 = system_clock::now();  // stop timer
  226:     auto duration = duration_cast<microseconds>(t2 - t1);         // duration in micr
oseconds
  227:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  228:     t_diff = t_diff/NLOOPS;                                      // duration per loo
p seconds
  229:
  230:
  231: // Check the correct result
  232:     cout << "\n C[17,0] = " << check << endl;
  233:     if (static_cast<unsigned int>(check) != L)
  234:     {
  235:         cout << "  !!   W R O N G  result   !!, should be " << L <<"\n";
  236:     }
  237:     cout << endl;
  238:
  239: // Timings  and Performance
  240:     cout << endl;
  241:     cout.precision(2);
  242:
  243:
  244:     double Gflops = (2.0*L*N*M) / t_diff / 1024 / 1024 / 1024;
  245:     double MemBandwidth = (2.0*L*N*M + M*N)/ t_diff / 1024 / 1024 / 1024 * sizeof(A[
0]);
  246:
  247:     cout << "Total duration : " << t_diff*NLOOPS << endl;
  248:     cout << "Timing in sec. : " << t_diff << endl;
  249:     cout << "GFLOPS         : " << Gflops << endl;
  250:     cout << "GiByte/s       : " << MemBandwidth << endl;
```

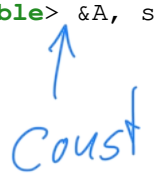
```
  251:
  252:
  253:
  254:     return vector<double>{t_diff, Gflops, MemBandwidth};
  255: }
  256:
  257:
  258: vector<double> test_D(const size_t &NLOOPS, const size_t &N, const size_t &p)
  259: {
  260:     cout << "#################### (D) ####################" << endl;
  261:     cout << "\nLOOPS = " << NLOOPS << endl;
  262:     cout << "\nN = " << N << endl;
  263:     cout << "\np = " << p << endl;
  264:
  265: //  Memory allocation
  266:     cout << "Memory allocation\n";
  267:
  268:     vector<double> a(p + 1, 0);
  269:     vector<double> x(N);
  270:
  271:     cout.precision(2);
  272:     cout << (1.0*(p + 1) + N) *sizeof(x[0]) / 1024 / 1024 / 1024 << " GByte Memory a
llocated\n";
  273:     cout.precision(6);
  274:
  275: //  Data initialization
  276:
  277:     for (size_t j = 0; j < N; ++j)
  278:         x[j] = 1.0*j;
  279:
  280:     for (size_t k = 0; k < p + 1; ++k)
  281:         a[k] = pow(-1.0, k);          // poly(x) = 1 - x + x^2 - x^3 + x^4 - ...
  282:
  283:
  284:
  285:     cout << "\nStart Benchmarking poly\n";
  286:
  287:     auto t1 = system_clock::now(); // start timer
  288: // Do calculation
  289:     vector<double> y(N);
  290:     double check;
  291:     double check_sum;
  292:
  293:     for (size_t i = 0; i < NLOOPS; ++i)
  294:     {
  295:         y = poly(a, x);
  296:         check = y[0];
  297:
  298:         check_sum += check; // prevents the optimizer from removing unused calculati
on results.
  299:     }
  300:
  301:     auto t2 = system_clock::now();  // stop timer
  302:     auto duration = duration_cast<microseconds>(t2 - t1);         // duration in micr
oseconds
  303:     double t_diff = static_cast<double>(duration.count()) / 1e6; // overall duration
 in seconds
  304:     t_diff = t_diff/NLOOPS;                                      // duration per loo
p seconds
  305:
  306:
  307:
  308: // Check the correct result
  309:     cout << "\n poly(" << x[0] << ") = " << check << endl;
  310:     if (abs(check - 1.0) > 1.0/1e6)
  311:     {
  312:         cout << "  !!   W R O N G  result   !!\n";
  313:     }
```

```
  314:     cout << endl;
  315:
  316:
  317: // Timings  and Performance
  318:     cout << endl;
  319:     cout.precision(2);
  320:
  321:
  322:     double Gflops = (N*(p + 1)*3.0) / t_diff / 1024 / 1024 / 1024;
  323:     double MemBandwidth = (N*(2.0 + 3.0*(p + 1)))/ t_diff / 1024 / 1024 / 1024 * siz
eof(x[0]);
  324:
  325:     cout << "Total duration : " << t_diff*NLOOPS << endl;
  326:     cout << "Timing in sec. : " << t_diff << endl;
  327:     cout << "GFLOPS         : " << Gflops << endl;
  328:     cout << "GiByte/s       : " << MemBandwidth << endl;
  329:
  330:
  331:
  332:     return vector<double>{t_diff, Gflops, MemBandwidth};
  333: }
```

```
    1: #pragma once
    2: #include <vector>
    3: #include <functional>
    4: using namespace std;
    5:
    6:
    7:
    8:
    9: vector<double> test_A(const size_t &NLOOPS, const size_t &N, const function<double(c
onst vector<double>&, const vector<double>&)>& scalar_function);
   10:
   11: vector<double> test_B(const size_t &NLOOPS, const size_t &N, const size_t &M, const
function<vector<double>(const vector<double>&, const vector<double>&)>& MatVec_function);
   12:
   13: vector<double> test_C(const size_t &NLOOPS, const size_t &L, const size_t &M, const
size_t &N, const function<vector<double>(const vector<double>&, const vector<double>&, size
_t const &shared_dim)>& MatMat_function);
   14:
   15: vector<double> test_D(const size_t &NLOOPS, const size_t &N, const size_t &p);
```

```cpp
#include "factorization_solve.h"
#include <vector>
#include <math.h> <cmath>
#include <assert.h>
#include <lapack.h>
#include <iostream>

using namespace std;


void factorization_solve(vector<double> &A, vector<double> &b, const int32_t &n_rhs)
{
    int32_t nelem = A.size();
    int32_t N = sqrt(nelem);
    assert(N == static_cast<int32_t>(b.size())/n_rhs);

    vector<int> IPIV(N); // pivot indices
    int info_factorization;

    dgetrf_(&N, &N, A.data(), &N, IPIV.data(), &info_factorization);

    const char transp = 'N';
    int info_solve;
    dgetrs_(&transp, &N, &n_rhs, A.data(), &N, IPIV.data(), b.data(), &N, &info_solv
e, 1); // 1 is length of parameter 'N'

}

void print_matrix(vector<double> &A, size_t M, size_t N)
{
    for (size_t i = 0; i < M; ++i)
    {
        for (size_t j = 0; j < N; ++j)
        {
            cout << A[N*i + j] << "\t";
        }
        cout << endl;
    }
    cout << endl;
}
```

```
    1: #pragma once
    2: #include <vector>
    3: #include <cstdint>
    4: using namespace std;
    5:
    6:
    7: /**     Solve linear system of equations with multiple right hand sides using LU fac
torization
    8:  *      @param[inout] A          NxN Matrix (1D access), gets modified to con
tain the LU decomposition of A
    9:  *  @param[inout] B              N x n_rhs Matrix of right-hand-sides (1D acc
ess), gets modified to contain the solution vectors x
   10:  *  @param[in] n_rhs        number of right-hand-sides b
   11:  *
   12: */
   13: void factorization_solve(vector<double> &A, vector<double> &b, const int32_t &n_rhs)
;
   14:
   15: /**     Print a matrix to console
   16:  *      @param[in] A                MxN Matrix (1D access)
   17:  *  @param[in] M            rows
   18:  *  @param[in] N            columns
   19:  *
   20: */
   21: void print_matrix(vector<double> &A, size_t M, size_t N);
```



```cpp
#include "factorization_solve_tests.h"
#include "factorization_solve.h"
#include "benchmarks.h"
#include <chrono>
#include <iostream>

using namespace std::chrono;

void CheckCorrectness()
{
    cout.precision(4);

    size_t N = 5;
    size_t n_rhs = 5;
    vector<double> A(N*N);
    vector<double> b(N*n_rhs);

    // initialize A
    for (size_t i = 0; i < N; ++i)
    {
        for (size_t j = 0; j < N; ++j)
            if (i == j)
                A[N*i + j] = 4.0;
            else
                A[N*i + j] = 1.0/((i - j)*(i - j));
    }
    const vector<double> A_copy = A;

    // initialize b as identity matrix
    for (size_t j = 0; j < N; ++j)
    {
        for (size_t l = 0; l < n_rhs; ++l)
            if (l == j)
                b[N*l + j] = 1.0;
            else
                b[N*l + j] = 0.0;
    }

    cout << "A = " << endl;
    print_matrix(A, N, N);
    cout << "b = " << endl;
    print_matrix(b, N, n_rhs);

    // solve system
    factorization_solve(A, b, n_rhs);

    vector<double> check_matrix_identity = MatMat_cBLAS(A_copy, b, N);
    cout << "A*A^{-1} = " << endl;
    print_matrix(check_matrix_identity, N, n_rhs);
}


void CheckDuration(const size_t &N)
{
    for (size_t n_rhs : {1, 2, 4, 8, 16, 32})
    {
        auto time_start = system_clock::now();

        vector<double> A(N*N);
        vector<double> b(N*n_rhs);

        // initialize A
        for (size_t i = 0; i < N; ++i)
        {
            for (size_t j = 0; j < N; ++j)
                if (i == j)
                    A[N*i + j] = 4.0;
                else
```

```
   69:                     A[N*i + j] = 1.0/((i - j)*(i - j));
   70:         }
   71:
   72:         // initialize b
   73:         for (size_t j = 0; j < N; ++j)
   74:             for (size_t l = 0; l < n_rhs; ++l)
   75:                 b[N*l + j] = 1.0;
   76:
   77:
   78:         // solve system
   79:         factorization_solve(A, b, n_rhs);
   80:
   81:
   82:
   83:         auto time_end = system_clock::now();
   84:         auto duration = duration_cast<microseconds>(time_end - time_start);        /
/ duration in microseconds
   85:         double t_diff = static_cast<double>(duration.count()) / 1e6;               /
/ overall duration in seconds
   86:
   87:         cout << "n_rhs = " << n_rhs << "\tTime per rhs: " << t_diff/n_rhs << endl;
   88:     }
   89: }
```

```
1: #pragma once
2: #include <cstddef>
3:
4:
5: void CheckCorrectness();
6:
7: void CheckDuration(const size_t &N);
```

```
    1: // see:   http://llvm.org/docs/CodingStandards.html#include-style
    2: #include "geom.h"
    3:
    4: #include <algorithm>
    5: #include <cassert>
    6: #include <fstream>
    7: #include <iostream>
    8: #include <list>
    9: #include <string>
   10: #include <vector>
   11: using namespace std;
   12:
   13: Mesh::Mesh(int ndim, int nvert_e, int ndof_e)
   14:     : _nelem(0), _nvert_e(nvert_e), _ndof_e(ndof_e), _nnode(0), _ndim(ndim), _ia(0),
 _xc(0)
   15: {
   16: }
   17:
   18: Mesh::~Mesh()
   19: {}
   20:
   21: void Mesh::SetValues(std::vector<double> &v, const std::function<double(double, doub
le)> &func) const
   22: {
   23:     int const nnode = Nnodes();            // number of vertices in mesh
   24:     assert( nnode == static_cast<int>(v.size()) );
   25:     for (int k = 0; k < nnode; ++k)
   26:     {
   27:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
   28:     }
   29: }
   30:
   31:
   32: void Mesh::Debug() const
   33: {
   34:     cout << "\n ################ Debug  M E S H  ####################\n";
   35:     cout << "\n ..............    Coordinates       ..................\n";
   36:     for (int k = 0; k < _nnode; ++k)
   37:     {
   38:         cout << k << " : " ;
   39:         for (int i = 0; i < _ndof_e; ++i )
   40:         {
   41:             cout << _xc[2*k+i] << "  ";
   42:         }
   43:         cout << endl;
   44:     }
   45:     cout << "\n ..............    Elements        ..................\n";
   46:     for (int k = 0; k < _nelem; ++k)
   47:     {
   48:         cout << k << " : ";
   49:         for (int i = 0; i < _ndof_e; ++i )
   50:             cout << _ia[_ndof_e * k + i] << "  ";
   51:         cout << endl;
   52:     }
   53:     return;
   54: }
   55:
   56: void Mesh::Write_ascii_matlab(std::string const &fname, std::vector<double> const &v
) const
   57: {
   58:     assert(Nnodes() ==  static_cast<int>(v.size()));  // fits vector length to mesh
information?
   59:
   60:     ofstream fout(fname);                             // open file ASCII mode
   61:     if ( !fout.is_open() )
   62:     {
   63:         cout << "\nFile " << fname << " has not been opened.\n\n" ;
   64:         assert( fout.is_open() && "File not opened."  );
```

```
   65:     }
   66:
   67:     string const DELIMETER(" ");     // define the same delimeter as in matlab/ascii_
read*.m
   68:     int const    OFFSET(1);          // convert C-indexing to matlab
   69:
   70:     // Write data: #nodes, #space dimensions, #elements, #vertices per element
   71:     fout << Nnodes() << DELIMETER << Ndims() << DELIMETER << Nelems() << DELIMETER <
< NverticesElements() << endl;
   72:
   73:     // Write cordinates: x_k, y_k   in seperate lines
   74:     assert( Nnodes()*Ndims() ==  static_cast<int>(_xc.size()));
   75:     for (int k = 0, kj = 0; k < Nnodes(); ++k)
   76:     {
   77:         for (int j = 0; j < Ndims(); ++j, ++kj)
   78:         {
   79:             fout << _xc[kj] << DELIMETER;
   80:         }
   81:         fout << endl;
   82:     }
   83:
   84:     // Write connectivity: ia_k,0, ia_k,1 etc  in seperate lines
   85:     assert( Nelems()*NverticesElements() ==  static_cast<int>(_ia.size()));
   86:     for (int k = 0, kj = 0; k < Nelems(); ++k)
   87:     {
   88:         for (int j = 0; j < NverticesElements(); ++j, ++kj)
   89:         {
   90:             fout << _ia[kj] + OFFSET << DELIMETER;     // C to matlab
   91:         }
   92:         fout << endl;
   93:     }
   94:
   95:     // Write vector
   96:     for (int k = 0; k < Nnodes(); ++k)
   97:     {
   98:         fout << v[k] << endl;
   99:     }
  100:
  101:     fout.close();
  102:     return;
  103: }
  104:
  105:
  106: void Mesh::Visualize(std::vector<double> const &v) const
  107: {
  108:     // define external command
  109:     const string exec_m("matlab -nosplash < visualize_results.m");                 /
/ Matlab
  110:     //const string exec_m("octave --no-window-system --no-gui visualize_results.m");
 // Octave
  111:     //const string exec_m("flatpak run org.octave.Octave visualize_results.m");
 // Octave (flatpak): desktop GH
  112:
  113:     const string fname("uv.txt");
  114:     Write_ascii_matlab(fname, v);
  115:
  116:     int ierror = system(exec_m.c_str());                                  // call ext
ernal command
  117:
  118:     if (ierror != 0)
  119:     {
  120:         cout << endl << "Check path to Matlab/octave on your system" << endl;
  121:     }
  122:     cout << endl;
  123:     return;
  124: }
  125:
  126:
```

```
  127:
  128:
  129: // ######################################################################
  130: Mesh_2d_3_square::Mesh_2d_3_square(int nx, int ny, int myid, int procx, int procy)
  131:     : Mesh(2, 3, 3), // two dimensions, 3 vertices, 3 dofs
  132:       _myid(myid), _procx(procx), _procy(procy), _neigh{{-1, -1, -1, -1}}, _color(0)
,
  133: _xl(0.0), _xr(1.0), _yb(0.0), _yt(1.0), _nx(nx), _ny(ny)
  134: {
  135:     //void IniGeom(int const myid, int const procx, int const procy, int neigh[], in
t &color)
  136:     int const ky = _myid / _procx;
  137:     int const kx = _myid % _procy;  //    MOD(myid,procx)
  138:     // Determine the neighbors of domain/rank myid
  139:     _neigh[0] = (ky == 0)       ?  -1 : _myid - _procx;    //   South
  140:     _neigh[1] = (kx == _procx - 1) ?  -1 : _myid + 1;      //   East
  141:     _neigh[2] = (ky == _procy - 1) ?  -1 : _myid + _procx;  //   North
  142:     _neigh[3] = (kx == 0)       ?  -1 : _myid - 1;        //   West
  143:
  144:     _color = (kx + ky) & 1 ;
  145:
  146:     // subdomain is part of unit square
  147:     double const hx = 1. / _procx;
  148:     double const hy = 1. / _procy;
  149:     _xl = kx * hx;                      //  left
  150:     _xr = (kx + 1) * hx;                //  right
  151:     _yb = ky * hy;                      //  bottom
  152:     _yt = (ky + 1) * hy;                //  top
  153:
  154:     // Calculate coordinates
  155:     int const nnode = (_nx + 1) * (_ny + 1); // number of nodes
  156:     Resize_Coords(nnode, 2);                 // coordinates in 2D [nnode][ndim]
  157:     GetCoordsInRectangle(_nx, _ny, _xl, _xr, _yb, _yt, GetCoords().data());
  158:
  159:     // Calculate element connectivity (linear triangles)
  160:     int const nelem = 2 * _nx * _ny;         // number of elements
  161:     Resize_Connectivity(nelem, 3);           // connectivity matrix [nelem][3]
  162:     GetConnectivityInRectangle(_nx, _ny, GetConnectivity().data());
  163:
  164:     return;
  165: }
  166:
  167:
  168: void Mesh_2d_3_square::SetU(std::vector<double> &u) const
  169: {
  170:     int dx    = _nx + 1;
  171:     for (int j = 0; j <= _ny; ++j)
  172:     {
  173:         int k = j * dx;
  174:         for (int i = 0; i <= _nx; ++i, ++k)
  175:         {
  176:             u[k] = 0.0;
  177:         }
  178:     }
  179:
  180: }
  181:
  182: void Mesh_2d_3_square::SetF(std::vector<double> &f) const
  183: {
  184:     int dx    = _nx + 1;
  185:     for (int j = 0; j <= _ny; ++j)
  186:     {
  187:         int k = j * dx;
  188:         for (int i = 0; i <= _nx; ++i, ++k)
  189:         {
  190:             f[k] = 1.0;
  191:         }
  192:     }
```

```
  193:
  194: }
  195:
  196:
  197: std::vector<int> Mesh_2d_3_square::Index_DirichletNodes() const
  198: {
  199:     int const dx = 1,
  200:               dy = _nx + 1,
  201:               bl  = 0,
  202:               br  = _nx,
  203:               tl  = _ny * (_nx + 1),
  204:               tr  = (_ny + 1) * (_nx + 1) - 1;
  205:     int const start[4] = { bl, br, tl, bl},
  206:                           end[4] = { br, tr, tr, tl},
  207:                                    step[4] = { dx, dy, dx, dy};
  208:
  209:     vector<int> idx(0);
  210:     for (int j = 0; j < 4; j++)
  211:     {
  212:         if (_neigh[j] < 0)
  213:         {
  214:             for (int i = start[j]; i <= end[j]; i += step[j])
  215:             {
  216:                 idx.push_back(i);        // node i is Dirichlet node
  217:             }
  218:         }
  219:     }
  220:     //    remove multiple elements
  221:     sort(idx.begin(), idx.end());                          // sort
  222:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
  223:
  224:     return idx;
  225: }
  226:
  227: void Mesh_2d_3_square::SaveVectorP(std::string const &name, vector<double> const &u)
 const
  228: {
  229: //  construct the file name for subdomain myid
  230:     const string tmp( std::to_string(_myid / 100) + to_string((_myid % 100) / 10) +
to_string(_myid % 10) );
  231:
  232:     const string namep = name + "." + tmp;
  233:     ofstream ff(namep.c_str());
  234:     ff.precision(6);
  235:     ff.setf(ios::fixed, ios::floatfield);
  236:
  237:     // assumes tensor product grid in unit square; rowise numbered (as generated in
class constructor)
  238:     // output is provided for tensor product grid visualization ( similar to Matlab-
surf() )
  239:     auto const &xc = GetCoords();
  240:     int k = 0;
  241:     for (int j = 0; j <= _ny; ++j)
  242:     {
  243:         for (int i = 0; i <= _nx; ++i, ++k)
  244:             ff << xc[2 * k + 0] << "   " << xc[2 * k + 1] << "   " << u[k] << endl;
  245:         ff << endl;
  246:     }
  247:
  248:     ff.close();
  249:     return;
  250: }
  251:
  252: void Mesh_2d_3_square::GetCoordsInRectangle(int const nx, int const ny,
  253:         double const xl, double const xr, double const yb, double const yt,
  254:         double xc[])
  255: {
  256:     const double hx = (xr - xl) / nx,
```

```
  257:                     hy = (yt - yb) / ny;
  258:
  259:     int k = 0;
  260:     for (int j = 0; j <= ny; ++j)
  261:     {
  262:         const double y0 = yb + j * hy;
  263:         for (int i = 0; i <= nx; ++i, k += 2)
  264:         {
  265:             xc[k  ] = xl + i * hx;
  266:             xc[k + 1] = y0;
  267:         }
  268:     }
  269:
  270:     return;
  271: }
  272:
  273: void Mesh_2d_3_square::GetConnectivityInRectangle(int const nx, int const ny, int ia
[])
  274: {
  275:     const int dx = nx + 1;
  276:     int k  = 0;
  277:     int l  = 0;
  278:     for (int j = 0; j < ny; ++j, ++k)
  279:     {
  280:         for (int i = 0; i < nx; ++i, ++k)
  281:         {
  282:             ia[l  ] = k;
  283:             ia[l + 1] = k + 1;
  284:             ia[l + 2] = k + dx + 1;
  285:             l += 3;
  286:             ia[l  ] = k;
  287:             ia[l + 1] = k + dx;
  288:             ia[l + 2] = k + dx + 1;
  289:             l += 3;
  290:         }
  291:     }
  292:     return;
  293: }
  294:
  295: // ################### still some old code (--> MPI) ############################
  296:
  297:
  298: //  Copies the values of w corresponding to the boundary
  299: //  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  300:
  301: void GetBound(int const ib, int const nx, int const ny, double const w[], double s[]
)
  302: {
  303:     const int //dx = 1,
  304:     dy = nx + 1,
  305:     bl  = 0,
  306:     br  = nx,
  307:     tl  = ny * (nx + 1),
  308:     tr  = (ny + 1) * (nx + 1) - 1;
  309:     switch (ib)
  310:     {
  311:         case 1:
  312:         {
  313:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  314:                 s[j] = w[i];
  315:             break;
  316:         }
  317:         case 3:
  318:         {
  319:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  320:                 s[j] = w[i];
  321:             break;
  322:         }
```

```
  323:         case 4:
  324:         {
  325:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  326:                 s[j] = w[i];
  327:             break;
  328:         }
  329:         case 2:
  330:         {
  331:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  332:                 s[j] = w[i];
  333:             break;
  334:         }
  335:         default:
  336:         {
  337:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  338:         }
  339:     }
  340:     return;
  341: }
  342:
  343: // ------------------------------------------------------------------------------
-------------------------
  344: // Computes w:  = w + s at nodes on  the boundary
  345: // South (ib == 1), East (ib == 2), North (ib == 3), West (ib == 4)
  346:
  347: void AddBound(int const ib, int const nx, int const ny, double w[], double const s[]
)
  348: {
  349:     int const dy = nx + 1,
  350:               bl  = 0,
  351:               br  = nx,
  352:               tl  = ny * (nx + 1),
  353:               tr  = (ny + 1) * (nx + 1) - 1;
  354:     switch (ib)
  355:     {
  356:         case 1:
  357:         {
  358:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  359:                 w[i] += s[j];
  360:             break;
  361:         }
  362:         case 3:
  363:         {
  364:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  365:                 w[i] += s[j];
  366:             break;
  367:         }
  368:         case 4:
  369:         {
  370:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  371:                 w[i] += s[j];
  372:             break;
  373:         }
  374:         case 2:
  375:         {
  376:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  377:                 w[i] += s[j];
  378:             break;
  379:         }
  380:         default:
  381:         {
  382:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  383:         }
  384:     }
  385:     return;
  386: }
```

```
387:
388:
389: // ########################################################################
390:
391: Mesh_2d_3_matlab::Mesh_2d_3_matlab(string const &fname)
392:     : Mesh(2, 3, 3),  // two dimensions, 3 vertices, 3 dofs
393:       bedges(0)
394: {
395:     ifstream ifs(fname);
396:     if (!(ifs.is_open() && ifs.good()))
397:     {
398:         cerr << "Mesh_2d_3_matlab: Error cannot open file " << fname << endl;
399:         assert(ifs.is_open());
400:     }
401:
402:     int const OFFSET(1);              // Matlab to C indexing
403:     cout << "ASCI file  " << fname << "  opened" << endl;
404:
405:     // Read some mesh constants
406:     int nnode, ndim, nelem, nvert_e;
407:     ifs >> nnode >> ndim >> nelem >> nvert_e;
408:     cout << nnode << "  " << ndim << "  " << nelem << "  " << nvert_e << endl;
409:     assert(ndim == 2 && nvert_e == 3);
410:
411:     // Allocate memory
412:     Resize_Coords(nnode, ndim);                  // coordinates in 2D [nnode][ndim]
413:     Resize_Connectivity(nelem, nvert_e);         // connectivity matrix [nelem][nvert
]
414:
415:     // Read ccordinates
416:     auto &xc = GetCoords();
417:     for (int k = 0; k < nnode * ndim; ++k)
418:     {
419:         ifs >> xc[k];
420:     }
421:
422:     // Read connectivity
423:     auto &ia = GetConnectivity();
424:     for (int k = 0; k < nelem * nvert_e; ++k)
425:     {
426:         ifs >> ia[k];
427:         ia[k] -= OFFSET;                 // Matlab to C indexing
428:     }
429:
430:     // additional read of boundary information (only start/end point)
431:     int nbedges;
432:     ifs >> nbedges;
433:
434:     bedges.resize(nbedges * 2);
435:     for (int k = 0; k < nbedges * 2; ++k)
436:     {
437:         ifs >> bedges[k];
438:         bedges[k] -= OFFSET;             // Matlab to C indexing
439:     }
440:
441:     return;
442: }
443:
444: // binary
445: //{
446: //ifstream ifs(fname, ios_base::in | ios_base::binary);
447: //if(!(ifs.is_open() && ifs.good()))
448: //{
449: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
450: //assert(ifs.is_open());
451: //}
452: //cout << "ReadBinMatrix: file opened" << file << endl;
453:
```

```
454:
455: //}
456:
457: // binaryIO.cpp
458: //void read_binMatrix(const string& file, vector<int> &cnt, vector<int> &col, vector
<double> &ele)
459: //{
460:
461: //ifstream ifs(file, ios_base::in | ios_base::binary);
462:
463: //if(!(ifs.is_open() && ifs.good()))
464: //{
465: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
466: //assert(ifs.is_open());
467: //}
468: //cout << "ReadBinMatrix: Opened file " << file << endl;
469:
470: //int _size;
471:
472: //ifs.read(reinterpret_cast<char*>(&_size), sizeof(int));   // old: ifs.read((char*)
&_size, sizeof(int));
473: //cnt.resize(_size);
474: //cout << "ReadBinMatrix: cnt size: " << _size << endl;
475:
476: //ifs.read((char*)&_size, sizeof(int));
477: //col.resize(_size);
478: //cout << "ReadBinMatrix: col size: " << _size << endl;
479:
480: //ifs.read((char*)&_size, sizeof(int));
481: //ele.resize(_size);
482: //cout << "ReadBinMatrix: ele size: " << _size << endl;
483:
484:
485: //ifs.read((char*)cnt.data(), cnt.size() * sizeof(int));
486: //ifs.read((char*)col.data(), col.size() * sizeof(int));
487: //ifs.read((char*)ele.data(), ele.size() * sizeof(double));
488:
489: //ifs.close();
490: //cout << "ReadBinMatrix: Finished reading matrix.." << endl;
491:
492: //}
493:
494:
495: std::vector<int> Mesh_2d_3_matlab::Index_DirichletNodes() const
496: {
497:     vector<int> idx(bedges);                                  // copy
498:
499:     sort(idx.begin(), idx.end());                             // sort
500:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
501:
502:     return idx;
503: }
504:
505:
506:
507:
508:
509:
510:
511:
512:
513:
514:
515:
516:
517:
518:
519:
```

```
520:
521:
522:
```

```
    1: #ifndef GEOM_FILE
    2: #define GEOM_FILE
    3: #include <array>
    4: #include <functional>             // function; C++11
    5: #include <string>
    6: #include <vector>
    7:
    8: /**
    9:  * Basis class for finite element meshes.
   10:  */
   11: class Mesh
   12: {
   13: public:
   14:     /**
   15:      * Constructor initializing the members with default values.
   16:      *
   17:      * @param[in] ndim  space dimensions (dimension for coordinates)
   18:      * @param[in] nvert_e  number of vertices per element (dimension for connectivi
ty)
   19:      * @param[in] ndof_e   degrees of freedom per element (= @p nvert_e for linear
elements)
   20:      */
   21:     explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0);
   22:
   23:     /**
   24:      * Destructor.
   25:      *
   26:      * See clang warning on
   27:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-vi
rtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
   28:      */
   29:     virtual ~Mesh();
   30:
   31:     /**
   32:      * Number of finite elements in (sub)domain.
   33:      * @return number of elements.
   34:      */
   35:     int Nelems() const
   36:     {
   37:         return _nelem;
   38:     }
   39:
   40:     /**
   41:      * Global number of vertices for each finite element.
   42:      * @return number of vertices per element.
   43:      */
   44:     int NverticesElements() const
   45:     {
   46:         return _nvert_e;
   47:     }
   48:
   49:     /**
   50:      * Global number of degrees of freedom (dof) for each finite element.
   51:      * @return degrees of freedom per element.
   52:      */
   53:     int NdofsElement() const
   54:     {
   55:         return _ndof_e;
   56:     }
   57:
   58:     /**
   59:      * Number of vertices in mesh.
   60:      * @return number of vertices.
   61:      */
   62:     int Nnodes() const
   63:     {
   64:         return _nnode;
   65:     }
```

```
   66:
   67:     /**
   68:      * Space dimension.
   69:      * @return number of dimensions.
   70:      */
   71:     int Ndims() const
   72:     {
   73:         return _ndim;
   74:     }
   75:
   76:     /**
   77:      * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
   78:       *
   79:      * @param[in] nelem    number of elements
   80:       * @param[in] nvert_e  number of vertices per element
   81:      */
   82:     void Resize_Connectivity(int nelem, int nvert_e)
   83:     {
   84:         SetNelem(nelem);               // number of elements
   85:         SetNverticesElement(nvert_e);  // vertices per element
   86:         _ia.resize(nelem * nvert_e);
   87:     }
   88:
   89:     /**
   90:      * Read connectivity information (g1,g2,g3)_i.
   91:      * @return convectivity vector [nelems*ndofs].
   92:      */
   93:     const std::vector<int>  &GetConnectivity() const
   94:     {
   95:         return _ia;
   96:     }
   97:
   98:     /**
   99:      * Access/Change connectivity information (g1,g2,g3)_i.
  100:      * @return convectivity vector [nelems*ndofs].
  101:      */
  102:     std::vector<int>  &GetConnectivity()
  103:     {
  104:         return _ia;
  105:     }
  106:
  107:     /**
  108:      * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
  109:       *
  110:      * @param[in] nnodes    number of nodes
  111:       * @param[in] ndim      space dimension
  112:      */
  113:     void Resize_Coords(int nnodes, int ndim)
  114:     {
  115:         SetNnode(nnodes);       // number of nodes
  116:         SetNdim(ndim);          // space dimension
  117:         _xc.resize(nnodes * ndim);
  118:     }
  119:
  120:     /**
  121:      * Read coordinates of vertices (x,y)_i.
  122:      * @return coordinates vector [nnodes*2].
  123:      */
  124:     const std::vector<double> &GetCoords() const
  125:     {
  126:         return _xc;
  127:     }
  128:
  129:     /**
  130:      * Access/Change coordinates of vertices (x,y)_i.
  131:      * @return coordinates vector [nnodes*2].
```

```
  132:      */
  133:     std::vector<double> &GetCoords()
  134:     {
  135:         return _xc;
  136:     }
  137:
  138:     /**
  139:      * Calculate values in vector @p v via function @p func(x,y)
  140:      * @param[in] v     vector
  141:       * @param[in] func  function of (x,y) returning a double value.
  142:      */
  143:     void SetValues(std::vector<double> &v, const std::function<double(double, double
)> &func) const;
  144:
  145:     /**
  146:      * Prints the information for a finite element mesh
  147:      */
  148:     void Debug() const;
  149:
  150:     /**
  151:      * Determines the indices of those vertices with Dirichlet boundary conditions
  152:      * @return index vector.
  153:      */
  154:     virtual std::vector<int> Index_DirichletNodes() const = 0;
  155:
  156:     /**
  157:      * Write vector @p v toghether with its mesh information to an ASCii file @p fna
me.
  158:       *
  159:       * The data are written in C-style.
  160:       *
  161:       * @param[in] fname  file name
  162:       * @param[in] v      vector
  163:      */
  164:     void Write_ascii_matlab(std::string const &fname, std::vector<double> const &v)
const;
  165:
  166:     /**
  167:      * Visualize @p v together with its mesh information via matlab or octave.
  168:       *
  169:       * Comment/uncomment those code lines in method Mesh:Visualize (geom.cpp)
  170:       * that are supported on your system.
  171:       *
  172:       * @param[in] v    vector
  173:       *
  174:       * @warning matlab files ascii_read_meshvector.m  visualize_results.m
  175:       *          must be in the executing directory.
  176:      */
  177:     void Visualize(std::vector<double> const &v) const;
  178:
  179:
  180: protected:
  181:     void SetNelem(int nelem)
  182:     {
  183:         _nelem = nelem;
  184:     }
  185:
  186:     void SetNverticesElement(int nvert)
  187:     {
  188:         _nvert_e = nvert;
  189:     }
  190:
  191:     void SetNdofsElement(int ndof)
  192:     {
  193:         _ndof_e = ndof;
  194:     }
  195:
  196:     void SetNnode(int nnode)
```

```
  197:     {
  198:         _nnode = nnode;
  199:     }
  200:
  201:     void SetNdim(int ndim)
  202:     {
  203:         _ndim = ndim;
  204:     }
  205:
  206: private:
  207:     int _nelem;         //!< number elements
  208:     int _nvert_e;       //!< number of vertices per element
  209:     int _ndof_e;        //!< degrees of freedom (d.o.f.) per element
  210:     int _nnode;         //!< number nodes/vertices
  211:     int _ndim;          //!< space dimension of the problem (1, 2, or 3)
  212:     std::vector<int> _ia;    //!< element connectivity
  213:     std::vector<double> _xc; //!< coordinates
  214: };
  215:
  216: /**
  217:  * 2D finite element mesh of the square consiting of linear triangular elements.
  218:  */
  219: class Mesh_2d_3_square: public Mesh
  220: {
  221: public:
  222:     /**
  223:      * Generates the f.e. mesh for the unit square.
  224:      *
  225:      * @param[in] nx    number of discretization intervals in x-direction
  226:      * @param[in] ny    number of discretization intervals in y-direction
  227:      * @param[in] myid  my MPI-rank / subdomain
  228:      * @param[in] procx number of ranks/subdomains in x-direction
  229:      * @param[in] procy number of processes in y-direction
  230:      */
  231:     Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);
  232:
  233:     /**
  234:      * Destructor
  235:      */
  236:     ~Mesh_2d_3_square() override
  237:     {}
  238:
  239:     /**
  240:      * Set solution vector based on a tensor product grid in the rectangle.
  241:      * @param[in] u solution vector
  242:      */
  243:     void SetU(std::vector<double> &u) const;
  244:
  245:     /**
  246:      * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
  247:      * @param[in] f rhs vector
  248:      */
  249:     void SetF(std::vector<double> &f) const;
  250:
  251:     /**
  252:      * Determines the indices of those vertices with Dirichlet boundary conditions
  253:      * @return index vector.
  254:      */
  255:     std::vector<int> Index_DirichletNodes() const override;
  256:
  257:     /**
  258:       * Stores the values of vector @p u of (sub)domain into a file @p name for furt
her processing in gnuplot.
  259:       * The file stores rowise the x- and y- coordinates together with the value fro
m  @p u .
  260:       * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny
 intervals.
  261:       *
```

```
  262:       * @param[in] name basename of file name (file name will be extended by the ran
k number)
  263:       * @param[in] u    local vector
  264:       *
  265:       * @warning   Assumes tensor product grid in unit square; rowise numbered
  266:       *            (as generated in class constructor).
  267:       *            The output is provided for tensor product grid visualization
  268:       *            ( similar to Matlab-surf() ).
  269:       *
  270:       * @see Mesh_2d_3_square
  271:       */
  272:      void SaveVectorP(std::string const &name, std::vector<double> const &u) const;
  273:
  274:      // here will still need to implement in the class
  275:      //  GetBound(), AddBound()
  276:      //  or better a generalized way with indices and their appropriate ranks for MPI
 communication
  277:
  278: private:
  279:      /**
  280:       * Determines the coordinates of the dicretization nodes of the domain [@p xl,
@p xr] x [@p yb, @p yt]
  281:       * which is discretized into @p nx x @p ny intervals.
  282:       *
  283:       * @param[in] ny   number of discretization intervals in y-direction
  284:       * @param[in] xl   x-coordinate of left boundary
  285:       * @param[in] xr   x-coordinate of right boundary
  286:       * @param[in] yb   y-coordinate of lower boundary
  287:       * @param[in] yt   y-coordinate of upper boundary
  288:       * @param[out] xc  coordinate vector of length 2n with x(2*k,2*k+1) as coodinat
es of node k
  289:       */
  290:
  291:      void GetCoordsInRectangle(int nx, int ny, double xl, double xr, double yb, doubl
e yt,
  292:                                double xc[]);
  293:      /**
  294:       * Determines the element connectivity of linear triangular elements of a FEM d
iscretization
  295:       * of a rectangle using @p nx x @p ny equidistant intervals for discretization.
  296:       * @param[in] nx   number of discretization intervals in x-direction
  297:       * @param[in] ny   number of discretization intervals in y-direction
  298:       * @param[out] ia  element connectivity matrix with ia(3*s,3*s+1,3*s+2) as node
 numbers od element s
  299:       */
  300:      void GetConnectivityInRectangle(int nx, int ny, int ia[]);
  301:
  302: private:
  303:      int _myid;           //!< my MPI rank
  304:      int _procx;          //!< number of MPI ranks in x-direction
  305:      int _procy;          //!< number of MPI ranks in y-direction
  306:      std::array<int, 4> _neigh; //!< MPI ranks of neighbors (negative: no neighbor bu
t b.c.)
  307:      int _color;          //!< red/black coloring (checker board) of subdomains
  308:
  309:      double _xl;          //!< x coordinate of lower left  corner of square
  310:      double _xr;          //!< x coordinate of lower right corner of square
  311:      double _yb;          //!< y coordinate or lower left  corner of square
  312:      double _yt;          //!< y coordinate of upper right corner of square
  313:      int    _nx;          //!< number of intervals in x-direction
  314:      int    _ny;          //!< number of intervals in y-direction
  315: };
  316:
  317: // ################### still some old code (--> MPI) ############################
  318: /**
  319:  * Copies the values of @p w corresponding to boundary @p ib
  320:  * onto vector s.  South (ib==1), East (ib==2), North (ib==3), West (ib==4).
  321:  * The vector @p s has to be long enough!!
```

```
  322:  * @param[in] ib    my local boundary
  323:  * @param[in] nx    number of discretization intervals in x-direction
  324:  * @param[in] ny    number of discretization intervals in y-direction
  325:  * @param[in] w     vector for all nodes of local discretization
  326:  * @param[out] s    short vector with values on boundary @p ib
  327: */
  328: // GH_NOTE: Absicherung bei s !!
  329: void GetBound(int ib, int nx, int ny, double const w[], double s[]);
  330:
  331:
  332: /**
  333:  * Computes @p w := @p w + @p s  at the interface/boundary nodes on the
  334:  * boundary @p ib .  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  335:  * @param[in] ib    my local boundary
  336:  * @param[in] nx    number of discretization intervals in x-direction
  337:  * @param[in] ny    number of discretization intervals in y-direction
  338:  * @param[in,out] w vector for all nodes of local discretization
  339:  * @param[in] s     short vector with values on boundary @p ib
  340: */
  341: void AddBound(int ib, int nx, int ny, double w[], double const s[]);
  342:
  343: // ################### Mesh from Matlab ############################
  344: /**
  345:  * 2D finite element mesh of the square consiting of linear triangular elements.
  346:  */
  347: class Mesh_2d_3_matlab: public Mesh
  348: {
  349: public:
  350:     /**
  351:      * Reads mesh data from a binary file.
  352:      *
  353:      * File format, see ascii_write_mesh.m
  354:      *
  355:      * @param[in] fname file name
  356:     */
  357:     explicit Mesh_2d_3_matlab(std::string const &fname);
  358:
  359:     /**
  360:      * Determines the indices of those vertices with Dirichlet boundary conditions.
  361:      * @return index vector.
  362:       *
  363:       * @warning All boundary nodes are considered as Dirchlet nodes.
  364:      */
  365:     std::vector<int> Index_DirichletNodes() const override;
  366:
  367: private:
  368:     /**
  369:      * Determines the indices of those vertices with Dirichlet boundary conditions
  370:      * @return index vector.
  371:      */
  372:     int Nnbedges() const
  373:     {
  374:         return static_cast<int>(bedges.size());
  375:     }
  376:
  377:     std::vector<int> bedges;     //!< boundary edges [nbedges][2] storing start/end
vertex
  378:
  379: };
  380:
  381: #endif
```

```
    1: #include "getmatrix.h"
    2: #include "userset.h"
    3:
    4: #include <algorithm>
    5: #include <cassert>
    6: #include <cmath>
    7: #include <iomanip>
    8: #include <iostream>
    9: #include <list>
   10: #include <vector>
   11: using namespace std;
   12:
   13:
   14: //  general routine for lin. triangular elements
   15:
   16: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3])
   17: //void CalcElem(const int* __restrict__ ial, const double* __restrict__ xc, double*
__restrict__ ske[3], double* __restrict__ fe)
   18: {
   19:     const int  i1  = 2 * ial[0],   i2 = 2 * ial[1],   i3 = 2 * ial[2];
   20:     const double x13 = xc[i3 + 0] - xc[i1 + 0],  y13 = xc[i3 + 1] - xc[i1 + 1],
   21:                  x21 = xc[i1 + 0] - xc[i2 + 0],  y21 = xc[i1 + 1] - xc[i2 + 1],
   22:                  x32 = xc[i2 + 0] - xc[i3 + 0],  y32 = xc[i2 + 1] - xc[i3 + 1];
   23:     const double jac = fabs(x21 * y13 - x13 * y21);
   24:
   25:     ske[0][0] = 0.5 / jac * (y32 * y32 + x32 * x32);
   26:     ske[0][1] = 0.5 / jac * (y13 * y32 + x13 * x32);
   27:     ske[0][2] = 0.5 / jac * (y21 * y32 + x21 * x32);
   28:     ske[1][0] = ske[0][1];
   29:     ske[1][1] = 0.5 / jac * (y13 * y13 + x13 * x13);
   30:     ske[1][2] = 0.5 / jac * (y21 * y13 + x21 * x13);
   31:     ske[2][0] = ske[0][2];
   32:     ske[2][1] = ske[1][2];
   33:     ske[2][2] = 0.5 / jac * (y21 * y21 + x21 * x21);
   34:
   35:     const double xm    = (xc[i1 + 0] + xc[i2 + 0] + xc[i3 + 0]) / 3.0,
   36:                  ym    = (xc[i1 + 1] + xc[i2 + 1] + xc[i3 + 1]) / 3.0;
   37:     //fe[0] = fe[1] = fe[2] = 0.5 * jac * FunctF(xm, ym) / 3.0;
   38:     fe[0] = fe[1] = fe[2] = 0.5 * jac * fNice(xm, ym) / 3.0;
   39: }
   40:
   41:
   42: // general routine for lin. triangular elements,
   43: // non-symm. matrix
   44: // node numbering in element:  a s c e n d i n g   indices !!
   45: // GH: deprecated
   46: void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
   47:              int const id[], int const ik[], double sk[], double f[])
   48: {
   49:     for (int i = 0; i < 3; ++i)
   50:     {
   51:         const int ii  = ial[i],           // row ii (global index)
   52:                   id1 = id[ii],           // start and
   53:                   id2 = id[ii + 1];       // end of row ii in matrix
   54:         int ip  = id1;
   55:         for (int j = 0; j < 3; ++j)       // no symmetry assumed
   56:         {
   57:             const int jj = ial[j];
   58:             bool not_found = true;
   59:             do       // find entry jj (global index) in row ii
   60:             {
   61:                 not_found = (ik[ip] != jj);
   62:                 ++ip;
   63:             }
   64:             while (not_found && ip < id2);
   65:
   66: #ifndef NDEBUG                 // compiler option -DNDEBUG switches off the check
   67:             if (not_found)      // no entry found !!
```

```
   68:             {
   69:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
   70:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
   71:                 assert(!not_found);
   72:             }
   73: #endif
   74:             sk[ip - 1] += ske[i][j];
   75:         }
   76:         f[ii] += fe[i];
   77:     }
   78: }
   79:
   80:
   81: // ----------------------------------------------------------------------------
   82:
   83:
   84:
   85:
   86: // ############################################################################
   87:
   88: CRS_Matrix::CRS_Matrix(Mesh const &mesh)
   89:     : _mesh(mesh), _nrows(0), _nnz(0), _id(0), _ik(0), _sk(0)
   90: {
   91:     Derive_Matrix_Pattern();
   92:     return;
   93: }
   94:
   95: void CRS_Matrix::Derive_Matrix_Pattern()
   96: {
   97:     int const nelem(_mesh.Nelems());
   98:     int const ndof_e(_mesh.NdofsElement());
   99:     auto const &ia(_mesh.GetConnectivity());
  100: //  Determine the number of matrix rows
  101:     _nrows = *max_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem);
  102:     ++_nrows;                                   // node numberng: 0 ... nnode-1
  103:     assert(*min_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem) == 0); // numberi
ng starts with 0 ?
  104:
  105: //  Collect for each node those nodes it is connected to (multiple entries)
  106: //  Detect the neighboring nodes
  107:     vector< list<int> > cc(_nrows);             //  cc[i] is the  list of nodes a no
de i is connected to
  108:     for (int i = 0; i < nelem; ++i)
  109:     {
  110:         int const idx = ndof_e * i;
  111:         for (int k = 0; k < ndof_e; ++k)
  112:         {
  113:             list<int> &cck = cc.at(ia[idx + k]);
  114:             cck.insert( cck.end(), ia.cbegin() + idx, ia.cbegin() + idx + ndof_e );
  115:         }
  116:     }
  117: //  Delete the multiple entries
  118:     _nnz = 0;
  119:     for (auto &it : cc)
  120:     {
  121:         it.sort();
  122:         it.unique();
  123:         _nnz += static_cast<int>(it.size());
  124:         // cout << it.size() << " :: "; copy(it->begin(),it->end(), ostream_iterator
<int,char>(cout,"  ")); cout << endl;
  125:     }
  126:
  127: // CSR data allocation
  128:     _id.resize(_nrows + 1);                   // Allocate memory for CSR row pointer
  129:     _ik.resize(_nnz);                         // Allocate memory for CSR column index
 vector
  130:
  131: //  copy CSR data
```

```
  132:     _id[0] = 0;                                  // begin of first row
  133:     for (size_t i = 0; i < cc.size(); ++i)
  134:     {
  135:         //cout << i << "   " << nid.at(i) << endl;;
  136:         const list<int> &ci = cc.at(i);
  137:         const auto nci = static_cast<int>(ci.size());
  138:         _id[i + 1] = _id[i] + nci; // begin of next line
  139:         copy(ci.begin(), ci.end(), _ik.begin() + _id[i] );
  140:     }
  141:
  142:     assert(_nnz == _id[_nrows]);
  143:     _sk.resize(_nnz);                       // Allocate memory for CSR column index v
ector
  144:     return;
  145: }
  146:
  147:
  148: void CRS_Matrix::Debug() const
  149: {
  150: //  ID points to first entry of row
  151: //  no symmetry assumed
  152:     cout << "\nMatrix  (nnz = " << _id[_nrows] << ")\n";
  153:
  154:     for (int row = 0; row < _nrows; ++row)
  155:     {
  156:         cout << "Row " << row << " : ";
  157:         int const id1 = _id[row];
  158:         int const id2 = _id[row + 1];
  159:         for (int j = id1; j < id2; ++j)
  160:         {
  161:             cout.setf(ios::right, ios::adjustfield);
  162:             cout << "[" << setw(2) << _ik[j] << "]  " << setw(4) << _sk[j] << "  ";
  163:         }
  164:         cout << endl;
  165:     }
  166:     return;
  167: }
  168:
  169: void CRS_Matrix::CalculateLaplace(vector<double> &f)
  170: {
  171:     assert(_mesh.NdofsElement() == 3);               // only for triangular, linear
elements
  172:     //cout << _nnz << " vs. " << _id[_nrows] << "  " << _nrows<< endl;
  173:     assert(_nnz == _id[_nrows]);
  174:
  175:     for (int k = 0; k < _nrows; ++k)
  176:     {
  177:         _sk[k] = 0.0;
  178:     }
  179:     for (int k = 0; k < _nrows; ++k)
  180:     {
  181:         f[k] = 0.0;
  182:     }
  183:
  184:     double ske[3][3], fe[3];
  185:     //  Loop over all elements
  186:     auto const nelem = _mesh.Nelems();
  187:     auto const &ia   = _mesh.GetConnectivity();
  188:     auto const &xc   = _mesh.GetCoords();
  189:
  190:     for (int i = 0; i < nelem; ++i)
  191:     {
  192:         CalcElem(ia.data() + 3 * i, xc.data(), ske, fe);
  193:         //AddElem(ia.data()+3 * i, ske, fe, _id.data(), _ik.data(), _sk.data(), f.da
ta()); // GH: deprecated
  194:         AddElem_3(ia.data() + 3 * i, ske, fe, f);
  195:     }
  196:
```

```
  197:     //Debug();
  198:
  199:     return;
  200: }
  201:
  202: void CRS_Matrix::ApplyDirichletBC(std::vector<double> const &u, std::vector<double>
&f)
  203: {
  204:     double const PENALTY = 1e6;
  205:     auto const idx = _mesh.Index_DirichletNodes();
  206:     int const nidx = static_cast<int>(idx.size());
  207:
  208:     for (int row = 0; row < nidx; ++row)
  209:     {
  210:         int const k = idx[row];
  211:         int const id1 = fetch(k, k); // Find diagonal entry of row
  212:         assert(id1 >= 0);
  213:         _sk[id1] += PENALTY;            // matrix weighted scaling feasible
  214:         f[k] += PENALTY * u[k];
  215:     }
  216:
  217:     return;
  218: }
  219:
  220: void CRS_Matrix::GetDiag(vector<double> &d) const
  221: {
  222:     assert( _nrows == static_cast<int>(d.size()) );
  223:
  224:     for (int row = 0; row < _nrows; ++row)
  225:     {
  226:         const int ia = fetch(row, row); // Find diagonal entry of row
  227:         assert(ia >= 0);
  228:         d[row] = _sk[ia];
  229:     }
  230:     return;
  231: }
  232:
  233: bool CRS_Matrix::Compare2Old(int nnode, int const id[], int const ik[], double const
 sk[]) const
  234: {
  235:     bool bn = (nnode == _nrows);     // number of rows
  236:     if (!bn)
  237:     {
  238:         cout << "#########   Error: " << "number of rows" << endl;
  239:     }
  240:
  241:     bool bz = (id[nnode] == _nnz);   // number of non zero elements
  242:     if (!bz)
  243:     {
  244:         cout << "#########   Error: " << "number of non zero elements" << endl;
  245:     }
  246:
  247:     bool bd = equal(id, id + nnode + 1, _id.cbegin()); // row starts
  248:     if (!bd)
  249:     {
  250:         cout << "#########   Error: " << "row starts" << endl;
  251:     }
  252:
  253:     bool bk = equal(ik, ik + id[nnode], _ik.cbegin()); // column indices
  254:     if (!bk)
  255:     {
  256:         cout << "#########   Error: " << "column indices" << endl;
  257:     }
  258:
  259:     bool bv = equal(sk, sk + id[nnode], _sk.cbegin()); // values
  260:     if (!bv)
  261:     {
  262:         cout << "#########   Error: " << "values" << endl;
```

```
  263:     }
  264:
  265:     return bn && bz && bd && bk && bv;
  266: }
  267:
  268:
  269: void CRS_Matrix::Mult(vector<double> &w, vector<double> const &u) const
  270: {
  271:     assert( _nrows == static_cast<int>(w.size()) );
  272:     assert( w.size() == u.size() );
  273:
  274:     for (int row = 0; row < _nrows; ++row)
  275:     {
  276:         double wi = 0.0;
  277:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
  278:         {
  279:             wi += _sk[ij] * u[ _ik[ij] ];
  280:         }
  281:         w[row] = wi;
  282:     }
  283:     return;
  284: }
  285:
  286: void CRS_Matrix::Defect(vector<double> &w,
  287:                         vector<double> const &f, vector<double> const &u) const
  288: {
  289:     assert( _nrows == static_cast<int>(w.size()) );
  290:     assert( w.size() == u.size() && u.size() == f.size() );
  291:
  292:     for (int row = 0; row < _nrows; ++row)
  293:     {
  294:         double wi = f[row];
  295:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
  296:         {
  297:             wi -= _sk[ij] * u[ _ik[ij] ];
  298:         }
  299:         w[row] = wi;
  300:     }
  301:     return;
  302: }
  303:
  304: int CRS_Matrix::fetch(int const row, int const col) const
  305: {
  306:     int const id2 = _id[row + 1];    // end   and
  307:     int       ip  = _id[row];        // start of recent row (global index)
  308:
  309:     while (ip < id2 && _ik[ip] != col)  // find index col (global index)
  310:     {
  311:         ++ip;
  312:     }
  313:     if (ip >= id2)
  314:     {
  315:         ip = -1;
  316: #ifndef NDEBUG                 // compiler option -DNDEBUG switches off the check
  317:         cout << "No column  " << col << "  in row  " << row << endl;
  318:         assert(ip >= id2);
  319: #endif
  320:     }
  321:     return ip;
  322: }
  323:
  324:
  325: void CRS_Matrix::AddElem_3(int const ial[3], double const ske[3][3], double const fe
[3], vector<double> &f)
  326: {
  327:     for (int i = 0; i < 3; ++i)
  328:     {
  329:         const int ii  = ial[i];           // row ii (global index)
```

```
330:         for (int j = 0; j < 3; ++j)       // no symmetry assumed
331:         {
332:             const int jj = ial[j];        // column jj (global index)
333:             int ip = fetch(ii, jj);       // find column entry jj in row ii
334: #ifndef NDEBUG                            // compiler option -DNDEBUG switches off the check
335:             if (ip < 0)                   // no entry found !!
336:             {
337:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
338:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
339:                 assert(ip >= 0);
340:             }
341: #endif
342:             _sk[ip] += ske[i][j];
343:         }
344:         f[ii] += fe[i];
345:     }
346: }
347:
```

```
    1: #ifndef GETMATRIX_FILE
    2: #define GETMATRIX_FILE
    3:
    4: #include "geom.h"
    5: #include <vector>
    6:
    7: /**
    8:  * Calculates the element stiffness matrix @p ske and the element load vector @p fe
    9:  * of one triangular element with linear shape functions.
   10:  * @param[in]   ial     node indices of the three element vertices
   11:  * @param[in]   xc      vector of node coordinates with x(2*k,2*k+1) as coodinates o
f node k
   12:  * @param[out] ske      element stiffness matrix
   13:  * @param[out] fe       element load vector
   14:  */
   15: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3]);
   16:
   17: /**
   18:  * Adds the element stiffness matrix @p ske and the element load vector @p fe
   19:  * of one triangular element with linear shape functions to the appropriate position
s in
   20:  * the symmetric stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik)
   21:  *
   22:  * @param[in] ial   node indices of the three element vertices
   23:  * @param[in] ske       element stiffness matrix
   24:  * @param[in] fe        element load vector
   25:  * @param[out] sk       vector non-zero entries of CSR matrix
   26:  * @param[in] id        index vector containing the first entry in a CSR row
   27:  * @param[in] ik        column index vector of CSR matrix
   28:  * @param[out] f        distributed local vector storing the right hand side
   29:  *
   30:  * @warning Algorithm requires indices in connectivity @p ial in ascending order.
   31:  *          Currently deprecated.
   32: */
   33: void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
   34:              int const id[], int const ik[], double sk[], double f[]);
   35:
   36:
   37: // ########################################################################
   38: /**
   39:  * Square matrix in CRS format (compressed row storage; also named CSR),
   40:  * see an <a href="https://en.wikipedia.org/wiki/Sparse_matrix">introduction</a>.
   41:  */
   42: class CRS_Matrix
   43: {
   44:     public:
   45:        /**
   46:         * Intializes the CRS matrix structure from the given discetization in @p mes
h.
   47:         *
   48:         * The sparse matrix pattern is generated but the values are 0.
   49:         *
   50:         * @param[in] mesh given discretization
   51:         *
   52:         * @warning A reference to the discretization @p mesh is stored inside this c
lass.
   53:         *          Therefore, changing @p mesh outside requires also
   54:         *          to call method @p Derive_Matrix_Pattern explicitely.
   55:         *
   56:         * @see Derive_Matrix_Pattern
   57:         */
   58:        explicit CRS_Matrix(Mesh const & mesh);
   59:
   60:       /**
   61:         * Destructor.
   62:         */
   63:        ~CRS_Matrix()
   64:        {}
```

```
   65:
   66:         /**
   67:          * Generates the sparse matrix pattern and overwrites the existing pattern.
   68:          *
   69:          * The sparse matrix pattern is generated but the values are 0.
   70:          */
   71:         void Derive_Matrix_Pattern();
   72:
   73:          /**
   74:          * Calculates the entries of f.e. stiffness matrix and load/rhs vector @p f f
or the Laplace operator in 2D.
   75:          * No memory is allocated.
   76:          *
   77:          * @param[in,out] f (preallocated) rhs/load vector
   78:          */
   79:         void CalculateLaplace(std::vector<double> &f);
   80:
   81:         /**
   82:          * Applies Dirichlet boundary conditions to stiffness matrix and to load vect
or @p f.
   83:          * The <a href="https://www.jstor.org/stable/2005611?seq=1#metadata_info_tab_
contents">penalty method</a>
   84:          * is used for incorporating the given values @p u.
   85:          *
   86:          * @param[in]     u (global) vector with Dirichlet data
   87:          * @param[in,out] f load vector
   88:          */
   89:         void ApplyDirichletBC(std::vector<double> const &u, std::vector<double> &f);
   90:
   91:         /**
   92:          * Extracts the diagonal elemenst of the sparse matrix.
   93:          *
   94:          * @param[in,out]  d  (prellocated) vector of diagonal elements
   95:          */
   96:         void GetDiag(std::vector<double> &d) const;
   97:
   98:         /**
   99:          * Performs the matrix-vector product  w := K*u.
  100:          *
  101:          * @param[in,out] w resulting vector (preallocated)
  102:          * @param[in]     u vector
  103:          */
  104:         void Mult(std::vector<double> &w, std::vector<double> const &u) const;
  105:
  106:          /**
  107:          * Calculates the defect/residuum w := f - K*u.
  108:          *
  109:          * @param[in,out] w resulting vector (preallocated)
  110:          * @param[in]     f load vector
  111:          * @param[in]     u vector
  112:          */
  113:         void Defect(std::vector<double> &w,
  114:                     std::vector<double> const &f, std::vector<double> const &u) const
;
  115:
  116:         /**
  117:                   * Number rows in matrix.
  118:                   * @return number of rows.
  119:                   */
  120:         int Nrows() const
  121:            {return _nrows;}
  122:
  123:         /**
  124:                   * Show the matrix entries.
  125:                   */
  126:         void Debug() const;
  127:
  128:         /**
```

```
  129:                  * Finds in a CRS matrix the access index for an entry at row @p row
 and column @p col.
  130:                  *
  131:                  * @param[in] row       row index
  132:                  * @param[in] col       column index
  133:                  * @return index for element (@p row, @p col). If no appropriate ent
ry exists then -1 will be returned.
  134:                  *
  135:                  * @warning assert() stops the function in case that matrix element
(@p row, @p col) doesn't exist.
  136:                 */
  137:         int fetch(int row, int col) const;
  138:
  139:        /**
  140:         * Adds the element stiffness matrix @p ske and the element load vector @p fe
  141:         * of one triangular element with linear shape functions to the appropriate p
ositions in
  142:         * the stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik).
  143:         *
  144:         * @param[in]     ial   node indices of the three element vertices
  145:         * @param[in]     ske   element stiffness matrix
  146:         * @param[in]     fe    element load vector
  147:         * @param[in,out] f         distributed local vector storing the right hand s
ide
  148:         *
  149:         * @warning Algorithm assumes  linear triangular elements (ndof_e==3).
  150:         */
  151:         void AddElem_3(int const ial[3], double const ske[3][3], double const fe[3],
std::vector<double> &f);
  152:
  153:          /**
  154:         * Compare @p this CRS matrix with an external CRS matrix stored in C-Style.
  155:         *
  156:         * The method prints statements on differences found.
  157:         *
  158:         * @param[in]     nnode  row number of external matrix
  159:         * @param[in]     id     start indices of matrix rows of external matrix
  160:         * @param[in]     ik     column indices of external matrix
  161:         * @param[in]     sk     non-zero values of external matrix
  162:         *
  163:         * @return true iff all data are identical.
  164:         */
  165:         bool Compare2Old(int nnode, int const id[], int const ik[], double const sk[]
) const;
  166:
  167:     private:
  168:         Mesh const & _mesh;       //!< reference to discretization
  169:         int _nrows;               //!< number of rows in matrix
  170:         int _nnz;                 //!< number of non-zero entries
  171:         std::vector<int> _id;     //!< start indices of matrix rows
  172:         std::vector<int> _ik;     //!< column indices
  173:         std::vector<double> _sk; //!< non-zero values
  174:
  175: };
  176:
  177:
  178: #endif
```

```cpp
#include "vdop.h"
#include "getmatrix.h"
#include "jacsolve.h"

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

// ########################################################################
//      const int neigh[], const int color,
//      const MPI::Intracomm& icomm,
void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
{
    const double omega   = 1.0;
    const int    maxiter = 1000;
    const double tol  = 1e-5,                   // tolerance
                 tol2 = tol * tol;              // tolerance^2

    int nrows = SK.Nrows();                     // number of rows == number of columns
    assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );

    cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s"  << endl;
    //  Choose initial guess
    for (int k = 0; k < nrows; ++k)
    {
        u[k] = 0.0;                             //  u := 0
    }

    vector<double> dd(nrows);                   // matrix diagonal
    vector<double>  r(nrows);                   // residual
    vector<double>  w(nrows);                   // correction

    SK.GetDiag(dd);                             //  dd := diag(K)
    ////DebugVector(dd);{int ijk; cin >> ijk;}

    //  Initial sweep
    SK.Defect(r, f, u);                         //  r := f - K*u

    vddiv(w, r, dd);                            //  w := D^{-1}*r
    double sigma0 = dscapr(w, r);               // s0 := <w,r>

    // Iteration sweeps
    int iter  = 0;
    double sigma = sigma0;
    while ( sigma > tol2 * sigma0 && maxiter > iter)
    {
        ++iter;
        vdaxpy(u, u, omega, w );                //  u := u + om*w
        SK.Defect(r, f, u);                     //  r := f - K*u
        vddiv(w, r, dd);                        //  w := D^{-1}*r
        sigma = dscapr(w, r);                   // s0 := <w,r>
//              cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
    }
    cout << "aver. Jacobi rate :  " << exp(log(sqrt(sigma / sigma0)) / iter) << "  (
" << iter << " iter)" << endl;
    cout << "final error: " << sqrt(sigma / sigma0) << " (rel)    " << sqrt(sigma) <<
 " (abs)\n";

    return;
}
```

```
    1: #ifndef JACSOLVE_FILE
    2: #define JACSOLVE_FILE
    3: #include "getmatrix.h"
    4: #include <vector>
    5:
    6:
    7: /**
    8:  * Solves linear system of equations  K @p u = @p f  via the Jacobi iteration.
    9:  * We use a distributed symmetric  CSR matrix @p SK and initial guess of the
   10:  * solution is set to 0.
   11:  * @param[in] SK        CSR matrix
   12:  * @param[in] f         distributed local vector storing the right hand side
   13:  * @param[out] u        accumulated local vector storing the solution.
   14: */
   15: void JacobiSolve(CRS_Matrix const &SK, std::vector<double> const &f, std::vector<dou
ble> &u);
   16:
   17:
   18: #endif
```

```cpp
#include "benchmarks.h"
#include "benchmark_tests.h"
#include "factorization_solve_tests.h"
#include <iostream>

int main()
{
    // ---------------------------------- 1. ----------------------------------
    // results in file "test_system.txt"



    // ---------------------------------- 2. ----------------------------------
    //      Memory      FLOPS       Read-write operations
    // (A)  2N          2N          2N
    // (B)  MN + N      2NM         2NM + M
    // (C)  ML + LM     2LNM        2LNM + MN
    // (D)   (p+1) + N   3N(p+1)     N(2 + 3(p+1))


    // ---------------------------------- 3. ----------------------------------
    // implementation in file "benchmarks.cpp"


    // ---------------------------------- 4.& 6. ----------------------------------
    vector<vector<double>> results;

    results.push_back(test_A(250, 50000000, scalar));
    results.push_back(test_A(250, 50000000, Kahan_skalar));
    results.push_back(test_A(250, 50000000, scalar_cBLAS));
    //                  Timing  GFLOPS  GiByte/s
    // ----------------------------------------
    // scalar           0.039   2.4     19
    // Kahan_skalar     0.037   2.5     20
    // scalar_cBLAS     0.032   2.9     23


    results.push_back(test_B(100, 20000, 10000, MatVec));
    results.push_back(test_B(100, 20000, 10000, MatVec_cBLAS));
    //                  Timing  GFLOPS  GiByte/s
    // ----------------------------------------
    // MatVec           0.1     3.6     29
    // MatVec_cBLAS     0.074   5       40


    results.push_back(test_C(25, 500, 1000, 1500, MatMat));
    results.push_back(test_C(25, 500, 1000, 1500, MatMat_cBLAS));
    //                  Timing  GFLOPS  GiByte/s
    // ----------------------------------------
    // MatMat           0.57    2.5     20
    // MatMat_cBLAS     0.019   75      6e+02   // unrealistic


    results.push_back(test_D(100, 100, 1000000));
    //                  Timing  GFLOPS  GiByte/s
    // ----------------------------------------
    //                  0.11    2.5     20


    cout << endl << "Timing\tGFLOPS\tGiByte/s" << endl;
    cout << "-----------------------------" << endl;
    for (size_t i = 0; i < results.size(); ++i)
        cout << results[i][0] << "\t" << results[i][1] << "\t" << results[i][2] << e
ndl;
    cout << endl;

```

```
    68:
    69:
    70:     // ---------------------------------- 5. ----------------------------------
    71:     // 5.(a) Observation: time to calculate norm is approximately half the time as f
or the scalar product.
    72:     // Reason: only have to access entries of x, so less memory that has to be acces
sed
    73:     //
    74:     // 5.(b) Runtime for Kahan_scalar is roughly the same as for the normal scalar p
roduct
    75:
    76:
    77:
    78:     // ---------------------------------- 6. ----------------------------------
    79:     // see 4.
    80:
    81:
    82:     // ---------------------------------- 7. ----------------------------------
    83:     CheckCorrectness();
    84:     // Checked correctness by computing the inverse of A
    85:
    86:     CheckDuration(5000);
    87:     // The solving time per RHS scales roughly with factor 1/n_rhs
    88:
    89:
    90:
    91:     // ---------------------------------- 8. ----------------------------------
    92:     // done seperately
    93:
    94:
    95:
    96:
    97:
    98:
    99:     return 0;
   100:
   101: }
```

```cpp
#include "userset.h"
#include <cmath>


double FunctF(double const x, double const y)
{
// return  std::sin(3.14159*1*x)*std::sin(3.14159*1*y);
//  return 16.0*1024. ;
// return (double)1.0 ;
    return x * x * std::sin(2.5 * 3.14159 * y);
}

double FunctU(const double /* x */, double const /* y */)
{
    return 1.0 ;
}
```

```
#ifndef USERSET_FILE
#define USERSET_FILE
#include <cmath>
/**
 * User function: f(@p x,@p y)
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value for right hand side f(@p x,@p y)
 */
double FunctF(double const x, double const y);

/**
 * User function: u(@p x,@p y)
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value for solution vector u(@p x,@p y)
 */
double FunctU(double const x, double const y);


/**
 * User function: f(@p x,@p y) = @f$ x^2 \sin(2.5\pi y)@f$.
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value f(@p x,@p y)
 */
inline double fNice(double const x, double const y)
{
    return x * x * std::sin(2.5 * 3.14159 * y);
}

/**
 * User function: f(@p x,@p y) = 0$.
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value 0
 */
inline double f_zero(double const x, double const y)
//double f_zero(double const  /*x*/, double const /*y*/)
{
    return 0.0 + 0.0*(x+y);
}

#endif
```

```
    1: #include "vdop.h"
    2: #include <cassert>               // assert()
    3: #include <cmath>
    4: #include <iostream>
    5: #include <vector>
    6: using namespace std;
    7:
    8:
    9: void vddiv(vector<double> &x, vector<double> const &y,
   10:            vector<double> const &z)
   11: {
   12:     assert( x.size() == y.size() && y.size() == z.size() );
   13:     size_t n = x.size();
   14:
   15:     for (size_t k = 0; k < n; ++k)
   16:     {
   17:         x[k] = y[k] / z[k];
   18:     }
   19:     return;
   20: }
   21:
   22: //*****************************************************************************
   23:
   24: void vdaxpy(std::vector<double> &x, std::vector<double> const &y,
   25:             double alpha, std::vector<double> const &z )
   26: {
   27:     assert( x.size() == y.size() && y.size() == z.size() );
   28:     size_t n = x.size();
   29:
   30:     for (size_t k = 0; k < n; ++k)
   31:     {
   32:         x[k] = y[k] + alpha * z[k];
   33:     }
   34:     return;
   35: }
   36: //*****************************************************************************
   37:
   38: double dscapr(std::vector<double> const &x, std::vector<double> const &y)
   39: {
   40:     assert( x.size() == y.size());
   41:     size_t n = x.size();
   42:
   43:     double    s = 0.0;
   44:     for (size_t k = 0; k < n; ++k)
   45:     {
   46:         s += x[k] * y[k];
   47:     }
   48:
   49:     return s;
   50: }
   51:
   52: //*****************************************************************************
   53: void DebugVector(vector<double> const &v)
   54: {
   55:     cout << "\nVector  (nnode = " << v.size() << ")\n";
   56:     for (size_t j = 0; j < v.size(); ++j)
   57:     {
   58:         cout.setf(ios::right, ios::adjustfield);
   59:         cout << v[j] << "   ";
   60:     }
   61:     cout << endl;
   62:
   63:     return;
   64: }
   65: //*****************************************************************************
   66: bool CompareVectors(std::vector<double> const &x, int const n, double const y[], dou
ble const eps)
   67: {
```

```
68:     bool bn = (static_cast<int>(x.size()) == n);
69:     if (!bn)
70:     {
71:         cout << "#########   Error: " << "number of elements" << endl;
72:     }
73:     //bool bv = equal(x.cbegin(),x.cend(),y);
74:     bool bv = equal(x.cbegin(), x.cend(), y,
75:                     [eps](double a, double b) -> bool
76:     { return std::abs(a - b) < eps * (1.0 + 0.5 * (std::abs(a) + std::abs(a))); }
77:                    );
78:     if (!bv)
79:     {
80:         assert(static_cast<int>(x.size()) == n);
81:         cout << "#########   Error: " << "values" << endl;
82:     }
83:     return bn && bv;
84: }
```

```
    1: #ifndef VDOP_FILE
    2: #define VDOP_FILE
    3: #include <vector>
    4:
    5: /** @brief  Element-wise vector divison x_k = y_k/z_k.
    6:  *
    7:  * @param[out] x  target vector
    8:  * @param[in]  y  source vector
    9:  * @param[in]  z  source vector
   10:  *
   11:  */
   12: void vddiv(std::vector<double> & x, std::vector<double> const& y,
   13:                                     std::vector<double> const& z);
   14:
   15: /** @brief  Element-wise daxpy operation x(k) = y(k) + alpha*z(k).
   16:  *
   17:  * @param[out] x  target vector
   18:  * @param[in]  y  source vector
   19:  * @param[in]  alpha  scalar
   20:  * @param[in]  z  source vector
   21:  *
   22:  */
   23: void vdaxpy(std::vector<double> & x, std::vector<double> const& y,
   24:                         double alpha, std::vector<double> const& z );
   25:
   26:
   27: /** @brief  Calculates the Euclidian inner product of two vectors.
   28:  *
   29:  * @param[in]  x vector
   30:  * @param[in]  y vector
   31:  * @return Euclidian inner product @f$\langle x,y \rangle@f$
   32:  *
   33:  */
   34: double dscapr(std::vector<double> const& x, std::vector<double> const& y);
   35:
   36:
   37: /**
   38:  * Print entries of a vector.
   39:  * @param[in] v     vector values
   40: */
   41: void DebugVector(std::vector<double> const &v);
   42:
   43: /** @brief  Compares an STL vector with POD vector.
   44:  *
   45:  * The accuracy criteria @f$ |x_k-y_k| < \varepsilon \left({1+0.5(|x_k|+|y_k|)}\righ
t) @f$
   46:  * follows the book by
   47:  * <a href="https://www.springer.com/la/book/9783319446592">Stoyan/Baran</a>, p.8.
   48:  *
   49:  * @param[in]  x    STL vector
   50:  * @param[in]  n    length of POD vector
   51:  * @param[in]  y    POD vector
   52:  * @param[in]  eps  relative accuracy criteria (default := 0.0).
   53:  * @return true iff pairwise vector elements are relatively close to each other.
   54:  *
   55:  */
   56: bool CompareVectors(std::vector<double> const& x, int n, double const y[], double co
nst eps=0.0);
   57:
   58: #endif
```