ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

Link to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single ".pdf" file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. "Solving" the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Problem 3-6: Sparse Approximate Inverse (SPAI [GRH97])

This problem studies the least squares aspects of the SPAI method, a technique used in the numerical solution of partial differential equations. We encounter an "exotic" sparse matrix technique and rather non-standard least squares problems. Please note that the matrices to which SPAI techniques are applied will usually be huge and extremely sparse, say, of dimension $10^7 \times 10^7$ with only 10^8 non-zero entries. Therefore sparse matrix techniques must be applied.

Requires only familiarity with [Lecture → Section 3.1] and [Lecture → Section 3.2].



Sub-problem (3-6.d) is connected with [Lecture → Section 10.3]. In case this section had not (yet) been covered in the course that sub-problem should be skipped.

Let $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, be a regular sparse matrix with at most $n \ll N$ non-zero entries per row and column. We define the space of matrices with the same pattern (on non-zero entries) as $\mathbf{A}$:

$$\mathcal{P}(\mathbf{A}) := \{\mathbf{X} \in \mathbb{R}^{N,N} : \ (\mathbf{A})_{ij} = 0 \ \Rightarrow \ (\mathbf{X})_{ij} = 0\} \ . \tag{3.6.1}$$

The "primitive" SPAI (sparse approximate inverse) $\mathbf{B}$ of $\mathbf{A}$ is defined as

$$\mathbf{B} := \operatorname*{argmin}_{\mathbf{X} \in \mathcal{P}(\mathbf{A})} \|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F \ , \tag{3.6.2}$$

where $\|\cdot\|_F$ stands for the Frobenius norm [Lecture → Def. 3.4.4.17]. The solution of (3.6.2) can be used as a so-called preconditioner for the acceleration of iterative methods for the solution of linear systems of equations, see [Lecture → Chapter 10]. An extended "self-learning" variant of the SPAI method is presented in [**GRH97**].

(3-6.a) ⚂ (20 min.) Show that the columns of $\mathbf{B}$ can be computed independently of each other by solving linear least squares problems. In the statement of these linear least squares problems write $\mathbf{b}_i$ for the columns of $\mathbf{B}$.

HINT 1 for (3-6.a): Remember that $\|\mathbf{M}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{M}\mathbf{e}_i\|_2^2$, where the $\mathbf{e}_i$ are the Cartesian unit vectors and the norm is the standard Euclidean norm. ⌟

SOLUTION of (3-6.a):

Let $\mathbf{x}_i, \mathbf{b}_i, \mathbf{a}_i$ denote the i-th columns of the matrices $\mathbf{X}, \mathbf{B}, \mathbf{A}$ in (3.6.2), respectively. Notice, that the values in the i-th column of $\mathbf{I} - \mathbf{A}\mathbf{X}$ depend only on the values in the vector $\mathbf{x}_i$. Hence, in order to minimize the Frobenius norm $\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2$ one needs to (independently) minimize each of the terms $\|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2$:

$$\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{I}\mathbf{e}_i - \mathbf{A}\mathbf{X}\mathbf{e}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2 .$$

The corresponding minimization problems read

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathcal{P}((\mathbf{A})_{:,i})} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2 , \quad \text{for} \quad i = 1, \ldots, N, \tag{3.6.3}$$

where for a vector $\mathbf{a} \in \mathbb{R}^N$ we define $\mathcal{P}(\mathbf{a}) := \{\mathbf{x} \in \mathbb{R}^N : (\mathbf{a})_i = 0 \ \Rightarrow \ (\mathbf{x})_i = 0\}$, which is the set of vectors of the same length, with potentially non-zero components at the same positions as $\mathbf{a}$

Note that (3.6.3) is not yet a least-squares problem in standard form. To convert it into such, we have to introduce

$$\mathcal{J} := \{j : (\mathbf{A})_{j,i} \neq 0\} \quad , \quad m := \sharp\mathcal{J} \, .$$

and define

$$\mathbf{A}_i := (\mathbf{A})_{:,\mathcal{J}} \in \mathbb{R}^{N,m} \, .$$

Then (3.6.3) can be stated as

$$\mathbf{y} \in \mathbb{R}^m: \quad \|\mathbf{e}_i - \mathbf{A}_i\mathbf{y}\|_2 \to \min \, .$$

▲

(3-6.b) ⚄ (60 min.) [depends on Sub-problem (3-6.a)]

Implement an efficient C++ function

```
Eigen::SparseMatrix<double> spai(const
   Eigen::SparseMatrix<double>& A);
```

for the computation of $\mathbf{B}$ according to (3.6.2). You may rely on the normal equations associated with the linear least squares problems for the columns of $\mathbf{B}$ or you may simply invoke the least squares solver of EIGEN, see [Lecture → Code 3.4.3.14].

HINT 1 for (3-6.b): For an efficient implementation, you can exploit the internal CRS data structure underlying `Eigen::SparseMatrix<double>`. Use

- `Eigen::SparseMatrix<double>::valuePtr()`,
- `Eigen::SparseMatrix<double>::innerIndexPtr()` and
- `Eigen::SparseMatrix<double>::outerIndexPtr()`, which

give the raw CCS format, in which the sparse matrix is stored, see [Lecture → § 2.7.1.4].

Moreover, build the output matrix $\mathbf{B}$ in EIGEN using the intermediate triplet format (`std::vector<Triplet<double>>`), see [Lecture → Section 2.7.2], before converting it to `Eigen::SparseMatrix<double>`. Use the method `setFromTriplets()`. ⌟

SOLUTION of (3-6.b):

The formula (3.6.3) is not yet in the canonical form. Fix i. Define $m_i := nnz(\mathbf{a}_i)$. Notice that $\mathcal{P}(\mathbf{a}_i) \simeq \mathbb{R}^{m_i}$ via the mapping $\mathbf{x} \mapsto [x_{j_1}, \ldots, x_{j_{m_i}}] =: \tilde{\mathbf{x}}$, where j_k, $k = 1, \ldots, m_i$ are the indices s.t. $a_{j_k} \neq 0$.

The product $\mathbf{Ax}$ can be expressed, equivalently, with $\mathbf{C}\tilde{\mathbf{x}}$, where $\mathbf{C} \in \mathbb{R}^{N \times m_i}$ is defined as the matrix $\mathbf{A}$, without the columns i, s.t. $a_i = 0$:

$$\mathbf{C} := [(\mathbf{A})_{:,j_1}, \ldots, (\mathbf{A})_{:,j_{m_i}}] \, .$$

We obtain, *cf.* the solution of Sub-problem (3-6.a),

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathbb{R}^{m_i}} \|\mathbf{e}_i - \mathbf{C}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \ldots, N.$$

This canonical form can be solved with the standard normal formula.

See Equation 3.6.6 for the EIGEN-based implementation in C++.

C++11-code 3.6.4: Computation of $\mathbf{B}$.

```
2  Eigen::SparseMatrix<double> spai(Eigen::SparseMatrix<double>& A) {
3    // Size check
4    assert(A.rows() == A.cols() && "Matrix must be square!");
5    unsigned int N = A.rows();
6
7    Eigen::SparseMatrix<double> B(N, N);
8
9    // TODO: (3-6.b) Compute the sparse approximate inverse of A by
         exploiting
10   // Eigen's internal CCS data structure
11   // START
12
13   // Needed to make sure ...Ptr functions return
14   // arrays specified in CCS format
15   A.makeCompressed();
16
17   // Obtain pointers to data of A
18   double* valPtr = A.valuePtr();
19   index_t* innPtr = A.innerIndexPtr();
20   index_t* outPtr = A.outerIndexPtr();
21
22   // Create vector for triplets of B and reserve enough space
23   std::vector<Eigen::Triplet<double>> triplets;
24   triplets.reserve(A.nonZeros());
25
26   // Loop over each column of A
27   for (unsigned int i = 0; i < N; ++i) {
28     // Number of non zeros in current column of A
29     index_t nnz_i = outPtr[i + 1] - outPtr[i];
30     // If column is empty:  skip column (matrix is not invertible)
31     if (nnz_i == 0) continue;
32
33     // Temporarily create a (small, dense) matrix on which normal
         formula
34     // will be applied.  We project the space P(A)
35     // onto P(a_i)
36     Eigen::SparseMatrix<double> C(N, nnz_i);
37     std::vector<Eigen::Triplet<double>> C_triplets;
38     C_triplets.reserve(nnz_i * nnz_i);
39
40     // Need to build matrix C.  To this end we remove all columns
41     // from A for which a_i = 0.
42     // To this end:  loop over all non-zero entries of the i-th column
43     // This loop has length n
44     for (unsigned int k = outPtr[i]; k < outPtr[i + 1]; ++k) {
45       // Row of this non-zero entry
46       index_t row_k = innPtr[k];
47       // Number of non-zero entries for row_k-th column
48       index_t nnz_k = outPtr[row_k + 1] - outPtr[row_k];
49       // Loop over all non-zeros of row_k-th-column
50       // This loop has length complexity n
51       for (unsigned int l = 0; l < nnz_k; ++l) {
52         unsigned int innIdx = outPtr[row_k] + l;
53         C_triplets.emplace_back(Eigen::Triplet<double>(
54             innPtr[innIdx], k - outPtr[i], valPtr[innIdx]));
55       }
56     }
57     C.setFromTriplets(C_triplets.begin(), C_triplets.end());
58     C.makeCompressed();
```

```
59 |
60 |   // Compute C^T C and solve normal equation.
61 |   // Complexity of product:  O(n^3)
62 |   // Complexity of solve:  O(n^3)
63 |   // Size of b is at most n.
64 |   Eigen::SparseMatrix<double> S = C.transpose() * C;
65 |   Eigen::MatrixXd M = Eigen::MatrixXd(S);
66 |   Eigen::VectorXd xt = C.row(i).transpose();
67 |   Eigen::VectorXd b = M.partialPivLu().solve(xt);
68 |
69 |   // Loop of length at most n.
70 |   for (unsigned int k = 0; k < b.size(); ++k) {
71 |     triplets.emplace_back(
72 |         Eigen::Triplet<double>(innPtr[outPtr[i] + k], i, b(k)));
73 |   }
74 | }
75 |
76 | // Build and return SPAI preconditioner
77 | B.setFromTriplets(triplets.begin(), triplets.end());
78 | // END
79 |
80 | B.makeCompressed();
81 | return B;
82 | }
```

Get it on GitLab (spai.hpp).

▲

(3-6.c) ⚀ (15 min.) [depends on Sub-problem (3-6.b)]

What is the total asymptotic computational effort of your implementation of `spai()` in terms of the problem size parameters N and n.

SOLUTION of (3-6.c):

In the body of the `for` loop (of size N) the matrix $\mathbf{C}$ is assembled with two for loops of length at most n, then the matrix $\mathbf{C}^\top\mathbf{C}$ is assembled with complexity $O(n^3)$ (note: $\mathbf{C}$ is sparse, with only n non-zeros per column), and then the corresponding linear system of size $n \times n$ is solved (complexity $O(n^3)$). Hence, the resulting total complexity is dominated by $O(Nn^3)$.

▲

(3-6.d) ⚁ (45 min.) [depends on Sub-problem (3-6.b)]

For $n \in \mathbb{N}$, consider a sparse s.p.d. [Lecture → Def. 1.1.2.6] matrix $\mathbf{A}$ given by the following C++-code

C++11-code 3.6.5: Initialization of matrix $\mathbf{A}$

```
2 | Eigen::SparseMatrix<double> init_A(unsigned int n) {
3 |   Eigen::SparseMatrix<double> L(n, n);
4 |   Eigen::SparseMatrix<double> R(n, n);
5 |   L.reserve(n);
6 |   R.reserve(n + 2 * (n - 1));
7 |   L.insert(0, 0) = std::exp(1. / n);
```

```
  R.insert(0, 0) = 2.;
  for (unsigned int i = 1; i < n; ++i) {
    L.insert(i, i) = std::exp((i + 1.) / n);
    R.insert(i, i) = 2.;
    R.insert(i - 1, i) = -1.;
    R.insert(i, i - 1) = -1.;
  }
  Eigen::SparseMatrix<double> A = kroneckerProduct(L, R);
  A.makeCompressed();
  return A;
}
```

Get it on GitLab (spai.hpp).

Solve the linear system of equations $\mathbf{Ax} = \mathbf{b}$ using EIGEN's **ConjugateGradient** (EIGEN documentation) iterative solver with initial guess $\mathbf{0}$ and default tolerance. Do this without a preconditioner and using the preconditioner $\frac{1}{2}(\mathbf{B} + \mathbf{B}^T)$, where $\mathbf{B}$ solves (3.6.2). Use $\mathbf{b} = [1,1,\ldots,1]^\top$. Write a C++ function

```
std::vector<std:pair<unsigned int, unsigned int>>
testSPAIPrecCG(unsigned int L);
```

that returns the number of CG iterations (with and without the SPAI preconditioner) versus the system size $N = n^2$ for $n = 2^{1,\ldots,L}$.

HINT 1 for (3-6.d): To use EIGEN's **ConjugateGradient** class without a preconditioner you can use EIGEN's **IdentityPreconditioner** class.
Be aware that this is not the default preconditioner of **ConjugateGradient**! ⌟

HINT 2 for (3-6.d): To use EIGEN's **ConjugateGradient** class with $\frac{1}{2}(\mathbf{B} + \mathbf{B}^T)$ as a preconditiioner, use the method `precoditioner()` and the provided class **SymmetrizedSPAIPreconditioner**. ⌟

SOLUTION of (3-6.d):

C++11-code 3.6.6: Testing the SPAI preconditioner.

```
std::vector<std::pair<unsigned int, unsigned int>> testSPAIPrecCG(
    unsigned int L) {
  std::vector<std::pair<unsigned int, unsigned int>> cg_iterations(L);

  // TODO: (3-6.d) Compute the iterations needed by the
     ConjugateGradient solver
  // without a preconditioner as well as with one

  // START

  // initialize r.h.s.  with
  // its maximal size we will use head() to get smaller versions of it
  Eigen::VectorXd b = Eigen::VectorXd::Ones(1 << (2 * L));

  // loop over n = 2^{1,...,L}, keep two running variables
  for (unsigned int n = 2, i = 0; n <= (1 << L); n <<= 1, ++i) {
    unsigned int N = n * n;
    Eigen::SparseMatrix<double> A = init_A(n);
    Eigen::SparseMatrix<double> B = spai(A);
    // compute Ax = b without preconditioner
    Eigen::ConjugateGradient<Eigen::SparseMatrix<double>,
```

```
                                Eigen::Lower | Eigen::Upper,
                                Eigen::IdentityPreconditioner>
        cg(A);
    Eigen::VectorXd sol = cg.solve(b.head(N));

    // compute Ax = b with preconditioner
    Eigen::ConjugateGradient<
        Eigen::SparseMatrix<double>, Eigen::Lower | Eigen::Upper,
        SymmetrizedSPAIPreconditioner<Eigen::SparseMatrix<double>>>
        cg_preconditioner;
    // note the wrapper around B.transpose():
    // this changes the storage order which has to match when
       executing
    // operator+
    SymmetrizedSPAIPreconditioner<Eigen::SparseMatrix<double>> preconditioner(
        0.5 * (B + Eigen::SparseMatrix<double>(B.transpose())));
    cg_preconditioner.preconditioner() = preconditioner;
    cg_preconditioner.compute(A);
    Eigen::VectorXd sol_prec = cg_preconditioner.solve(b.head(N));

    // check if both solutions coincide
    assert((sol - sol_prec).norm() <= 1e-9);

    cg_iterations[i] = {cg.iterations(), cg_preconditioner.iterations()};
  }
  // END

  return cg_iterations;
}
```

Get it on GitLab (spai.hpp).

▲

End Problem 3-6 , 140 min.