

ETH Lecture 401-4671-00L Advanced Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2023

(C) Seminar für Angewandte Mathematik, ETH Zürich

URL: <https://people.math.ethz.ch/~grsam/ADVNCSE>

Link to the current version of this homework problem collection

Contents

0.1	General Information	2
0.1.1	Weekly Homework Assignments	2
0.1.2	Importance of Homework	3
0.1.3	Corrections and Grading of Assignments	3
0.1.4	Codes and Templates	3
0.1.5	Hints and Solutions	3
1	Boundary Element Methods (BEM)	4
2	Local Low-Rank Compression of Non-Local Operators	5
	Problem 2-1: Efficient Computation of Gravitational Forces	5
	Problem 2-2: Imbalanced cluster tree for 1D clustering	10
	Problem 2-3: Tensor-Product Chebychev Interpolation	11
	Problem 2-4: Local Low-Rank Approximation by Clustering	14
	Problem 2-5: Evaluation of Trigonometric Polynomials	25
	Problem 2-6: Merging of low-rank matrices	29
	Problem 2-7: A special family of hierarchical matrices	31
3	Convolution Quadrature	34
	Problem 3-1: Computing with lower triangular Toeplitz matrices	34
	Problem 3-2: Abel integral equation	37
	Problem 3-3: Convolution-based Absorbing Boundary Condition	41
	Problem 3-4: Fractional Parabolic Evolution Problem	44
4	(Algebraic) Multigrid Methods	48
	Problem 4-1: Stationary Linear Iterations	48
	Problem 4-2: Matrix-Free Multigrid for Scalar Dirichlet BVPs	51
	Problem 4-3: Galerkin Construction of Coarse-Grid Matrices	55
	Problem 4-4: The Ruge-Stüben Algebraic Multigrid Method: A Case Study	59
	Problem 4-5: The Ruge-Stüben Algebraic Multigrid Method: Implementation	61

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every one or two weeks, we publish a list of problems (cf. Section ??) that should be solved.

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 in a difficult problem and you tried to find a solution for a sufficient amount of time.

Note that you are expected to answer questions on homework coding projects in the mid-term **code review!**

0.1.3 Corrections and Grading of Assignments

The assistant(s) is/are happy to look at your solution *upon request* and you can simply give her or him your handwritten solution during the tutorial class. Do not hesitate to discuss your solutions even if those are incomplete and/or incorrect. Feedback from the assistants is most valuable in case of incomplete or incorrect solutions. You will get feedback about your submissions usually during tutorial classes.

You can also submit your codes to the assistant(s) by *uploading them to your branch* of the **Git code repository**.

0.1.4 Codes and Templates

For each problem involving coding you will find templates on [GitLab](#) (the specific link is provided in the problem sheet). Templates may be used as starting point for your solutions. All solutions we provide will be based on the templates. You are not forced to use the templates, but using them will allow you to focus on the problems. This will also simplify the discussion with the assistant(s).

All templates come with a self-contained `CMake` file that can be used to compile only the problem you are working on, see the explanations included in the Git repository.

You can obtain templates and solutions by simply cloning the AdvNumMeth **Git code repository**

0.1.5 Hints and Solutions

Hints and solutions are kept in separate “.pdf” files. The URLs for these files will be supplied in this homework problem/project collection.

Chapter 1

Boundary Element Methods (BEM)

Chapter 2

Local Low-Rank Compression of Non-Local Operators

Problem 2-1: Efficient Computation of Gravitational Forces

As we saw in [Lecture → Section 2.1.2], long-range interactions between “particles” invariably lead to non-local models. A prominent example are the gravitational forces acting between the stars of a galaxy, see [Lecture → § 2.1.2.1]. In this problem we compute those forces approximately using the clustering-type algorithm presented in [Lecture → § 2.1.2.8].

The coding part of this problem is based on C++ and EIGEN.

Templates: [Get it on](#)  [GitLab](#).

Master solution: [Get it on](#)  [GitLab](#).


▷ problem name/problem code folder: GravitationalForces in [Code repository](#)

We consider a *planar* cluster of $n \in \mathbb{N}$ stars, which are regarded as mere mass points. Assuming a non-dimensional model (after scaling), the i -th star has mass $m_i > 0$ and its position is described by the coordinate vector $\mathbf{x}^i = \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix} \in \mathbb{R}^2$, $i = 1, \dots, n$. According to the law of gravitational attraction, the force acting on star j due to star i is

$$\mathbf{f}^{j,i} = \frac{m_i m_j}{4\pi} \frac{\mathbf{x}^i - \mathbf{x}^j}{\|\mathbf{x}^i - \mathbf{x}^j\|^3}, \quad (2.1.1)$$

where we employed a scaling that assigns the value 1 to the gravitational constant. The total force on star j is then obtained by simply adding up the forces (2.1.1), cf. [Lecture → Eq. (2.1.2.2)]

$$\mathbf{f}^j = \frac{m_j}{4\pi} \sum_{\substack{i=1 \\ i \neq j}}^n m_i \frac{\mathbf{x}^i - \mathbf{x}^j}{\|\mathbf{x}^j - \mathbf{x}^i\|^3}, \quad j = 1, \dots, n. \quad (2.1.2)$$

(2-1.a)  (20 min.) In the file `gravitationalforces.cpp` implement a C++ function

```
std::vector<Eigen::Vector2d>
initStarPositions(unsigned int n, double mindist);
```

that places n stars uniformly randomly in the unit square $[0, 1]^2$ ensuring a minimal distance `mindist` between them. That minimal distance must be smaller than $\frac{1}{2\sqrt{n}}$, otherwise it is set to that value.

SOLUTION for (2-1.a) → [2-1-1-0:gfs1.pdf](#)



(2-1.b) 📄 (30 min.) In the file `gravitationalforces.cpp` supply a C++ function

```
std::vector<Eigen::Vector2d> computeForces_direct (
    const std::vector<Eigen::Vector2d> &masspositions,
    const std::vector<double> &masses);
```

that computes and returns all the forces $f^j \in \mathbb{R}^2$, $j = 1, \dots, n$, by direct evaluation of the formula (2.1.2). The parameter `masspositions` passes the coordinate vectors x^i , whereas the masses m_i are supplied in `masses`.

SOLUTION for (2-1.b) → [2-1-2-0:.pdf](#) ▲

At the core of the algorithm of [Lecture → § 2.1.2.8] is a **quadtree** data structure. The **nodes** of that tree should hold the following information:

- A characterization of a **sub-cluster** of the stars, which can be stored as a vector of their indices.
- The location of the center of gravity and the total mass for the associated sub-cluster.
- A Cartesian **bounding box** for the sub-cluster.
- Pointers to four **son nodes**, some of which may be nil.

These requirements are reflected by the design of the class **StarQuadTree** listed as Code 2.1.5

C++ code 2.1.5: Class for representing a quadtree.
 Get it on 🐙 [GitLab](#) (`gravitationalforces.h`).

```
2 class StarQuadTree {
3   public:
4     StarQuadTree() = delete;
5     StarQuadTree(const StarQuadTree &) = delete;
6     StarQuadTree(StarQuadTree &&) noexcept = default;
7     StarQuadTree &operator=(const StarQuadTree &) = delete;
8     StarQuadTree &operator=(StarQuadTree &&) noexcept = default;
9     // Constructor, which also builds the tree
10    StarQuadTree(const std::vector<Eigen::Vector2d> &starpos,
11                const std::vector<double> &starmasses);
12    virtual ~StarQuadTree() = default;
13
14  protected:
15    struct StarQuadTreeNode {
16      // Constructor: does the recursive construction of the quadtree
17      StarQuadTreeNode(std::vector<unsigned int> star_idx, Eigen::Matrix2d bbox,
18                      StarQuadTree &tree);
19      virtual ~StarQuadTreeNode() = default;
20      // In this code: leaf nodes hold only a single star!
21      [[nodiscard]] inline bool isLeaf() const {
22        return this->star_idx.size() == 1;
23      }
24      std::vector<unsigned int> star_idx_; // Indices of stars in sub-cluster
25      Eigen::Matrix2d bbox_; // Bounding box of sub-cluster
26      Eigen::Vector2d center; // Center of gravity of sub-cluster
27      double mass; // Total mass of stars in
28      // sub-cluster
29      // Pointers to children nodes
30      std::array<std::unique_ptr<StarQuadTreeNode>, 4> sons_{nullptr};
31    };
32  public:
33    const int n; // Total number of stars
```

```

34  std::unique_ptr<StarQuadTreeNode> root_;           // Root node
35  const std::vector<Eigen::Vector2d> starpos_;     // "Point star" positions
36  const std::vector<double> starmasses_;         // Star masses
37  unsigned int no_leaves_{0};                    // number of leaves
38  unsigned int no_clusters_{0};                  // number of genuine star clusters
39
40  // Printing the tree (useful only for small trees)
41  void outputQuadTree(std::ostream &o) const;
42  };

```

The following convention may be adopted for characterizing a 2D bounding box by the numbers stored in the 2×2 -matrix `StarQuadTree::bbox_`: the first column gives the coordinates of the lower-left corner, the second column those of the upper-right corner.

Remark. The contents of a `std::unique_ptr` object are destroyed and the memory is freed when the object reaches the end of its lifetime. So when an `StarQuadTreeNode` object ceases to exist, the `std::unique_ptr` objects stored in its `sons_` data member are destroyed, which triggers calls to the destructors of the `StarQuadTreeNode` objects they point to.

(2-1.c) 📄 (60 min.) Taking for granted that all stars are located in $[0, 1]^2$, implement the constructor of `StarQuadTree::StarQuadTreeNode` so that an object of type `StarQuadTree` represents a full quadtree for the star cluster, when the constructors have been executed. Leaves of the quadtree should all hold one star.

SOLUTION for (2-1.c) → [2-1-3-0:gfs4.pdf](#) ▲

The class `StarQuadTreeClustering` defined in Code 2.1.7 implements the efficient computation of gravitational forces as discussed in [Lecture → § 2.1.2.8].

C++ code 2.1.7: Class for efficient quadtree-based computation of gravitational forces.
 Get it on [GitLab](#) (`gravitationalforces.cpp`).

```

2  class StarQuadTreeClustering : public StarQuadTree {
3  public:
4      StarQuadTreeClustering() = delete;
5      StarQuadTreeClustering(const StarQuadTreeClustering &) = delete;
6      StarQuadTreeClustering(StarQuadTreeClustering &&) noexcept = default;
7      StarQuadTreeClustering &operator=(const StarQuadTreeClustering &) = delete;
8      StarQuadTreeClustering &operator=(StarQuadTreeClustering &&) noexcept =
9          default;
10     // The only non-trivial constructor
11     StarQuadTreeClustering(const std::vector<Eigen::Vector2d> &starpos,
12                           const std::vector<double> &starmasses);
13     virtual ~StarQuadTreeClustering() = default;
14
15     // Decide the admissibility of a sub-cluster with respect to a point
16     [[nodiscard]] bool isAdmissible(const StarQuadTreeNode &node,
17                                   Eigen::Vector2d p, double eta) const;
18     // Compute total gravitational force on a single star with index j
19     [[nodiscard]] Eigen::Vector2d forceOnStar(unsigned int j, double eta) const;
20 };

```

(2-1.d) 📄 (60 min.) In analogy to [Lecture → Eq. (2.1.2.9)] a sub-cluster with bounding box $B \subset \mathbb{R}^2$ is called **admissible** with respect to a point $p \in \mathbb{R}^2$, if

$$\text{dist}(B; p) := \inf\{\|y - p\| : y \in B\} \geq \eta \cdot \text{diam}(B) := \inf\{\|y - z\| : y, z \in B\}, \quad (2.1.8)$$

where $\eta > 0$ is a given parameter, or if the sub-cluster is a leaf of the quadtree.

Code the member function

```
bool StarQuadTreeClustering::isAdmissible(
    const StarQuadTreeNode &node,
    Eigen::Vector2d p, double eta) const;
```

such that it complies with (2.1.8). The `node` argument encodes a sub-cluster, `p` passes the point p , and `eta` the control parameter $\eta > 0$.

SOLUTION for (2-1.d) → [2-1-4-0:gfs4.pdf](#) ▲

(2-1.e) 🕒 (60 min.) In the file `gravitationalforces.cpp` complete the code of the member function

```
Eigen::Vector2d forceOnStar(unsigned int j, double eta) const;
```

for the efficient approximate computation of the force vector $f^j \in \mathbb{R}^2$ based on the quadtree and the admissibility condition (2.1.8) (with control parameter `eta`):

$$f^j \approx \tilde{f}^j := \frac{m_j}{4\pi} \sum_{C \in \mathcal{A}} m_C \frac{c^C - x^j}{\|x^j - c^C\|}, \quad (2.1.10)$$

where \mathcal{A} is the set of all admissible sub-clusters with respect to x^j , m_C is the total mass of the stars in sub-cluster C , and $c^C \in \mathbb{R}^2$ its center of gravity.

SOLUTION for (2-1.e) → [2-1-5-0:gfs5.pdf](#) ▲

(2-1.f) 🕒 (30 min.) In the file `gravitationalforces.cpp` write the C++ function

```
std::vector<double> forceError(
    const StarQuadTreeClustering &qt,
    const std::vector<double> &etas);
```

that computes the error measure $\text{err}(\eta) := \max_j \|\tilde{f}^j - f^j\|$, where f^j/\tilde{f}^j is the exact/approximate force acting on the j -th star. The argument `etas` passes the values of the admissibility control parameter η , for which the error should be computed. Tabulate this error as a function of $\eta \in \{1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0\}$ for $n = 1000$ stars whose position are set by `initStarPositions()`. To that end extend `main()` in `gravitationalforces_main.cpp`.

SOLUTION for (2-1.f) → [2-1-6-0:gfs6.pdf](#) ▲

(2-1.g) 🕒 (30 min.) Measure and compare the runtimes of

- `computeForces_direct()` and
- `StarQuadTreeClustering::forceOnStar()` called for $j = 1, \dots, n$

for $n = 20, 40, 80, 160, 320, 640, 1280, 2560$ (generating star clusters by calling `initStarPositions()`). Do this by means of a function

```
std::pair<double, double> measureRuntimes(unsigned int n);
```

that returns the runtimes for 1. and 2. in milliseconds. Use the admissibility parameter $\eta = 0.5$.

Create a doubly logarithmic plot of the runtimes as a function of the number n of stars.

Of course, runtimes have to be measured by taking the minimum over several (5) runs.

HIDDEN HINT 1 for (2-1.g) → [2-1-7-0:gfh7.pdf](#)

SOLUTION for (2-1.g) → [2-1-7-1:gfs7.pdf](#)



End Problem 2-1 , 290 min.

Problem 2-2: Imbalanced cluster tree for 1D clustering

In this problem we study the consequences of giving up balanced nodal index sets for a binary cluster tree, cf. the discussion in [Lecture → § 2.3.2.20] and [Lecture → Ex. 2.3.2.29].


As in [Lecture → Ex. 2.3.2.28] and [Lecture → Ex. 2.3.2.29] we consider the setting of a clustering technique for the compression of a kernel collocation matrix in 1D with collocation points $\xi_i := 2^{-i+1}$, $i = 1, \dots, n$.

We recursively build a *binary* cluster tree employing the following splitting rule: the index sets of the sons of a non-leaf cluster w are defined by


$$\mathcal{I}(\text{1st son of } w) = \{i \in \mathcal{I}(w) : \xi_i \leq \gamma_w\}, \quad (2.2.1)$$

$$\mathcal{I}(\text{2nd son of } w) = \{i \in \mathcal{I}(w) : \xi_i > \gamma_w\}, \quad (2.2.2)$$

where $\gamma_w := \frac{1}{2}(\max\{\xi_i\}_{i \in \mathcal{I}(w)} + \min\{\xi_i\}_{i \in \mathcal{I}(w)})$ is the “midpoint” of the cluster w .

(2-2.a)  Give a visual rendering of the resulting cluster tree for $n = 6$ as in [Lecture → Ex. 2.3.2.17].

SOLUTION for (2-2.a) → [2-2-1-0:ubt1.pdf](#) ▲

(2-2.b)  Consider the application of the algorithm of **buildRec()** from [Lecture → Code 2.3.3.5] to the cluster tree of Sub-problem (2-2.a) with the geometric admissibility condition

$$\text{adm}(v, w) = \text{true} \Leftrightarrow v, w \text{ not leaves and } \eta(\text{box}(v), \text{box}(w)) \leq \frac{1}{2}.$$

Here η is the admissibility measure from [Lecture → Eq. (2.2.2.7)] and $\text{box}(v)$ is the bounding box of a cluster as defined in [Lecture → Def. 2.3.2.19].

Assume the same cluster tree of Sub-problem (2-2.a) for both x - and y -dimensions. Determine the number of far-field cluster pairs created by the recursion as a function of n .

SOLUTION for (2-2.b) → [2-2-2-0:sp2ubt.pdf](#) ▲

End Problem 2-2

Problem 2-3: Tensor-Product Chebychev Interpolation

In [Lecture → Section 2.2.1.3] we learned about local low-rank matrix compression based on bi-directional kernel interpolation. The most widely used interpolation scheme is tensor-product Chebychev interpolation as explained in [Lecture → Rem. 2.3.4.14]. In this problem we examine its implementation with a focus on the computation of the matrix factors \mathbf{U}_v , $\mathbf{C}_{v \times w}$, and \mathbf{V}_w introduced in [Lecture → Eq. (2.3.4.8)].

This problem involves from-scratch C++ code development based on EIGEN.

▷ problem name/problem code folder: TensorProductChebIntp in [Code repository](#)

To begin with we consider one-dimensional Lagrangian polynomial interpolation based on Chebychev nodes, see [NumCSE course → Section 5.2] and [NumCSE course → Section 6.2.3]. In the “standard interval” $[-1, 1]$ and for polynomial degree $q - 1$, $q \in \mathbb{N}$, the q **Chebychev nodes** are

$$t_j := \cos\left(\frac{2j-1}{2q} \pi\right), \quad j = 1, \dots, q. \quad (2.3.1)$$

These Chebychev nodes are the zeros of the q -th Chebychev polynomial, which can be represented as

$$T_q(x) := \cos(q \arccos(x)) \quad \text{for } -1 \leq x \leq 1. \quad (2.3.2)$$

In [NumCSE course → Section 5.2.3.1] we saw an efficient algorithm for the evaluation of an interpolating polynomial through given points for many arguments. That algorithm is based on the **barycentric interpolation formula** [NumCSE course → Eq. (5.2.3.3)]:

$$p(x) = \sum_{i=1}^q \frac{\lambda_i}{x - t_i} y_i \cdot \left(\sum_{i=1}^q \frac{\lambda_i}{x - t_i} \right)^{-1}, \quad x \neq t_\ell, \quad [\text{Lecture} \rightarrow \text{Eq. (2.3.4.16)}]$$

with weights
$$\lambda_i = \frac{1}{(t_i - t_1) \dots (t_i - t_{i-1})(t_i - t_{i+1}) \dots (t_i - t_q)}, \quad i = 1, \dots, q. \quad [\text{Lecture} \rightarrow \text{Eq. (2.3.4.17)}]$$

(2-3.a)  (15 min.) Write

$$N_q(x) := \prod_{j=1}^q (x - t_j), \quad x \in \mathbb{R}, \quad (2.3.3)$$

for the so-called **nodal polynomial** belonging to the set $\{t_j\}$ of interpolation nodes. Show that the barycentric weights λ_i can be computed as

$$\lambda_i = \frac{1}{N_q'(t_i)}, \quad N_q'(x) := \frac{dN_q}{dx}(x), \quad x \in \mathbb{R}. \quad (2.3.4)$$

SOLUTION for (2-3.a) → 2-3-1-0: .pdf Note that $N_q'(t_i) \neq 0$, because t_i is a simple zero of N_q .

▲

(2-3.b)  (20 min.) [depends on Sub-problem (2-3.a)]

Show that for Chebychev nodes t_j as given in 2.3.1 the barycentric weights λ_i can be computed as

$$\lambda_i = \frac{2^{q-1}}{q} (-1)^{i-1} \sin\left(\frac{2i-1}{2q} \pi\right), \quad i = 1, \dots, q. \quad (2.3.7)$$

HIDDEN HINT 1 for (2-3.b) → [2-3-2-0:tpchh2.pdf](#)

SOLUTION for (2-3.b) → [2-3-2-1:tpchs2.pdf](#) ▲

(2-3.c) 🕒 (30 min.) [depends on Sub-problem (2-3.b)]

In the file `tensorproductchebintp.h` implement the templated C++ function

```
template <typename FUNCTOR>
std::vector<double> chebInterpEval1D (
    unsigned int q, FUNCTOR f,
    std::vector<double> &x);
```

that evaluates the Chebychev interpolant p of degree $q - 1$ on $[-1, 1]$ of a function $f : [-1, 1] \rightarrow \mathbb{R}$ (passed through the functor argument `f`) for arguments x^i (contained in the sequence `x`), and returns the sequence $(p(x^i))_i$.

SOLUTION for (2-3.c) → [2-3-3-0:.pdf](#) ▲

(2-3.d) 🕒 (60 min.) [depends on Sub-problem (2-3.c)]

In the file `tensorproductchebintp.h` implement a templated function

```
template <typename FUNCTOR>
std::vector<double> chebInterpEval2D (
    unsigned int q, FUNCTOR f,
    std::vector<Eigen::Vector2> &x);
```

that extends `chebInterpEval1D()` to two-dimensional Chebychev interpolation on $[-1, 1]^2$ into the space of bi-variate polynomial of degree $q - 1$ in each direction. Of course, now `f` describes a function $f : [-1, 1]^2 \rightarrow \mathbb{R}$, whereas `x` is a sequence of coordinate vectors (x^i) of points in \mathbb{R}^2 . The function is to return the sequence of values of the interpolant in those points.

SOLUTION for (2-3.d) → [2-3-4-0:tpcs4.pdf](#) ▲

(2-3.e) 🕒 (30 min.) [depends on Sub-problem (2-3.d)]

Based on `ChebInterpEval2D()` write a C++ function

```
template <typename FUNCTOR>
std::vector<double> genChebInterpEval2D (
    unsigned int q, FUNCTOR f,
    Eigen::Vector2d a, Eigen::Vector2d b,
    std::vector<Eigen::Vector2d> &x);
```

that performs 2D Chebychev polynomial interpolation of degree $q - 1$ on the box $[a_1, b_1] \times [a_2, b_2] \subset \mathbb{R}^2$, $a_i < b_i$, and evaluates the interpolant on in all the points whose coordinate vectors are passed in `x`. The information on the box is supplied by the two vectors `a` and `b`.

SOLUTION for (2-3.e) → [2-3-5-0:tpcs5.pdf](#) ▲

(2-3.f) 🕒 (120 min.) A coding challenge: Implement a templated C++ function

```
template <int DIM, typename FUNCTOR>
std::vector<double> ChebInterpEvaldD (
    unsigned int q, FUNCTOR f,
    std::vector<Eigen::Matrix<double, DIM, 1>> &x);
```

that can evaluate a degree- $q - 1$ d -dimensional polynomial tensor-product Chebychev interpolant p of $f : [-1, 1]^d \rightarrow \mathbb{R}$ in points with coordinates $x^i \in \mathbb{R}^d$ passed through x . The dimension $d \in \mathbb{N}$ is given as the template parameter **DIM**. The sequence $(p(x^i))_i$ is returned.

SOLUTION for (2-3.f) → [2-3-6-0:tpcis6.pdf](#) ▲

End Problem 2-3, 275 min.

Problem 2-4: Local Low-Rank Approximation by Clustering

In [Lecture → Section 2.3] we learned about the fundamental algorithm for the approximation of kernel collocation matrices with asymptotically smooth singular kernels. The key components of this algorithm are cluster trees [Lecture → Section 2.3.2], a near-field ↔ far-field splitting based on a geometric admissibility condition. In this problem we examine suitable data structures and algorithms for the one-dimensional case, that is, for collocation points located in a real interval.

This problem entails considerable coding in C++ based on EIGEN and a C++ 1D clustering library.
 ▷ problem name/problem code folder: KernMatLLRApprox in [Code repository](#)

In this problem we extend the codes in the `HMAT/CLUSTERING` directory in the course's [GitLab code repository](#) in order to realize a local low-rank approximation $\tilde{M} \in \mathbb{R}^{n,n}$ for a kernel collocation matrix [Lecture → Def. 2.1.3.1]

$$\mathbf{M} := [G(\xi_i, \xi_j)]_{i,j=1,\dots,n} \in \mathbb{R}^{n,n}, \quad n \in \mathbb{N}, \quad (2.4.1)$$

for given collocation points $\xi_i \in \mathbb{R}, i = 1, \dots, n$. $G : \mathbb{R} \times \mathbb{R} \setminus \{x = y\} \rightarrow \mathbb{R}$ is a given, possibly singular, but asymptotically smooth kernel function [Lecture → Rem. 2.2.2.1].

In the file `clustertree.h` you find the definitions of templated C++ classes meant to represent a cluster tree [Lecture → Def. 2.3.2.14] of points. The file `matrixpartition.h` describes tree-based data types representing the partitioning of a kernel collocation matrix into near-field and far-field sub-matrices (blocks) guided by a geometric admissibility condition.

(2-4.a) ☐ (20 min.) Study the classes defined in [Lecture → Code 2.3.2.21], [Lecture → Code 2.3.2.22], [Lecture → Code 2.3.2.23] and the member functions listed in [Lecture → Code 2.3.2.27] and [Lecture → Line 6]. ▲

(2-4.b) ☐ (20 min.) In the file `kernmatllrapprox.h` implement a C++ function

```
template <typename NODE>
bool checkClusterTree(const HMAT::ClusterTree<NODE> &T);
```

that verifies that the **ClusterTree** object passed in `T` complies with the definition

Definition [Lecture → Def. 2.3.2.14]. Cluster tree

Let $\mathbb{I} \subset \mathbb{N}$ be an index set, $\mathcal{T} := (\mathcal{V}, r, \mathcal{E})$ a tree, and $\mathcal{I} : \mathcal{V} \rightarrow 2^{\mathbb{I}}$ a mapping that assigns a subset of \mathbb{I} to every node of \mathcal{T} . We call $\mathcal{T}_{\mathbb{I}} := (\mathcal{V}, r, \mathcal{E}, \mathbb{I}, \mathcal{I})$ a **cluster tree** for \mathbb{I} , if

- (i) the subset corresponding to the root is \mathbb{I} : $\mathcal{I}(r) = \mathbb{I}$,
- (ii) the subset associated with each non-leaf node is the union of the subsets of its sons

$$\mathcal{I}(w) = \bigcup \{ \mathcal{I}(v) : v \in \text{sons}(w) \} \quad \forall w \in \mathcal{V}, \text{sons}(w) \neq \emptyset, \quad [\text{Lecture} \rightarrow \text{Eq. (2.3.2.15)}]$$

- (iii) the sets belonging to different sons of a node are disjoint

$$\forall w \in \mathcal{V}: v_1, v_2 \in \text{sons}(w) \Rightarrow \mathcal{I}(v_1) \cap \mathcal{I}(v_2) = \emptyset. \quad [\text{Lecture} \rightarrow \text{Eq. (2.3.2.16)}]$$

and returns **true**, if it does.

The template parameter **NODE** must refer to a type compatible with **HMAT::CtNode**. In particular it must provide a member function **std::vector<std::size_t> I()** **const** that tells the indices associated with a **NODE** object. No ordering of those indices is assumed.

SOLUTION for (2-4.b) → 2-4-2-0:llras2.pdf

C++ code 2.4.2: Implementation of `checkClusterTree()` →GITLAB

```

2 // clang-format off
3 template <typename NODE>
4 bool checkClusterTree(const HMAT::ClusterTree<NODE> &T) {
5     if (T.root == nullptr) return false;
6     bool ok = true; // Flag to be returned
7     // Index set  $\mathbb{I}$  underlying the cluster tree
8     const std::vector<std::size_t> idxset{T.root->I()};
9     // Array for remapping indices to integers from 0 to  $\#\mathbb{I}$ 
10    const std::size_t maxidx = *std::max_element(idxset.begin(), idxset.end());
11    // A bit of a gamble: indices must not be too large compared to
12    // their number
13    assertm((maxidx < 2 * idxset.size()), "Maximal index too large");
14    std::vector<int> remap_idx(maxidx + 1, -1);
15    for (int k = 0; k < idxset.size(); ++k) remap_idx[idxset[k]] = k;
16    std::vector<unsigned int> c(idxset.size(), 0); // Work array
17
18    // Recursive checking
19    std::function<void(const NODE *)> ctcheck_rec =
20        [&](const NODE *node) -> void {
21            if (node == nullptr) return;
22            // Indices present in the parent node are marked in the work array
23            std::fill(c.begin(), c.end(), 0); // Initialize work array with zero
24            const std::vector<std::size_t> idxset_node{node->I()}; // Index set of
25            // node
26            for (std::size_t idx : idxset_node) {
27                if (idx > maxidx) { ok = false; return; } // Index not present in root
28                else {
29                    const int pos = remap_idx[idx];
30                    if (pos < 0) { ok = false; return; } // index not in root index set
31                    c[pos] = 1; // mark index as present in parent node
32                }
33            }
34            // Visit all sons
35            bool has_sons = false; // no sons at all ?
36            for (const NODE *son : node->sons) {
37                if (son != nullptr) {
38                    has_sons = true;
39                    const std::vector<std::size_t> idxset_son{son->I()}; // son's index set
40                    // Increment entries of work array for indices present in a son
41                    for (std::size_t idx : idxset_son) {
42                        if (idx > maxidx) { ok = false; return; } // Index not present in
43                        // root
44                        else {
45                            const int pos = remap_idx[idx];
46                            if (pos < 0) { ok = false; return; } // Index not present in root
47                            c[pos] += 1;
48                        } } }
49                    // The work array may only contain zeros or 2s at this point
50                    if (has_sons) {
51                        for (unsigned int c_ent : c) {
52                            if ((c_ent != 0) and (c_ent != 2)) { ok = false; return; }
53                        }
54                        // Recursive call
55                        if (ok and has_sons) {
56                            for (const NODE *son : node->sons) {
57                                ctcheck_rec(son);
58                                if (!ok) return;
59                            } } }
60                    } } }

```

```

57     };
58     // Launch recursion
59     ctcheck_rec(T.root);
60     return ok;
61 }

```



In case an index stored in the root is very large, the maximally allowed size of the index remapping array will be exceeded and the code will fail.

Remark. If we assume that the indices held by the root of the cluster tree are $0, 1, 2, \dots$, that is, a contiguous sequence of integers, then implementation becomes much simpler and we can dispense with the hash table. This assumption is made in other parts of the code, see Ass. 2.4.4 below. ▲

(2-4.c) ◻ (20 min.) Familiarize yourself with the C++ codes listed in [Lecture → Section 2.3.3]: [Lecture → Code 2.3.3.3], [Lecture → Code 2.3.3.4], [Lecture → Code 2.3.3.5], [Lecture → Code 2.3.3.7]. ▲

(2-4.d) ◻ (30 min.) [depends on Sub-problem (2-4.c)]

In the file `kernmatllraprox.h` write a templated C++ function

```

template <typename NODE>
bool checkMatrixPartition(
const HMAT::BlockPartition<NODE, HMAT::IndexBlock<NODE>,
HMAT::IndexBlock<NODE>> &partmat);

```

that returns true, if the sets of index pairs stored in `partmat.farField` and `partmat.nearField` really form a partition of the product of the index sets for rows and columns of the matrix. The **NODE** must meet the same requirements as the type of the same name in Sub-problem (2-4.b).



You must not take for granted that the indices stored in the root nodes are sequences of consecutive integers starting from 0.

SOLUTION for (2-4.d) → [2-4-4-0:.pdf](#) ▲

We employ local low-rank compression based on **bi-directional interpolation** as explained in [Lecture → § 2.3.4.6]. Remember that on a far-field block defined by two nodes v and w of the involved cluster trees, it relies on the rank- q approximation of the associated block of the kernel collocation matrix

$$\begin{aligned}
 \mathbf{U}_v &:= \left[b_k^v(\mathbf{x}^i) \right]_{\substack{i \in \mathcal{I}(v) \\ k=1, \dots, q}} \in \mathbb{R}^{\#\mathcal{I}(v), q}, \\
 \mathbf{M}|_{v \times w} \approx \widetilde{\mathbf{M}}|_{v \times w} &:= \mathbf{U}_v \mathbf{C}_{v \times w} \mathbf{V}_w^\top, \quad \mathbf{C}_{v \times w} := \left[G(\mathbf{t}_v^k, \mathbf{t}_w^\ell) \right]_{k, \ell=1, \dots, q} \in \mathbb{R}^{q, q}, \\
 \mathbf{V}_w &:= \left[b_\ell^w(\mathbf{y}^j) \right]_{\substack{j \in \mathcal{I}(w) \\ \ell=1, \dots, q}} \in \mathbb{R}^{\#\mathcal{I}(w), q}.
 \end{aligned}$$

[Lecture → Eq. (2.3.4.8)]

In the current setting b_k^v and b_ℓ^w are the Lagrange polynomials for **Chebyshev interpolation** on the bounding boxes of v and w , respectively, while \mathbf{t}_v^k and \mathbf{t}_w^ℓ designate the interpolation nodes, the Chebyshev nodes for the intervals associated with v and w .

We make a few simplifying assumptions valid for the remainder of the problem:

Assumption 2.4.4. Setting for clustering

- The collocation nodes are the same in x - and y -direction.
- We consider only one spatial dimension: the collocation nodes $\xi_i, i = 1, \dots, n$, are located in a bounded interval $\subset \mathbb{R}$.
- The cluster trees for both directions are the same: $\mathcal{T}_I = \mathcal{T}_J$.
- The index set belonging to the root of the underlying cluster tree is contiguous $\{1, \dots, n\}$ (or $0, \dots, n-1$ for C++ indexing).

(2-4.e) ☒ Every cluster ($\hat{=}$ node of the cluster tree) w contributes the matrix

$$\mathbf{V}_w := [b_\ell^w(\xi_j)]_{\substack{j \in \mathcal{I}(w) \\ \ell=1, \dots, q}} \in \mathbb{R}^{\#\mathcal{I}(w), q}$$

to the local rank- q representation of every far-field block containing the cluster w . This is captured by the following specialization of the data type of nodes of the cluster tree.

C++ code 2.4.5: Special class for nodes of a cluster tree supporting the bi-directional interpolation approach →GITLAB

```

2  template <int DIM>
3  class InterpNode : public HMAT::CtNode<DIM> {
4  public:
5      // Constructor from sequence of points; initializes V
6      InterpNode(const std::vector<HMAT::Point<DIM>> _pts, std::size_t _q,
7                 int _dir = 0)
8          : HMAT::CtNode<DIM>(_pts, _dir), q(_q), sons{nullptr, nullptr},
9            clust_omega(_q), V(_pts.size(), _q) { initV(); }
10     virtual ~InterpNode() = default;
11     // Is the node a leaf node ?
12     [[nodiscard]] bool isLeaf() const override {
13         return !(sons[0]) and !(sons[1]);
14     }
15     protected:
16     // Initialization of the low-rank factor matrix V_w
17     void initV();
18     public:
19     const std::size_t q; // Rank, no of interpolation nodes
20     Eigen::MatrixXd V; // low-rank factor V ∈ ℝ^{k,q}
21     std::array<InterpNode *, 2> sons; // Pointers to sons (of type
22         InterpNode!)
23     Eigen::VectorXd clust_omega; // for cluster-local linear algebra
24 };

```

Implement the member function `initV()` meant to initialize the data member `V` storing the matrix \mathbf{V}_w for the cluster.

HIDDEN HINT 1 for (2-4.e) → 2-4-5-0:kmlh5.pdf

SOLUTION for (2-4.e) → 2-4-5-1:.pdf ▲

(2-4.f) ☒ (20 min.) The following listing defines a class **BiDirChebInterpBlock**, which is a suitable data type for the **FFB** template argument of **BlockPartition** from [Lecture → Code 2.3.3.3].

C++ code 2.4.8: Data type for far-field block in the case of bi-directional Chebychev interpolation. →GITLAB

```

2  template <class NODE, typename KERNEL>
3  class BiDirChebInterpBlock : public HMAT::IndexBlock<NODE> {

```

```

4 public:
5 using kernel_t = KERNEL;
6 BiDirChebInterpBlock(NODE &_nx, NODE &_ny, KERNEL _Gfun, std::size_t _q);
7 // Constructor that should not be called, needed to avoid
8 // compilation errors
9 BiDirChebInterpBlock(NODE &_nx, NODE &_ny)
10 : HMAT::IndexBlock<NODE>(_nx, _ny), q(0) {
11     throw std::runtime_error("Invalid constructor");
12 }
13 virtual ~BiDirChebInterpBlock() = default;
14
15 KERNEL G; // kernel function G
16 const int q; // No of interpolation nodes
17 Eigen::MatrixXd C; //  $C \in \mathbb{R}^{q,q}$ , see [Lecture → Eq. (2.3.4.8)]
};

```

Here the template argument **KERNEL** is a functor of type `std::function<double(double, double)>` providing the kernel function G . The type **Node** must fit the class **CtNode** shown in [Lecture → Code 2.3.2.22].

Implement the constructor of **BiDirChebInterpBlock** such that it initializes the data members in a way compatible with local low-rank compression based on q -point bi-directional Chebychev interpolation. Ass. 2.4.4 applies.

SOLUTION for (2-4.f) → [2-4-6-0:kml1rs6.pdf](#) ▲

(2-4.g) 🕒 (20 min.) The following class is meant to represent a near-field block in a partitioned matrix.

C++ code 2.4.12: Data type for near-field block → [GITLAB](#)

```

2 template <class NODE, typename KERNEL>
3 class NearFieldBlock : public HMAT::IndexBlock<NODE> {
4 public:
5 using kernel_t = KERNEL;
6 NearFieldBlock(NODE &nx, NODE &ny, KERNEL _Gfun);
7 // Constructor that should not be called, needed to avoid
8 // compilation errors
9 NearFieldBlock(NODE &_nx, NODE &_ny) : HMAT::IndexBlock<NODE>(_nx, _ny) {
10     throw std::runtime_error("Invalid constructor");
11 }
12 virtual ~NearFieldBlock() = default;
13
14 KERNEL G; // kernel function G
15 Eigen::MatrixXd Mloc; // local kernel collocation matrix
16 };

```

As in the case of **BiDirChebInterpBlock** the template argument **KERNEL** is a functor of type `std::function<double(double, double)>` providing the kernel function G . Also the type **Node** must fit the class **CtNode** shown in [Lecture → Code 2.3.2.22].

Complete the implementation of the constructor, which is supposed to initialize the data members, in particular the matrix `Mloc`.

SOLUTION for (2-4.g) → [2-4-7-0:.pdf](#) ▲

The partitioned matrices arising from local low-rank compression are represented by objects of the following type, derived from the class **BlockPartition** listed in [Lecture → Code 2.3.3.3]. Remember that

a **BlockPartition** object contains data members `farfield` and `nearField` that are sequences of objects of type **FFB** and **NFB**, respectively.

C++ code 2.4.15: Class for partitioned matrix with near-field/far-field splitting →GITLAB

```

2  template <class NODE, typename FFB, typename NFB>
3  class BiDirChebBlockPartition : public HMAT::BlockPartition<NODE, FFB, NFB> {
4  public:
5      using kernel_t = typename NFB::kernel_t;
6      BiDirChebBlockPartition (std::shared_ptr<LLRClusterTree<NODE>> _rowT,
7                              std::shared_ptr<LLRClusterTree<NODE>> _colT,
8                              kernel_t _Gfun, std::size_t _q, double eta0 = 2.0)
9          : HMAT::BlockPartition<NODE, FFB, NFB>(_rowT, _colT), G(_Gfun), q(_q) {
10         HMAT::BlockPartition<NODE, FFB, NFB>::init(eta0);
11     }
12     virtual ~BiDirChebBlockPartition () = default;
13
14 protected:
15     // Construct an instance of far-field block type
16     virtual FFB makeFarFieldBlock(NODE &nx, NODE &ny) {
17         ffb_cnt++;
18         return FFB(nx, ny, G, q);
19     }
20     // Construct an instance of near-field block type
21     virtual NFB makeNearFieldBlock(NODE &nx, NODE &ny) {
22         nfb_cnt++;
23         return NFB(nx, ny, G);
24     }
25
26 public:
27     kernel_t G; // Reference to the kernel function
28     const std::size_t q; // degree+1 of interpolating polynomial
29     unsigned int ffb_cnt{0}; // Counter for far-field blocks
30     unsigned int nfb_cnt{0}; // Counter for near-field blocks
31 };

```

The particular local low-rank compression based on q -point bi-directional Chebychev interpolation can then be encapsulated by the type

```

template <typename KERNEL>
using BiDirChebPartMat1D =
    BiDirChebBlockPartition<InterpNode<1>,
        BiDirChebInterpBlock<HMAT::CtNode<1>, KERNEL>,
        NearFieldBlock<HMAT::CtNode<1>, KERNEL>>;

```

which, then, can be used as follows

C++ code 2.4.16: Creating an object of type BiDirChebPartMat1D →GITLAB

```

2  const int q = 5; // Degree of Chebychev interpolation
3  const int npts = 16; // Number of collocation points
4  std::vector<HMAT::Point<1>> pts;
5  for (int n = 0; n < npts; n++) {
6      HMAT::Point<1> p;
7      p.idx = n;
8      p.x[0] = static_cast<double>(n) / (npts - 1);
9      pts.push_back(p);
10 }
11 // Allocate cluster tree object (the same for both directions)

```

```

12 auto T_row = std::make_shared<
13     KernMatLLRAprox::LLRClusterTree<KernMatLLRAprox::InterpNode<1>>>(q);
14 T_row->init(pts);
15 auto T_col = std::make_shared<
16     KernMatLLRAprox::LLRClusterTree<KernMatLLRAprox::InterpNode<1>>>(q);
17 T_col->init(pts);
18 // Kernel for 1D collocation
19 struct Kernel {
20     Kernel() = default;
21     double operator()(double x, double y) const {
22         return ((x != y) ? -std::log(std::abs(x - y)) : 0.0);
23     }
24 } G;
25 KernMatLLRAprox::BiDirChebPartMat1D<Kernel> Mt(T_row, T_col, G, q, 2.0);

```

Storing data in clusters

We have to pass two copies of the same cluster tree to the constructor of **BiDirChebPartMat1D**, because the “reference implementation” in this homework project stores temporary data directly in clusters. This also prevents us to pass the cluster trees as **const** objects.

Remark 2.4.18 (Index-based management of cluster-local data) A probably better design of the code would endow the clusters of the cluster trees with contiguous integer indices starting from zero. Those could be used to reference cluster-local data kept elsewhere. A similar indexing scheme could be applied to the leaves of the block (cluster) tree as well in order to manage data belonging to matrix blocks. ┘

(2-4.h) 🕒 (60 min.) In the file `kernmatllraprox.h` complete the code of the C++ function

```

template <typename NODE, typename FFB, typename NFB>
unsigned int computeSparsityMeasure (
    const HMAT::BlockPartition<NODE, FFB, NFB> &blockpart);

```

that computes the **sparsity measure** of a matrix block partitioning encoded in the `blockpart` object.

Definition [Lecture → Def. 2.3.4.27]. Sparsity measure of block partition

Let $\mathbb{F} := \{I_k \times J_k\}_k$ be a block partition of $\mathbb{D} := \mathbb{I} \times \mathbb{J}$ based on the cluster trees $\mathcal{T}_{\mathbb{I}}$ and $\mathcal{T}_{\mathbb{J}}$. Then the **sparsity measure** of \mathbb{F} bounds the number of occurrences of a cluster in cluster pairs

$$\text{spm}(\mathbb{F}) := \max \left\{ \max_{v \in \mathcal{T}_{\mathbb{I}}} \#\{w \in \mathcal{T}_{\mathbb{J}} : (v, w) \in \mathbb{F}\}, \max_{w \in \mathcal{T}_{\mathbb{J}}} \#\{v \in \mathcal{T}_{\mathbb{I}} : (v, w) \in \mathbb{F}\} \right\}. \quad ((??))$$

The template argument **NODE** must be a type compatible with **CtNode** from [Lecture → Code 2.3.2.22], while **FFB** and **NFB** describe far-field and near-field blocks and can be assumed to have been derived from **IndexBlock** as shown in [Lecture → Code 2.3.3.4].

SOLUTION for (2-4.h) → [2-4-8-0:.pdf](#)

C++ code 2.4.19: Implementation of `computeSparsityMeasure()` →GITLAB

```

2 template <typename NODE, typename FFB, typename NFB>
3 unsigned int computeSparsityMeasure (
4     const HMAT::BlockPartition<NODE, FFB, NFB> &blockpart,
5     std::ostream *out = nullptr) {
6     assertm((blockpart.rowT and blockpart.colT), "Missing trees!");

```

```

7 // Set up hash maps for nodes of both trees
8 using nf_node_t = typename NFB::node_t;
9 using hashmap_t = std::unordered_map<const nf_node_t *, int>;
10 using keyval_t = typename hashmap_t::value_type;
11 hashmap_t nodemap_row;
12 hashmap_t nodemap_col;
13 // Maximal node counts for row and column clusters
14 int xnode_maxcnt = 0;
15 int ynode_maxcnt = 0;
16 // Run through all far-field blocks and also determine maximal
17 // cluster count
18 for (const auto &ffb : blockpart.farField) {
19     // If the current block is based on a cluster, increment that
20     // cluster's
21     // count in the hash map
22     if (nodemap_row.find(&ffb.nx) == nodemap_row.end()) {
23         nodemap_row[&ffb.nx] = 1;
24         xnode_maxcnt = std::max(xnode_maxcnt, 1);
25     } else {
26         xnode_maxcnt = std::max(xnode_maxcnt, nodemap_row[&ffb.nx] += 1);
27     }
28     if (nodemap_col.find(&ffb.ny) == nodemap_col.end()) {
29         nodemap_col[&ffb.ny] = 1;
30         ynode_maxcnt = std::max(ynode_maxcnt, 1);
31     } else {
32         ynode_maxcnt = std::max(ynode_maxcnt, nodemap_col[&ffb.ny] += 1);
33     }
34 }
35 // Run through all near-field blocks
36 for (const auto &nfb : blockpart.nearField) {
37     // If the current block is based on a cluster, increment that
38     // cluster's
39     // count in the hash map
40     if (nodemap_row.find(&nfb.nx) == nodemap_row.end()) {
41         nodemap_row[&nfb.nx] = 1;
42         xnode_maxcnt = std::max(xnode_maxcnt, 1);
43     } else {
44         xnode_maxcnt = std::max(xnode_maxcnt, nodemap_row[&nfb.nx] += 1);
45     }
46     if (nodemap_col.find(&nfb.ny) == nodemap_col.end()) {
47         nodemap_col[&nfb.ny] = 1;
48         ynode_maxcnt = std::max(ynode_maxcnt, 1);
49     } else {
50         ynode_maxcnt = std::max(ynode_maxcnt, nodemap_col[&nfb.ny] += 1);
51     }
52 }
53 // Print node statistics if requested
54 if (out) {
55     auto output = [] (const hashmap_t &nodemap, std::ostream *out) -> void {
56         if (out) {
57             for (const keyval_t kv : nodemap) {
58                 const nf_node_t *node_p = kv.first;
59                 const int node_count = kv.second;
60                 const std::vector<size_t> idxs{node_p->l()};
61                 (*out) << "Node with idx set = ";
62                 for (size_t idx : idxs) {
63                     (*out) << idx << ' ';
64                 }
65                 (*out) << " occurs " << node_count << " times " << std::endl;
66             }
67         }
68     };
69     output(nodemap, &out);
70 }

```

```

65     }
66     };
67     (*out) << "ROW CLUSTER TREE" << std::endl;
68     output(nodemap_row, out);
69     (*out) << "COLUMN CLUSTER TREE" << std::endl;
70     output(nodemap_col, out);
71     }
72     return std::max(xnode_maxcnt, ynode_maxcnt);
73 }

```

Remark. The index-based design discussed in ?? would lead to a simpler code for `computeSparsityMeasure()`.

(2-4.i) 🕒 (90 min.) In the file `kernmatllrapprox.h` implement an *efficient* C++ function

```

template <class NODE, typename FFB, typename NFB>
    Eigen::VectorXd mvLLRPartMat(
        BiDirChebBlockPartition<NODE, FFB, NFB> &llrcmat, const
        Eigen::VectorXd &x);

```

that returns the result of the multiplication of the matrix M_t with the vector x .

The template argument **PARTMAT1D** must be a type like **BiDirChebPartMat1D<KERNEL>** that represents a block-partitioned matrix created by a clustering algorithm based on bi-directional interpolation.

HIDDEN HINT 1 for (2-4.i) → [2-4-9-0:mvkml1rh1.pdf](#)

SOLUTION for (2-4.i) → [2-4-9-1:kmlrsx.pdf](#)

(2-4.j) 🕒 (45 min.) [depends on Sub-problem (2-4.i)]

Under Ass. 2.4.4 in the file `kernmatllrapprox.h` complete the implementation of the C++ function

```

template <typename KERNEL>
    std::pair<double, double> approxErrorLLR(
        const BiDirChebPartMat1D<KERNEL> &Mt);

```

which computes

1. the scaled Frobenius norm of the error incurred by local low-rank compression, that is, the quantity

$$\text{err} := \left(\frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n (\mathbf{M})_{ij}^2 - (\widetilde{\mathbf{M}})_{ij}^2 \right)^{1/2}, \quad (2.4.21)$$

where $\mathbf{M} \in \mathbb{R}^{n,n}$ is the (exact) kernel collocation matrix, while $\widetilde{\mathbf{M}} \in \mathbb{R}^{n,n}$ is its approximation.

2. the scaled Frobenius norm of the exact kernel collocation matrix:

$$\left(\frac{1}{n^2} \text{norm} := \sum_{i=1}^n \sum_{j=1}^n (\mathbf{M})_{ij}^2 \right)^{1/2}.$$

The two (scaled) norms are returned in that order.

HIDDEN HINT 1 for (2-4.j) → [2-4-10-0:llrhy.pdf](#)

SOLUTION for (2-4.j) → [2-4-10-1:llrsy.pdf](#)

(2-4.k) 🕒 (45 min.) In the file `kernmatllrapprox.cpp` realize a C++ function

```
bool validateLLR(unsigned int q);
```

that performs a validation of the codes underlying **BirDirChebPartMat1D** for bi-directional Chebychev interpolation relying on q points.



Idea: If the kernel function $(x, y) \mapsto G(x, y)$ is a polynomial of degree $\leq q - 1$ in both variables, then compression does not introduce any error: $\mathbf{M} = \widetilde{\mathbf{M}}$!

Hence we check whether the compression error = 0 for the kernel functions

$$G(x, y) = x^s y^t, \quad 0 \leq s, t \leq q - 1.$$

For the test use $n = 64$ equispaced collocation points $\xi_i = \frac{i}{63}, i = 0, \dots, 63$,

HIDDEN HINT 1 for (2-4.k) → [2-4-11-0:llcrzh.pdf](#)

SOLUTION for (2-4.k) → [2-4-11-1:llrzs.pdf](#) ▲

(2-4.l) 🕒 (30 min.) [depends on Sub-problem (2-4.j)]

In the file `kernmatllrapprox.cpp` write a function

```
void tabulateConvergenceLLR(std::vector<unsigned int> &&n_vec,  
                             std::vector<unsigned int> &&q_vec,  
                             double eta = 2.0);
```

that tabulates the *relative* compression error $\|\mathbf{M} - \widetilde{\mathbf{M}}\|_F / \|\mathbf{M}\|_F$ (in the Frobenius matrix norm) for the asymptotically smooth **logarithmic kernel function**

$$G(x, y) = -\log|x - y|, \quad x, y \in \mathbb{R}, x \neq y, \quad (2.4.24)$$

for $n \geq 2$ equispaced collocation points $\xi_i = \frac{i}{n-1}, i = 0, \dots, n-1$, and bi-directional Chebychev interpolation with q points in every direction. Sequences of values for n and q are passed as arguments. Use $\eta = 2.0$ in the geometric admissibility condition.

Compute the compression error for $n \in \{10, 20, 40, 60, 80, 160, 320, 640, 1280\}$, $q \in \{3, 4, 5, 6, 7\}$ and deduce its asymptotic convergence as $q \rightarrow \infty$.

SOLUTION for (2-4.l) → [2-4-12-0:llrqs.pdf](#) ▲

(2-4.m) 🕒 (30 min.) [depends on Sub-problem (2-4.i)]

Measure the runtime of forming $\mathbf{M}\vec{\mu}$, and $\widetilde{\mathbf{M}}\vec{\mu}$ for equispaced collocation points $\xi_i = \frac{i}{n-1}, n = 2^\ell, \ell = 10, 11, \dots, 18$, and $\vec{\mu} = \mathbf{1}$ (vector with all entries = 1). Do this within a function

```
void runtimeMatVec(  
    std::vector<unsigned int> &&n_vec, unsigned int n_runs = 3,  
    unsigned int q = 5, double eta = 2.0);
```

for equispaced collocation points $\xi_i = \frac{i}{n-1}, i = 0, \dots, n-1$, bi-directional Chebychev interpolation with $q = 4$ points in every direction, and admissibility parameter $\eta = 0.5$. Tabulate the measured runtimes, more precisely the minimal runtime measured for three successive calls of `mvLLRPartMat()` for $\widetilde{\mathbf{M}}\vec{\mu}$.

Discuss the behavior of the runtime as a function of n for large n .

HIDDEN HINT 1 for (2-4.m) → [2-4-13-0:lrrfh7.pdf](#)

SOLUTION for (2-4.m) → [2-4-13-1:llrsr.pdf](#) ▲

End Problem 2-4, 430 min.

Problem 2-5: Evaluation of Trigonometric Polynomials

In [NumCSE course → Section 4.2] we learned about the **discrete Fourier transform** (DFT), which is closely linked to the evaluation of a trigonometric polynomial at equidistant points. Since the DFT can be performed with almost (up to a logarithmic factor) linear complexity by means of the FFT algorithm [NumCSE course → Section 4.3], this special evaluation can be carried out very fast. What about evaluation in general points? This problem will present an efficient algorithm for the *approximate* evaluation based on local low-rank compression of a special kernel collocation matrix [DR95].

Implementation will rely on the C++ clustering codes of [Lecture → Section 2.3]. This problem partly relies on the code developed in Problem 2-4. It is expected that this problem has already been studied and solved.

Templates: [Get it on 🍷 GitLab](#).

Master solution: [Get it on 🍷 GitLab](#).

▷ problem name/problem code folder: EvalTrigPoly in [Code repository](#)

A trigonometric polynomial can be regarded as the restriction of a regular polynomial to the unit circle in the complex plane [NumCSE course → Def. 5.6.1.1]. For given $n \in \mathbb{N}$ consider a polynomial of degree $n - 1$ with complex coefficients,

$$q(z) := \sum_{j=0}^{n-1} \gamma_j z^j, \quad \gamma_j \in \mathbb{C}, \quad z \in \mathbb{C}, \quad (2.5.1)$$

and restrict it to the unit circle parameterized by the complex exponential,

$$\{z \in \mathbb{C} : |z| = 1\} = \{\exp(-2\pi i x) : x \in [0, 1[\}. \quad (2.5.2)$$

This yields the trigonometric polynomial

$$p(x) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x), \quad x \in [0, 1[. \quad (2.5.3)$$

In this problem we define the vector space of trigonometric polynomials of degree $n - 1$ as

$$\mathcal{P}_n^T := \{x \in \mathbb{R} \mapsto \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x) : \gamma_j \in \mathbb{C}\}. \quad (2.5.4)$$

Note that this is different from the definition used in [NumCSE course → Section 6.5].

(2-5.a) 🕒 (20 min.) Refresh yourself on the **discrete Fourier transform** (DFT) [NumCSE course → Def. 4.2.1.18], its inverse, and how to use EIGEN's DFT support [NumCSE course → § 4.2.1.21].



(2-5.b) 🕒 [depends on Sub-problem (2-5.a)]

In the file `evaltrigpoly.cpp` realize an *efficient* function

```
Eigen::VectorXcd evalTrigPolyEquid(const Eigen::VectorXcd &coeffs);
```

which returns the vector

$$[p(\ell/n)]_{\ell=0}^{n-1} \in \mathbb{C}^n,$$

where p is the trigonometric polynomial defined in (2.5.3) and `coeff` passes the n coefficients κ_j , $j = 0, \dots, n - 1$.

HIDDEN HINT 1 for (2-5.b) → [2-5-2-0:etph2.pdf](#)

SOLUTION for (2-5.b) → [2-5-2-1:.pdf](#) ▲

(2-5.c) ☒ (30 min.) Given $n \in \mathbb{N}$ derive an explicit formula for the **cardinal basis** of \mathcal{P}_n^T with respect to the equispaced interpolation nodes $x_\ell := \frac{\ell}{n}, \ell = 0, \dots, n-1$.

HIDDEN HINT 1 for (2-5.c) → [2-5-3-0:etph3x.pdf](#)

HIDDEN HINT 2 for (2-5.c) → [2-5-3-1:etph3.pdf](#)

SOLUTION for (2-5.c) → [2-5-3-2:ethph3s.pdf](#) ▲

Now we are given $m \in \mathbb{N}$ and m distinct points $x_\ell \in [0, 1[, \ell = 0, \dots, m-1$. The task is to find an efficient algorithm for approximately computing the m values $p(x_\ell)$, where p is a trigonometric polynomial

$$p(x) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x), \quad x \in [0, 1[, \quad (2.5.3)$$

given through its coefficients $\gamma_j \in \mathbb{C}, j \in \{0, \dots, n-1\}$.

(2-5.d) ☐ (20 min.) In the file `evaltrigpoly.cpp` implement an **efficient** C++ function

```
Eigen::VectorXcd evalTrigPoly(const Eigen::VectorXcd &gamma, const
    Eigen::VectorXd &x);
```

that returns the column vector $[p(x_\ell)]_{\ell=0}^{m-1} \in \mathbb{C}^m$, when given the coefficients $\gamma_0, \dots, \gamma_{n-1}$ of a trigonometric polynomial p in the parameter `gamma` and the points $x_\ell, \ell = 0, \dots, m-1$, in `x`.

SOLUTION for (2-5.d) → [2-5-4-0:etp3as.pdf](#) ▲

(2-5.e) ☒ (30 min.) [depends on Sub-problem (2-5.c)]

Assuming $x_\ell \notin \{k/n : k = 0, \dots, n-1\}$ for all $\ell = 0, \dots, m-1$, find a vector $\vec{\zeta} = [\zeta_\ell]_{\ell=0}^{m-1} \in \mathbb{C}^m$ and a **kernel function** $G : [0, 1[\times [0, 1[\rightarrow \mathbb{C}$ such that

$$[p(x_\ell)]_{\ell=0}^{m-1} = \text{diag}(\vec{\zeta}) \cdot [G(x_\ell, k/n)]_{\substack{\ell=0, \dots, m-1 \\ k=0, \dots, n-1}} \cdot \vec{\varphi}, \quad \vec{\varphi} := [p(k/n)]_{k=0}^{n-1}. \quad (2.5.11)$$

Here $\text{diag}(\vec{\zeta})$ is the diagonal matrix that has the entries of the vector $\vec{\zeta}$ on its diagonal.

SOLUTION for (2-5.e) → [2-5-5-0:.pdf](#) ▲

We want to rely on the clustering techniques introduced in [Lecture → Section 2.3] to realize an efficient approximate evaluation of the matrix \times vector product in (2.5.11).

(2-5.f) ☒ (30 min.) [depends on Sub-problem (2-5.e)]

Assuming geometric binary cluster trees for the sets of collocation points $\{x_\ell\}_{\ell=0}^{m-1}$ and $\{\frac{k}{n}\}_{k=0}^{n-1}$ propose an **admissibility condition** in the spirit of [Lecture → ??] that will render bi-directional Chebychev interpolation on the resulting far-field blocks uniformly exponentially convergent.

HIDDEN HINT 1 for (2-5.f) → [2-5-6-0:etph5.pdf](#)

SOLUTION for (2-5.f) → [2-5-6-1:etpsp5s.pdf](#) ▲

(2-5.g) ☐ (30 min.) [depends on Sub-problem (2-5.f), ??]

From the class **BiDirChebBlockPartition** as listed in Code 2.4.15 (Problem 2-4) derive a class **BlockPartitionPeriodic** with an overloaded `adm()` member function that realizes the admissibility condition

found in Sub-problem (2-5.f). The constructor should remain the same. The code should be added to `evaltrigpoly.h`.

Of course the derived class can only deal with the one-dimensional setting $d = 1$ and this should be checked during compile time.

SOLUTION for (2-5.g) → [2-5-7-0:.pdf](#) ▲

(2-5.h) 🕒 (45 min.) We have seen that the formula (2.5.11) derived in Sub-problem (2-5.e) does not work in case $x_\ell = k/n$ for some $\ell \in \{0, \dots, m-1\}$, $k \in \{0, \dots, n-1\}$.

Develop a criterion when an $x_\ell \approx k/n$ can be replaced with k/n without incurring a relative error in $p(x_\ell)$ larger than some prescribed tolerance $\tau \in]0, 1[$.

Implement your criterion in a C++ function (in the file `evaltrigpoly.cpp`)

```
std::pair<
    std::vector<unsigned int>,
    std::vector<std::complex<double>>>
    checkX(const Eigen::VectorXcd &gamma,
           const Eigen::VectorXd &x, double tau);
```

that returns

1. the indices ℓ of all those x_ℓ s, which are classified a close to a value k/n with respect to the evaluation of p and tolerance τ , and
2. the values $p(k/n)$ for those k s.

Throw an exception, in case $x_\ell \notin [0, 1]$.



The asymptotic computational cost of the execution of `checkX()` must not be worse than $m \log(m)$ for $m \rightarrow \infty$, $m := x.size()$. Your criterion has to be chosen in a way such that this requirement can be satisfied.

SOLUTION for (2-5.h) → [2-5-8-0:.pdf](#) ▲

(2-5.i) 🕒 (90 min.) [depends on Sub-problem (2-5.e), Sub-problem (2-5.b)]

In the file `evaltrigpoly.cpp` implement a C++ function

```
Eigen::VectorXcd evalTrigPolyApprox(
    const Eigen::VectorXcd &gamma,
    const Eigen::VectorXd &x, unsigned int q);
```

The first three arguments have the same format and meaning as those of `evalTrigPoly()` from Sub-problem (2-5.d) and so does the return value. The fourth argument q is the number of points used in both directions for bi-directional polynomial kernel interpolation on far-field blocks.

The function should rely on local low-rank compression based on bi-directional Chebychev interpolation as elaborated in Problem 2-4. Make use of the class `BlockPartitionPeriodic` from Sub-problem (2-5.g). Rely on `checkX()` from Sub-problem (2-5.h) with τ the machine precision to trigger a special treatment of those x_ℓ s close to values k/n .

SOLUTION for (2-5.i) → [2-5-9-0:etp7s.pdf](#) ▲

(2-5.j) 🕒 (30 min.) [depends on Sub-problem (2-5.i)]

In the file `evaltrigpoly.cpp` code a C++ function

```
void tabulateCvgLLREvalTrigPoly(unsigned int n);
```

that tabulates

$$\text{err}(q) := \text{evalTrigPoly}(c, x) - \text{evalTrigPolyApprox}(c, x, q), \quad q \in \{3, 4, 5, 6, 7\}$$

for

- $c[j] = \frac{1}{j^2}, j = 0, \dots, n-1,$
- $x[k] = \sqrt{k/n}, k = 0, \dots, n-1.$

What can you say, qualitatively and quantitatively, about the (empirical) convergence of $q \mapsto \text{err}(q)$?

SOLUTION for (2-5.j) → [2-5-10-0:.pdf](#) ▲

(2-5.k) 📄 (20 min.)

```
void tabulateRuntimesLLREvalTrigPoly(unsigned int q,
std::vector<unsigned int> &nseq);
```

that measures and tabulates the runtime (minimum over five independent runs) of `evalTrigPolyApprox()`

- for the specified number q of interpolation points in each direction,
- for the trigonometric polynomial with coefficients

$$c[j] = \frac{1}{j^2}, j = 0, \dots, n-1,$$

- for the evaluation nodes

$$x[k] = \sqrt{k/n}, k = 0, \dots, n-1,$$

- and different values for the number n of evaluation nodes/degree of the trigonometric polynomial, which are passed through the `nseq` argument.

The table should also include the runtimes of `evalTrigPoly()` from Sub-problem (2-5.d) for the same evaluation.

Report the runtimes for $q = 4$ and $n \in \{100, 500, 2500, 12500, 62500, 312500\}$.

HIDDEN HINT 1 for (2-5.k) → [2-5-11-0:etplrrfh7.pdf](#)

SOLUTION for (2-5.k) → [2-5-11-1:etps9.pdf](#) ▲

References

- [DR95] A. Dutt and V. Rokhlin. “Fast Fourier transforms for non-equispaced data II”. In: *Appl. Comput. Harmon. Anal.* 2 (1995), pp. 85–100 (cit. on p. 25).

End Problem 2-5, 345 min.

Problem 2-6: Merging of low-rank matrices

In this problem you are asked to implement the algorithm outlined in [Lecture → § 2.4.2.25] for the low-rank recompression of matrices created by concatenation of low-rank matrices.

Template: Get it on  [GitLab](#).

Solution: Get it on  [GitLab](#).

Involves implementation in C++ based on EIGEN


▷ problem name/problem code folder: LowRankMerge in [Code repository](#)

Given $n \in \mathbb{N}$, $q \in \mathbb{N}$, and two rank- q matrices in factorized form

$$\mathbf{X}_i = \mathbf{A}_i \cdot \mathbf{B}_i^\top, \quad \mathbf{A}_i \in \mathbb{R}^{n,q}, \quad \mathbf{B}_i \in \mathbb{R}^{n,q}, \quad i = 1, 2, \quad (2.6.1)$$

we seek to compute a rank- q best approximation of $\mathbf{Z} := [\mathbf{X}_1 \ \mathbf{X}_2] \in \mathbb{R}^{n,2n}$, also in factorized form

$$\mathbf{Z} \approx \tilde{\mathbf{Z}} = \mathbf{U} \cdot \mathbf{V}^\top, \quad \mathbf{U}, \mathbf{V} \in \mathbb{R}^{n,q}.$$


(2-6.a)  (20 min.) For $n \in \mathbb{N}$ we consider collocation points $\xi_i := \frac{i}{n}$, $i \in \{1, \dots, n\}$ and the kernel collocation matrices

$$\mathbf{K}_1 = [\sin(\xi_i - \xi_j)]_{i,j=1}^n, \quad \mathbf{K}_2 = [\cos(\xi_i - \xi_j - \frac{1}{2n})]_{i,j=1}^n.$$

Determine the rank of \mathbf{K}_1 and \mathbf{K}_2 and their low-rank factorized form.

HIDDEN HINT 1 for (2-6.a) → [2-6-1-0:1h.pdf](#)


SOLUTION for (2-6.a) → [2-6-1-1:lrm1.pdf](#) ▲

(2-6.b)  (90 min.) Using EIGEN, in the file `lowrankmerge.cpp` implement a C++ function

```
std::pair<MatrixXd, MatrixXd> low_rank_merge (
    const MatrixXd &A1, const MatrixXd &B1,
    const MatrixXd &A2, const MatrixXd &B2);
```

that takes two rank- q -matrices $\mathbf{X}_1, \mathbf{X}_2 \in \mathbb{R}^{n,n}$ in factorized form as in (2.6.1) and returns the rank- q best approximation $\tilde{\mathbf{Z}}$ of the “horizontal” concatenation \mathbf{Z} in the form of its low-rank factors.

SOLUTION for (2-6.b) → [2-6-2-0:lrm2.pdf](#) ▲

(2-6.c)  30 In the file `lowrankmerge` implement a C++ function

```
std::pair<double, double> test_low_rank_merge (size_t n);
```

that uses the low-rank matrices $\mathbf{K}_1, \mathbf{K}_2$ from Sub-problem (2-6.a), applies `low_rank_merge()` to them and returns the maximum norm (entry-wise) and scaled Frobenius norm

$$\frac{1}{n} \|\mathbf{Z} - \tilde{\mathbf{Z}}\|_F$$

of the approximation error.

Tabulate the errors for $n = 2^p$, $p = 3, \dots, 12$. To that end you may introduce some auxiliary function or simply extend the `main()` function in the file `lowrankmerge_main.cpp`.

HIDDEN HINT 1 for (2-6.c) → [2-6-3-0:1h.pdf](#)


SOLUTION for (2-6.c) → [2-6-3-1:lrm3.pdf](#) ▲

(2-6.d)  By copying and modifying `low_rank_merge()` create a C++ function

```
std::pair<MatrixXd, MatrixXd> adap_rank_merge (
    const MatrixXd &A1, const MatrixXd &B1,
    const MatrixXd &A2, const MatrixXd &B2, double rtol, double atol);
```

that does *adaptive low-rank merging* in the spirit of [Lecture → Rem. 2.4.2.20]. The arguments `rtol`, `atol` pass the desired relative tolerances.

SOLUTION for (2-6.d) → [2-6-4-0:lr4.pdf](#) ▲

(2-6.e)  Code a C++ function

```
std::pair<double, size_t> test_adap_rank_merge (size_t n, double
    rtol);
```

that returns the scaled Frobenius norm of the approximation error and the rank required to achieve the prescribed tolerance `rtol`, when using `adap_rank_merge()` from Sub-problem (2-6.d). Internally the function should use the low-rank matrices X_1, X_2 from Sub-problem (2-6.a) of size n .

For the low-rank matrices K_1 and K_2 examined in Sub-problem (2-6.a) tabulate the resulting approximation errors and ranks for

- $n = 2^p, p = 3, \dots, 12, \text{rtol} = 10^{-4}$,
- $\text{rtol} \in \{10^{-1}, 10^{-2}, \dots, 10^{-8}\}, n = 500$.

SOLUTION for (2-6.e) → [2-6-5-0:lr5.pdf](#) ▲

End Problem 2-6 , 110 min.

Problem 2-7: A special family of hierarchical matrices

In this problem we study hierarchical matrices with a special structure, see [Lecture → Ex. 2.4.1.21] and [Hac15, Ch. 3]. For them we investigate the complexity of basic operations.

Definition 2.7.1. Semi-separable hierarchical matrices

For $p \in \mathbb{N}$, the set $\mathcal{S}_p \subset \mathbb{R}^{n,n}$, $n := 2^p$, of square hierarchical matrices [Lecture → Def. 2.4.1.2] with local rank $q \in \mathbb{N}$ is defined as follows:

- ◆ Row and column trees coincide and are binary balanced cluster trees $\mathcal{T}_{\mathbb{I}}$ of $\mathbb{I} := \{1, \dots, n\}$.
- ◆ The underlying admissibility condition [Lecture → Def. 2.3.3.2] is

$$\text{adm}(v, w) = \text{true} \Leftrightarrow \mathcal{I}(v) \cap \mathcal{I}(w) = \emptyset, \quad (2.7.2)$$

where $\mathcal{I}(v)$, $\mathcal{I}(w)$ are the index sets associated with the nodes $v, w \in \mathcal{T}_{\mathbb{I}}$, see [Lecture → Def. 2.3.2.14].

For the remainder of this problem we fix $q \in \mathbb{N}$. We also assume a storage format for hierarchical matrices in \mathcal{S}_p meeting [Lecture → Ass. 2.4.1.7].

Assumption 2.7.3.

We assume that every leaf node of the cluster tree contains q indices.

Remark 2.7.4 (Counting elementary arithmetic operations) Below you will be asked to determine the number of elementary arithmetic operations required for certain matrix operations. In this case, apply the following rules:

- A dot product of two n -vectors requires n operations.
- A SAXPY operation for two n -vectors involves n operations (SAXPY operations are of the form $\vec{\xi} := \alpha \vec{v} + \vec{\mu}$, $\alpha \in \mathbb{R}$, $\vec{v}, \vec{\mu} \in \mathbb{R}^n$)
- In the context of economical SVD and QR-decomposition of a rank- q matrix, assume that its factorization $\mathbf{A}\mathbf{B}^T$, with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,q}$, is given with no cost.

(2-7.a) ☐ (15 min.) Using the notations from Def. 2.7.1, give a visual rendering of the cluster tree $\mathcal{T}_{\mathbb{I}}$ for $p = 3$ as in [Lecture → Ex. 2.3.2.17].

SOLUTION for (2-7.a) → [2-7-1-0:shm1.pdf](#) ▲

(2-7.b) ☐ (10 min.) Prove that (2.7.2) is a valid admissibility condition in the sense of [Lecture → Def. 2.3.3.2].

SOLUTION for (2-7.b) → [2-7-2-0:ss1a.pdf](#) ▲

(2-7.c) ☐ (10 min.) Assume a **compatible ordering** of the index sets held by the clusters according to [Lecture → § 2.4.5.5].

Taking the cue from [Lecture → Fig. 107], sketch the block partition of a matrix $\mathbf{H} \in \mathcal{S}_4$ highlighting the far-field blocks.

SOLUTION for (2-7.c) → [2-7-3-0:shm2.pdf](#) ▲

(2-7.d) ☹️ (20 min.) Let $\mathbb{F}_p \subset 2^{\mathbb{I}} \times 2^{\mathbb{I}}$ denote the partition of $\mathbb{I} \times \mathbb{I}$ inducing the block partition of hierarchical matrices $\in \mathcal{S}_p$. Compute the **sparsity measure** $\text{spm}(\mathbb{F}_p)$ according to [Lecture → Def. 2.3.4.27].

SOLUTION for (2-7.d) → 2-7-4-0:hms3.pdf ▲

(2-7.e) ☹️ (30 min.) Denote by $W_{mv}(p)$ the number of arithmetic operations required for the multiplication of $\mathbf{H} \in \mathcal{S}_p$ with a vector $\vec{\mu} \in \mathbb{R}^n$, $n = 2^p$. Derive a **recursion formula** for $W_{mv}(p)$ and find a closed-form expression. Apply the rules of Rem. 2.7.4.

HIDDEN HINT 1 for (2-7.e) → 2-7-5-0:hpmv4.pdf

SOLUTION for (2-7.e) → 2-7-5-1:spmv4.pdf ▲

(2-7.f) ☹️ (30 min.) Develop a recursive algorithm realized as the function

$[\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n,q}] \leftarrow \text{ssmlrm}(\mathcal{H}\text{-matrix } \mathbf{H} \in \mathcal{S}_p, \text{Matrix } \mathbf{A}, \text{Matrix } \mathbf{B});$

(in *pseudocode notation* as in the course notes) for the multiplication of $\mathbf{H} \in \mathcal{S}_p$ with a rank- q matrix in factorized form $\mathbf{A}\mathbf{B}^\top$, $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,q}$, $n := 2^p$. The resulting rank- q matrix should again be returned in factorized form.

HIDDEN HINT 1 for (2-7.f) → 2-7-6-0:hpmm5.pdf

SOLUTION for (2-7.f) → 2-7-6-1:spmm5.pdf ▲

(2-7.g) ☹️ (20 min.) Denote by $W_{mr}(p)$ the number of elementary operations involved in a single invocation of your function `ssmlrm()` from Sub-problem (2-7.f) for a semi-separable hierarchical matrix $\in \mathcal{S}_p$. Find a closed-form expression for $W_{mr}(p)$.

SOLUTION for (2-7.g) → 2-7-7-0:spmm6.pdf ▲

(2-7.h) ☹️ (60 min.) Taking the cue from `hmat_mult_add()` from [Lecture → Code 2.4.4.31], devise a recursive algorithm (as a function

`void ssmm(ref \mathcal{H} -matrix $\mathbf{H} \in \mathcal{S}_p$, const \mathcal{H} -matrix $\mathbf{Y} \in \mathcal{S}_p$, const \mathcal{H} -matrix $\mathbf{Z} \in \mathcal{S}_p$);`

in pseudocode notation) for the *approximate* multiplication of two matrices $\mathbf{Y}, \mathbf{Z} \in \mathcal{S}_p$ with the result added to another hierarchical matrix $\mathbf{H} \in \mathcal{S}_p$.

The following functions discussed in class can be used:

- `low_rank_update()` from [Lecture → Code 2.4.3.3],
- `low_rank_sum()` from [Lecture → Code 2.4.2.24],
- `hmat_mult_dense()` and `hmat_trnaspose_mult_dense()` as used in [Lecture → Code 2.4.4.31].

HIDDEN HINT 1 for (2-7.h) → 2-7-8-0:hsmrc1.pdf

HIDDEN HINT 2 for (2-7.h) → 2-7-8-1:hsmrc1.pdf

HIDDEN HINT 3 for (2-7.h) → 2-7-8-2:hsmrc2.pdf

HIDDEN HINT 4 for (2-7.h) → 2-7-8-3:hsmrc3.pdf

HIDDEN HINT 5 for (2-7.h) → 2-7-8-4:hsmmr4.pdf

SOLUTION for (2-7.h) → 2-7-8-5:ssmrc.pdf ▲

(2-7.i) ☹️ (20 min.) Assuming a *compatible ordering* of the index sets of the clusters according to [Lecture → § 2.4.5.5], we consider a *normalized lower-triangular* semi-separable \mathcal{H} -matrix $\mathbf{L} \in \mathcal{S}_p$.

What is the asymptotic computational cost of calling `hmat_forw_elim()` from [Lecture → Code 2.4.5.13] for \mathbf{L} ?

SOLUTION for (2-7.i) → [2-7-9-0:luss.pdf](#) ▲

(2-7.j) 🧩 (20 min.) Pursue the considerations of [Lecture → § 2.4.5.14] about recursive LU-decomposition of \mathcal{H} -matrices for the special case of semi-separable \mathcal{H} -matrices according to Def. 2.7.1. What kind of matrix equations are encountered in a recursive algorithm?

HIDDEN HINT 1 for (2-7.j) → [2-7-10-0:hsmrc.pdf](#)

SOLUTION for (2-7.j) → [2-7-10-1:lussa.pdf](#) ▲

(2-7.k) 🧩 (30 min.) Following `ssmlrm()` from Sub-problem (2-7.f), in pseudocode notation outline a function

$[\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,q}] \leftarrow \text{ssmat_triag_solve}(\mathcal{H}\text{-matrix } \mathbf{L} \in \mathcal{S}_p, \text{ matrix } \mathbf{U} \in \mathbb{R}^{n,q}, \mathbf{V} \in \mathbb{R}^{n,q})$

that takes an invertible lower-triangular semi-separable \mathcal{H} -matrix $\mathbf{L} \in \mathcal{S}_p$ as argument and finds two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,q}$, $n := 2^p$, such that

$$\mathbf{L}(\mathbf{AB}^\top) = \mathbf{UV}^\top.$$

HIDDEN HINT 1 for (2-7.k) → [2-7-11-0:9bhi1.pdf](#)

HIDDEN HINT 2 for (2-7.k) → [2-7-11-1:9bhi1.pdf](#)

HIDDEN HINT 3 for (2-7.k) → [2-7-11-2:9bhi2.pdf](#)

SOLUTION for (2-7.k) → [2-7-11-3:ss9b.pdf](#) ▲

(2-7.l) 🧩 (20 min.) Denote by $W_{hts}(p)$ the computational cost for a call to the function `ssmat_triag_solve()` from Sub-problem (2-7.k) for $\mathbf{L} \in \mathcal{S}_p$. Derive a recursion for this quantity and try to find a closed-form expression.

SOLUTION for (2-7.l) → [2-7-12-0:compl10.pdf](#) ▲

References

[Hac15] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Vol. 49. Springer Series in Computational Mathematics. Springer, Heidelberg, 2015, pp. xxv+511 (cit. on p. 31).

End Problem 2-7, 285 min.

Chapter 3

Convolution Quadrature

Problem 3-1: Computing with lower triangular Toeplitz matrices

In [Lecture → Eq. (3.1.4.22)], we learnt that discrete convolution of causal sequences boils down to matrix-vector multiplication for lower triangular Toeplitz matrices.

In this problem, we practise the relevant efficient algorithms presented in [Lecture → § 3.1.5.1], [Lecture → § 3.1.5.14], and [Lecture → § 3.1.5.21].

Involves C++ coding based on EIGEN

▷ problem name/problem code folder: LowTriangToeplitz in [Code repository](#)

A lower triangular square Toeplitz matrix

$$\mathbf{K} := \begin{bmatrix} f_0 & 0 & \dots & \dots & \dots & 0 \\ f_1 & f_0 & 0 & \dots & & \vdots \\ f_2 & f_1 & f_0 & 0 & \dots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ f_N & f_{N-1} & \dots & f_2 & f_1 & f_0 \end{bmatrix} \in \mathbb{C}^{N+1, N+1} \quad (3.1.1)$$

is uniquely defined by specifying the sequence (f_0, \dots, f_N) . We adopt the notation $\text{ltp}(f_0, \dots, f_N)$ to reference such a matrix.

(3-1.a) 🕒 (10 min.) True or False: The product of two Toeplitz matrices yields a Toeplitz matrix. Justify your answer.

SOLUTION for (3-1.a) → [3-1-1-0:tps1.pdf](#) ▲

(3-1.b) 🕒 (15 min.) Show that the product of two lower triangular square Toeplitz matrices yields a lower triangular square Toeplitz matrix.

SOLUTION for (3-1.b) → [3-1-2-0:tps2.pdf](#) ▲

(3-1.c) 🕒 (60 min.) Implement an efficient C++ function

```
Eigen::VectorXcd ltpMult(const Eigen::VectorXcd& f,
                        const Eigen::VectorXcd& g);
```

which computes the sequence associated with the product of two square lower triangular Toeplitz matrices passed through their sequences in vectors \mathbf{f} and \mathbf{g} . Do not forget to test your implementation.

HIDDEN HINT 1 for (3-1.c) → [3-1-3-0:tp3h1.pdf](#)

HIDDEN HINT 2 for (3-1.c) → [3-1-3-1:tp3h2.pdf](#)

SOLUTION for (3-1.c) → [3-1-3-2:stp4.pdf](#) ▲

(3-1.d) 🕒 (30 min.) Compare the runtime and complexity of your implementation for `ltpMult()` with the EIGEN based matrix-vector product. To that end in the file `lowtriangtoeplitz.cpp` implement a C++ function

```
std::tuple<double, double, double> runtimes_ltpMult(unsigned int
N);
```

that measures and returns the runtimes for multiplying two $(N+1) \times (N+1)$ lower-triangular Toeplitz matrices

1. when using EIGEN's built-in multiplication of dense matrices (the time it takes to initialize these matrices should not be taken into account),
2. when multiplying one of the Toeplitz matrices with the vector defining the second, and,
3. when using `ltpMult()` from Sub-problem (3-1.c).

The Toeplitz matrices can simply be based on random sequences that can be generated by `Eigen::VectorXcd::Random(N+1)`.

Extend `lowtriangtoeplitz_main.cpp` so that it outputs the measured runtimes for $N = 2^p - 1$, $p = 3, 4, \dots, 13$ and create a doubly logarithmic plot the runtimes versus N using, e.g., a short PYTHON script.

HIDDEN HINT 1 for (3-1.d) → [3-1-4-0:gfh7.pdf](#)

SOLUTION for (3-1.d) → [3-1-4-1:tpsp4.pdf](#) ▲

(3-1.e) 🕒 (60 min.) For matrix sizes $n = 2^p$, $p \in \mathbb{N}$, in the file `lowtriangtoeplitz.cpp` implement the recursive algorithm for the solution of the triangular linear system of equations $\mathbf{K}\mathbf{u} = \mathbf{y}$ derived in [Lecture → § 3.1.5.21] in the C++ function

```
Eigen::VectorXcd ltpSolve(const Eigen::VectorXcd& f, const
Eigen::VectorXcd& y);
```

which should return $\mathbf{K}^{-1}\mathbf{y}$, \mathbf{K} as in (3.1.1), provided that $f_0 \neq 0$. Pay attention to efficiency!

HIDDEN HINT 1 for (3-1.e) → [3-1-5-0:tp5h1.pdf](#)

SOLUTION for (3-1.e) → [3-1-5-1:tpsp5.pdf](#) ▲

(3-1.f) 🕒 (30 min.) In the file `lowtriangtoeplitz.cpp` write a C++ function

```
std::pair<double, double> runtimes_lptSolve(unsigned int n);
```

that measures the runtimes of for solving an $n \times n$ lower-triangular Toeplitz linear system by means of

1. EIGEN's built-in triangular solver as explained in [NumCSE course → Rem. 2.5.0.5],
2. your function `ltpSolve()` from Sub-problem (3-1.e).

Use a Toeplitz matrix generated by the sequence $(1, 1/2, 1/3, 1/4, \dots)$ and a right-hand side vector $[1, \dots, 1]^T$.

Extend `lowtriangtoeplitz_main.cpp` so that it outputs the measured runtimes for $N = 2^p - 1$, $p = 3, 4, \dots, 13$ and create a doubly logarithmic plot the runtimes versus N using, e.g., a short PYTHON script.

SOLUTION for (3-1.f) → [3-1-6-0:tpsp6.pdf](#) ▲

End Problem 3-1 , 205 min.

Problem 3-2: Abel integral equation

[Lecture → Section 3.2.1] introduces the Abel integral equation as an example of a convolution equation, for an unknown scalar-valued function. This problem is split in three parts: In part I, we will try to understand some properties of the Abel integral equation. In part II and III, we solve the Abel integral equation using a Galerkin discretization method and convolution quadrature, respectively.

Involves C++ coding based on EIGEN.

▷ problem name/problem code folder: `AbelIntegralEquation` in [Code repository](#)

For an unknown causal function $u : \mathbb{R} \rightarrow \mathbb{R}$, the Abel integral equation [Lecture → Eq. (3.2.1.6)] in the standard form reads

$$(Au)(t) := \int_0^t \frac{u(\xi)}{\sqrt{t-\xi}} d\xi = y(t), \quad 0 \leq t \leq 1, \quad (3.2.1)$$

with a given a causal right-hand side function $y : \mathbb{R} \rightarrow \mathbb{R}$.

(3-2.a) ☐ (10 min.) Let $*$ denote the convolution of causal functions,

Definition [Lecture → Def. 3.1.1.2]. **Convolution on the real line**

Given two functions $f, g \in L^1(\mathbb{R})$, their **convolution** $f * g \in L^1(\mathbb{R})$ is defined as

$$(f * g)(t) := \int_{\mathbb{R}} f(t - \xi) g(\xi) d\xi = \int_{\mathbb{R}} f(\xi) g(t - \xi) d\xi, \quad t \in \mathbb{R}.$$

The equation (3.2.1) can be written in convolution form $f * u = y$. Find the causal **convolution kernel**, that is, the causal function $f : \mathbb{R} \rightarrow \mathbb{R}$.

SOLUTION for (3-2.a) → [3-2-1-0:a1.pdf](#) ▲

(3-2.b) ☐ (10 min.) [depends on Sub-problem (3-2.a)]

Compute or look up the Laplace transform of the kernel function f found in Sub-problem (3-2.a).

SOLUTION for (3-2.b) → [3-2-2-0:ae2.pdf](#) ▲

(3-2.c) ☒ (30 min.) Directly relying on the definition of the **Abel integral operator** A from (3.2.1), use substitutions to verify

$$(A^2u)(t) = \pi \int_0^t u(\xi) d\xi, \quad 0 \leq t \leq 1, \quad \forall \text{ causal continuous } u : \mathbb{R} \rightarrow \mathbb{R}. \quad (3.2.3)$$

HIDDEN HINT 1 for (3-2.c) → [3-2-3-0:ae31.pdf](#)

HIDDEN HINT 2 for (3-2.c) → [3-2-3-1:ae32.pdf](#)

SOLUTION for (3-2.c) → [3-2-3-2:aiea3.pdf](#) ▲

First we examine a **spectral Galerkin discretization** [NumPDE course → Section 4.3] of (3.2.1) based on the trial and test space

$$V_p := \{v : [0, 1] \rightarrow \mathbb{R} : v \in \mathcal{P}_p(\mathbb{R}), v(0) = 0\} \quad (3.2.5)$$

of **causal polynomials** of degree $\leq p$, $p \in \mathbb{N}$, that is, we also require the polynomials to vanish in 0.

For the computation of the Galerkin matrices, we use the monomial basis

$$\mathfrak{B}_h := \{t \mapsto t^q, q = 1, \dots, p\} \implies V_p = \text{Span}\{\mathfrak{B}_h\}. \quad (3.2.6)$$

Remark: The monomial basis is notoriously affected by instability, recall [NumPDE course → Rem. 4.3.0.5]. A better choice of basis would rely on Legendre polynomials [NumPDE course → Def. 4.3.0.9] or Chebychev polynomials [NumCSE course → Def. 6.2.3.3]. For the sake of simplicity, we do not consider these more numerically stable choices here.

(3-2.d) 🕒 (20 min.) Give explicit formulas for the entries of the Galerkin matrix and the right hand side vector resulting from the spectral polynomial Galerkin discretization of (3.2.1) using the basis (3.2.6).

SOLUTION for (3-2.d) → [3-2-4-0:aesg1.pdf](#) ▲

(3-2.e) 🕒 (30 min.) [depends on Sub-problem (3-2.d)]

Directly compute the entries of the Galerkin matrix for the spectral causal polynomial Galerkin discretization of (3.2.1) using the basis (3.2.6).

HIDDEN HINT 1 for (3-2.e) → [3-2-5-0:sg2h1.pdf](#)

SOLUTION for (3-2.e) → [3-2-5-1:aesg2.pdf](#) ▲

(3-2.f) 🕒 (45 min.) [depends on Sub-problem (3-2.e)]

Based on EIGEN implement a C++ function

```
template <typename FUNCTION>
Eigen::VectorXd poly_spec_abel(FUNCTION&& y, size_t p, unsigned int
    m);
```

that relies on the causal polynomial spectral Galerkin approach with polynomials up to degree p to solve (3.2.1) and returns the values of the solution in the nodes of an equidistant grid of $[0, 1]$ with $m \in \mathbb{N}$ cells. The argument y passes a functor object providing the *causal* right hand side function y .

Use p -point Gauss-Legendre quadrature [NumCSE course → Def. 7.4.2.18] to compute the entries of the right hand side vector. Nodes and weights of Gauss-Legendre quadrature are available by calling the function `gauleg()`, which is already implemented in `abelintegralequation.cpp`.

SOLUTION for (3-2.f) → [3-2-6-0:aesg3.pdf](#) ▲

(3-2.g) 🕒 (30 min.) [depends on Sub-problem (3-2.f)]

For $y(t) = t$ use the causal polynomial spectral Galerkin method implemented in Sub-problem (3-2.f) to solve (3.2.1) for $p = 2, 3, \dots, 10$ and tabulate the maximum norm of the discretization error (approximated by sampling on a fine equidistant grid). Show the empirical rate of convergence (EOC) in an additional column.

The exact solution is $u(\xi) = \frac{2}{\pi} \sqrt{\xi}$.

Describe qualitatively and quantitatively the empiric convergence of the method.

SOLUTION for (3-2.g) → [3-2-7-0:aesg4.pdf](#) ▲

(3-2.h) 🕒 (30 min.) [depends on Sub-problem (3-2.b)]

Next, we apply **convolution quadrature** for the discretization of (3.2.1). Determine the convolution quadrature weights, when convolution quadrature based on the implicit Euler method [Lecture → Def. 3.3.2.14] is applied to the Abel integral operator.

HIDDEN HINT 1 for (3-2.h) → [3-2-8-0:cq1h0.pdf](#)

HIDDEN HINT 2 for (3-2.h) → [3-2-8-1:cq1h1.pdf](#)

SOLUTION for (3-2.h) → [3-2-8-2:aiecq1.pdf](#) ▲

(3-2.i) 🕒 (30 min.) Determine the convolution quadrature weights for the Abel integral operator, when convolution quadrature based on the **BDF-2 2-step method** [Lecture → Eq. (3.4.1.10)] is used instead, which for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ and timestep $\tau > 0$ reads

$$\frac{3}{2}\mathbf{y}_{n+2} - 2\mathbf{y}_{n+1} + \frac{1}{2}\mathbf{y}_n = \tau\mathbf{f}(t_{n+2}, \mathbf{y}_{n+2}), \quad n \in \mathbb{Z}. \quad (3.2.15)$$

HIDDEN HINT 1 for (3-2.i) → [3-2-9-0:cq2h0.pdf](#)

HIDDEN HINT 2 for (3-2.i) → [3-2-9-1:cq2h1.pdf](#)

SOLUTION for (3-2.i) → [3-2-9-2:aiecq2.pdf](#) ▲

(3-2.j) 🕒 (45 min.) [depends on Sub-problem (3-2.h)]

Implement a C++ function

```
template<typename FUNCTOR>
Eigen::VectorXd cq_ieul_abel (FUNCTOR&& f, unsigned int N);
```

which solves the discrete convolution equation arising from the convolution quadrature discretization of (3.2.1), based on the implicit Euler method with timestep $\tau = N^{-1}$, and returns the solution vector. You will face a linear system of equations to which, for the sake of simplicity, you may apply EIGEN's built-in triangular elimination solver discussed in [NumCSE course → Rem. 2.5.0.5].

Remark. Of course, a really efficient implementation with optimal asymptotic complexity for $N \rightarrow \infty$ should rely on the *divide-and-conquer algorithm* for lower triangular Toeplitz linear systems developed in [Lecture → ??]. You may have already coded that algorithm in Problem 3-1. Unfortunately, the function `ltpSolve()` from that problem can only deal with N being a power of 2.

HIDDEN HINT 1 for (3-2.j) → [3-2-10-0:cq3h1.pdf](#)

SOLUTION for (3-2.j) → [3-2-10-1:aiecq3.pdf](#) ▲

(3-2.k) 🕒 (45 min.) [depends on Sub-problem (3-2.i)]

Implement a C++ function

```
template<typename FUNCTOR>
Eigen::VectorXd cq_bdf2_abel (FUNCTOR&& f, unsigned int N);
```

which solves the discrete convolution equation arising from the convolution quadrature discretization of (3.2.1), this time based on the **BDF-2 method** with timestep $\tau = N^{-1}$. It should return a vector of approximations of $u = u(t)$ in the mesh nodes $t_j = \frac{j}{N}$, $j = 0, \dots, N$. Further instructions for Sub-problem (3-2.j) still apply.

SOLUTION for (3-2.k) → [3-2-11-0:aiecq4.pdf](#) ▲

(3-2.l) 🕒 (30 min.) We consider (3.2.1) with the right-hand side function $y(t) = t$, for which we can compute the exact solution $u(\xi) = \frac{2}{\pi}\sqrt{\xi}$, $\xi \in [0, 1]$.

Write a main function in `abelintegralequation_main.cpp` that tabulates the maximum norms of the (restrictions to the grid of the) discretization errors for the CQ-based discretization of (3.2.1) relying on both the implicit Euler method and the BDF-2 method. Moreover, use an additional column to print

the empirical rate of convergences (EOC). Use an appropriate sequence of timesteps N to demonstrate the convergence properties of the methods.

Of course, you should rely on your implementations of `cq_ieul_abel()` and `cq_bdf2_abel()` from Sub-problem (3-2.j) and Sub-problem (3-2.k)

Describe the empirical convergence of the methods in qualitative and quantitative terms.

SOLUTION for (3-2.l) → [3-2-12-0:aiecq5.pdf](#)



End Problem 3-2, 355 min.

Problem 3-3: Convolution-based Absorbing Boundary Condition

Here we study a two-point boundary value problem with “impedance-type” boundary conditions, structurally similar to what was discussed in [Lecture → Section 3.2.2].

Involves C++ coding based on EIGEN.

Template: [Get it on 🍷 GitLab.](#)

Solution: [Get it on 🍷 GitLab.](#)

▷ problem name/problem code folder: AbsorbingBoundaryConditions in [Code repository](#)

For an unknown causal function $u(x, t)$ on $]0, 1[\times]0, 1[$, we consider the following evolution problem in one spatial dimension:

$$-\frac{\partial^2 u}{\partial x^2} + \sin(\pi x)u = 0 \quad \text{in }]0, 1[\times]0, 1[, \quad (3.3.1a)$$

$$u(x, 0) = 0 \quad \text{for } 0 < x < 1 , \quad (3.3.1b)$$

$$\frac{\partial u(0, t)}{\partial x} = g(t) \quad \text{for } 0 \leq t \leq 1 , \quad (3.3.1c)$$

$$\frac{\partial u(1, t)}{\partial x} = -(f * u(1, \cdot))(t) \quad \text{for } 0 \leq t \leq 1 . \quad (3.3.1d)$$

Here, $g : \mathbb{R} \rightarrow \mathbb{R}$ is a given continuous causal function, and f is a continuous causal function which models some absorbing boundary conditions at $x = 1$. Only the Laplace transform of f is known:

$$F(s) = \frac{\log s}{s^2 + 1}, \quad s \in \mathbb{C}^+ .$$

A *finite-element Galerkin semi-discretization in space* based on continuous piecewise linear finite elements [NumPDE course → Section 2.3], on a equidistant spatial mesh \mathcal{M} of $[0, 1]$ with meshwidth $h := \frac{1}{M} > 0$, $M \in \mathbb{N}$, leads to an evolution problem:

$$\mathbf{A}\vec{\mu}(t) + (f * \mathbf{B}\vec{\mu}(\cdot))(t) = \vec{\varphi}(t) \quad (3.3.2)$$

for the time-dependent vector $\vec{\mu} : [0, 1] \rightarrow \mathbb{R}^{N+1}$ of basis expansion coefficients. Here, $N + 1$ is the dimension of the finite element space.

For discretization in time, use a **multistep convolution quadrature** with uniform timestep $\tau > 0$ based on the second-order backward difference formula (**BDF-2**), see [Lecture → Ex. 3.4.1.5], [Lecture → Eq. (3.4.1.10)].

(3-3.a) ☞ Find the matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{N, N}$, $N := M + 1$, and the right-hand side vector $\vec{\varphi} = \vec{\varphi}(t)$. Use the standard nodal basis of hat functions, see [NumPDE course → Section 2.3.2].

In principle, closed form expressions can be computed for the entries of the matrices \mathbf{A}, \mathbf{B} , but here you should rely on the local/composite trapezoidal quadrature rule on each element of the spatial mesh [NumPDE course → Eq. (2.3.3.8)]:

$$\int_0^1 \varphi(x) dx \approx \sum_{\ell=1}^M \frac{1}{2} h (\varphi(h(\ell-1)) + \varphi(h\ell)) . \quad (3.3.3)$$

SOLUTION for (3-3.a) → [3-3-1-0:a1.pdf](#) ▲

(3-3.b) ☞ What is the domain of analyticity of F ?

SOLUTION for (3-3.b) → [3-3-2-0:ae2.pdf](#) ▲

(3-3.c) ☒ The CQ-weights are explicitly given by their integral form

$$w_\ell^{F,\tau} = r^{-\ell} \int_0^1 e^{-2\pi i \ell \varphi} F\left(\frac{\delta(re^{2\pi i \varphi})}{\tau}\right) d\varphi, \quad \ell \in \mathbb{Z}, \quad 0 < r < 1. \quad ((??))$$

Approximating the integral from of the CQ-weights by the composite trapezoidal rule with $N + 1$ nodes, as discussed in the lecture (see [Lecture → Eq. (3.4.3.18)]), yields

$$w_\ell^{F,\tau} \approx \tilde{w}_\ell^{F,\tau} := \frac{r^{-\ell}}{N+1} \sum_{k=0}^N \exp(-2\pi i \frac{\ell k}{N+1}) f_k, \quad \ell \in \mathbb{Z},$$

$$f_k := F\left(\frac{1}{\tau} \delta\left(r \exp(2\pi i \frac{k}{N+1})\right)\right), \quad k = 0, \dots, N.$$

The radius of integral contour $r < 1$ must strike a balance between numerical errors and roundoff errors. An analysis of those errors, that goes beyond the scope of the lecture, shows that an appropriate choice is $r = \text{EPS}^{-\frac{1}{2N+1}}$. Here, **EPS** denotes the machine precision, which is roughly 10^{-16} for 64-bit floating-point numbers (double).

Implement a C++ function

```
template <typename FFUNC, typename DFUNC>
VectorXd cqweights_by_dft(const FFUNC& F, const DFUNC& delta,
    double tau, size_t M,);
```

which computes the approximations $w_\ell^{F,\tau}$, for $\ell = 0, \dots, n$, by evaluating the integral using the $N + 1$ -point uniform trapezoidal rule. The argument **F** passes a functor which evaluates the transfer function F , the argument **delta** passes a functor which evaluates the characteristic function $\delta(z)$.

For testing purposes, you can use $\delta(z) = 1 - z$ and $F(s) = s$. The (exact !) resulting convolution quadrature weights then recover the coefficients of the implicit Euler method, i.e.

$$\text{CQ}_\tau(F) = \left(\frac{1}{\tau}, \frac{-1}{\tau}, 0, \dots, 0\right) \in \mathbb{R}^{M+1}.$$

SOLUTION for (3-3.c) → [3-3-3-0:abca2a.pdf](#) ▲

(3-3.d) ☒ Perform temporal discretization of (3.3.2) by means of multistep convolution quadrature based on BDF-2, which is characterized by

$$\delta(z) := \frac{1}{2}z^2 - 2z + \frac{3}{2},$$

with $M \in \mathbb{N}$ timesteps. Implement a C++ function

```
template <typename FUNC>
Eigen::VectorXd solve_IBVP(const FUNC& g, size_t M, size_t N);
```

which uses Galerkin finite element discretization in space, as described above (based on a uniform mesh with N cells), and convolution quadrature in time (with M uniform timesteps on $[0, 1]$), and returns the values of the solution at time $t = 1$. The argument **phi** passes a function object providing the *causal* right hand side function g . The argument **p** specifies the number of quadrature nodes used for the trapezoidal rule to compute the convolution weights, see Sub-problem (3-3.c).

HIDDEN HINT 1 for (3-3.d) → [3-3-4-0:a3h1.pdf](#)

SOLUTION for (3-3.d) → [3-3-4-1:abca3.pdf](#) ▲

(3-3.e) ☐ Let $u_{N,k} \approx u_N(k\tau)$ denote the fully discrete solution for $g(t) = \sin(\pi t)$, where $u_N(t)$ is the solution of the semi-discrete problem and $\tau > 0$ is the timestep.

Consider $u_{ref} := u_{4096,4096}$ as the reference solution. Tabulate

$$e_{rel} = \left[\int_0^1 \left| \frac{du_{ref}}{dx} - \frac{du_{N,M}}{dx} \right|^2 dx \right]^{\frac{1}{2}} \left[\int_0^1 \left| \frac{du_{ref}}{dx} \right|^2 dx \right]^{-\frac{1}{2}}$$

for

- $N = 16, 32, 64, \dots, 1024$, with $M = 4096$
- $M = 16, 32, 64, \dots, 1024$, with $N = 4096$

and examine the convergence behaviour.

HIDDEN HINT 1 for (3-3.e) → [3-3-5-0:a4h1.pdf](#)

SOLUTION for (3-3.e) → [3-3-5-1:abca3.pdf](#) ▲

End Problem 3-3

Problem 3-4: Fractional Parabolic Evolution Problem

[Lecture → Ex. 3.6.0.1] introduced evolution initial-boundary value problems (IVPs) on space-time cylinders with fractional derivatives in time, which generalize the heat equation $\partial_t u - \Delta u = f$. Following the policy of the method of lines after finite-element Galerkin discretization in space we can employ convolution quadrature for discretization in time. This problem will explore solution strategies for the resulting fully discrete problem.

This problem requires familiarity with ?? and ??, in addition to proficiency in numerical linear algebra. It includes C++ coding tasks.

▷ problem name/problem code folder: FractionalHeatEquation in [Code repository](#)

A specific [anomalous diffusion](#) process on the unit square $\Omega :=]0, 1[^2$ is modeled by the half-order fractional diffusion initial/boundary value problem

$$F(\partial_t)u(x, t) - \Delta_x u(x, t) = f(x, t) \quad \text{in } \Omega \times]0, T[, \tag{3.4.1a}$$

$$u(x, t) = 0 \quad \text{on } \partial\Omega \times]0, T[, \tag{3.4.1b}$$

$$u(x, t) = 0 \quad \text{for all } x \in \Omega, t < 0 , \tag{3.4.1c}$$

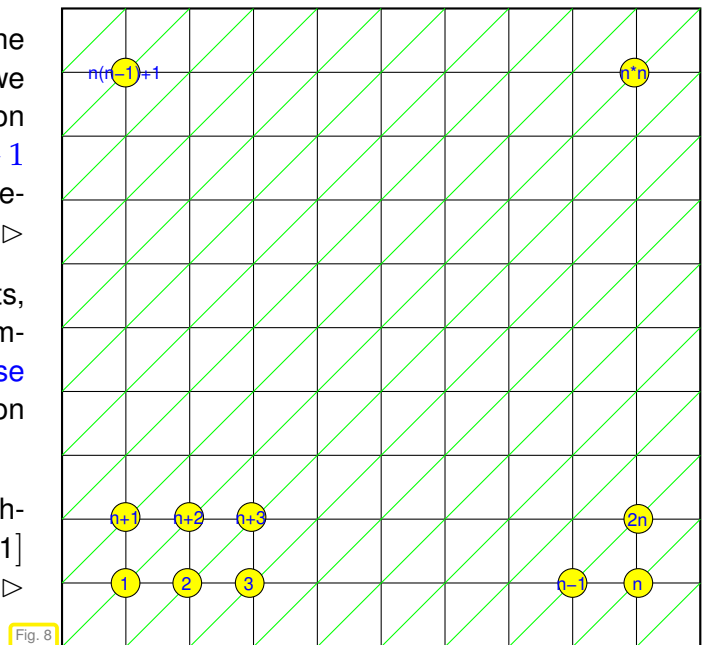
with $F(s) = \sqrt{s}$, $s \in \mathbb{C} \setminus]-\infty, 0]$. By virtue of [Lecture → Eq. (3.2.1.8)], the equation (3.4.1a) can be rewritten as Abel integral equation in time

$$\int_0^t \frac{u(x, \tau)}{\pi\sqrt{t-\tau}} d\tau - \Delta_x u(x, t) = f(x, t) \quad \text{in } \Omega \times]0, T[. \tag{3.4.2}$$

In the spirit of the method-of-lines approach to the discretization of evolution problems, in space we employ a finite-element Galerkin discretization on a “tensor-product triangular mesh” \mathcal{M}_n with $n + 1$ cells in both directions, $n \in \mathbb{N}$, see the drawing beside. ▷

We rely on lowest-order Lagrangian finite elements, the discrete trial and test spaces $\mathcal{S}_1^0(\mathcal{M}_n)$ with imposed zero boundary conditions [NumPDE course → Section 2.4.2], and the standard tent function bases.

We number the $N := n^2$ basis functions lexicographically as explained in [NumPDE course → § 4.1.2.1] and indicated in the drawing. ▷



Moreover, for the approximate evaluation of all occurring integrals over Ω we use the \mathcal{M}_N -local 2D trapezoidal quadrature rule (three-point vertex-based quadrature rule) [NumPDE course → Eq. (2.4.6.10)].

In [Lecture → Ex. 3.6.0.1] we saw that after spatial discretization we arrive at the convolution causal evolution problem in \mathbb{R}^N , $N := n^2$,

$$(F(\partial_t)\mathbf{M}\vec{\mu})(t) + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t) , \quad t \in [0, T] , \quad \vec{\mu}(t) = 0 \quad \forall t < 0 , \quad [\text{Lecture} \rightarrow \text{Eq. (3.6.0.3)}]$$

$$\begin{aligned} \text{with } \mathbf{M} &:= \left[\int_{\Omega} b_h^i(x) b_h^j(x) dx \right]_{i,j=1}^N \in \mathbb{R}^{N,N}, \\ \mathbf{A} &:= \left[\int_{\Omega} \mathbf{grad} b_h^i(x) \cdot \mathbf{grad} b_h^j(x) dx \right]_{i,j=1}^N \in \mathbb{R}^{N,N}, \quad [\text{Lecture} \rightarrow \text{Eq. (3.6.0.4)}] \\ \vec{\varphi}(t) &:= \left[\int_{\Omega} f(x) b_h^i(x) dx \right]_{i=1}^N \in \mathbb{R}^N. \end{aligned}$$

As explained in [NumPDE course \rightarrow Section 4.1.3], in the current particular setting the Galerkin matrices and the right-hand side vector from [Lecture \rightarrow Eq. (3.6.0.4)] are

$$\mathbf{A} = \begin{bmatrix} \mathbf{T} & -\mathbf{I} & 0 & \cdots & \cdots & 0 \\ -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & & \vdots \\ 0 & -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & -\mathbf{I} & \mathbf{T} & -\mathbf{I} \\ 0 & \cdots & \cdots & 0 & -\mathbf{I} & \mathbf{T} \end{bmatrix}, \mathbf{T} := \begin{bmatrix} 4 & -1 & 0 & & & 0 \\ -1 & 4 & -1 & & & \vdots \\ 0 & -1 & 4 & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & -1 & 4 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 4 \end{bmatrix} \in \mathbb{R}^{n,n}, \quad (3.4.3)$$

$$\mathbf{M} = h^2 \mathbf{I}_N, \quad (3.4.4)$$

$$\vec{\varphi}(t) = h^2 [f(t, \mathbf{x}_i)]_{i=1}^N, \quad (3.4.5)$$

where $h := \frac{1}{n+1}$ is the meshwidth, and \mathbf{x}_i is the i -th interior node of \mathcal{M}_n . We remark that \mathbf{A} as in (3.4.3) is a so-called **Poisson matrix**. Obviously \mathbf{A} and \mathbf{M} are symmetric and we know that both matrices are also *positive definite*.

The file `fractionalheatequation.h` contains the definition of the following class

C++ code 3.4.6: Class definition of `SqrtsMplusA` \rightarrow [GITLAB](#)

```

2 class SqrtsMplusA {
3 public:
4     // Constructor initializes sparse matrix
5     SqrtsMplusA(unsigned int n, std::complex<double> s);
6     SqrtsMplusA(const SqrtsMplusA &) = delete;
7     SqrtsMplusA &operator=(const SqrtsMplusA &) = delete;
8     SqrtsMplusA(SqrtsMplusA &&) = default;
9     SqrtsMplusA &operator=(SqrtsMplusA &&) = default;
10    virtual ~SqrtsMplusA() = default;
11
12    // Matrix x vector
13    template <typename SCALAR>
14    [[nodiscard]] Eigen::VectorXcd eval(
15        const Eigen::Matrix<SCALAR, Eigen::Dynamic, 1> &v) const;
16    // Solve sparse linear system, LU decomposition on demand!
17    template <typename SCALAR>
18    [[nodiscard]] Eigen::VectorXcd solve(
19        const Eigen::Matrix<SCALAR, Eigen::Dynamic, 1> &y);
20
21 private:
22    std::unique_ptr<Eigen::SparseMatrix<std::complex<double>>> p_matrix_;
23    std::unique_ptr<Eigen::SparseLU<Eigen::SparseMatrix<std::complex<double>>>>
24        p_LUdec_{nullptr};
25
26 public:
27    static unsigned int solve_cnt;
28    static unsigned int ludec_cnt;

```

29 };

Objects of this class represent matrices $\mathbf{F}_s := \sqrt{s}\mathbf{M} + \mathbf{A}$, $s \in \mathbb{C} \setminus]-\infty, 0]$ and allow the computation of matrix-vector products $\mathbf{F}_s \mathbf{x}$, $\mathbf{x} \in \mathbb{C}^N$ via the `eval()` member function and the (direct) solution of linear systems of equations $\mathbf{F}_s \mathbf{x} = \mathbf{y}$ via the `solve()` member function. The member functions are implemented in `fractionalheatequation.cpp`.

(3-4.a) ☐ (20 min.) The temporal discretization of [Lecture → Eq. (3.6.0.3)] by means of the simplest convolution-quadrature scheme based on implicit Euler timestepping [Lecture → Def. 3.3.2.14] and with uniform timestep $\tau > 0$, $\tau := T/M$, $M \in \mathbb{N}$, leads to the recurrence relation for the basis expansion coefficient vectors $\vec{\mu}_j$, $j = 0, \dots, M$:

$$\mathbf{M} \left(\sum_{\ell=0}^n w_{n-\ell}^\tau \vec{\mu}_\ell \right) + \mathbf{A} \vec{\mu}_n = \vec{\varphi}(\tau n), \quad n = 0, \dots, M, \quad w_j^\tau := \left(\text{CQ}_\tau^{\text{IE}}(F) \right)_j, \quad j \in \mathbb{Z}. \quad (3.4.7)$$

In the file `fractionalheatequation.cpp` implement a C++ function

```
Eigen::VectorXd cqWeights(unsigned int M, double tau);
```

that returns the vector $\left[w_j^\tau \right]_{j=0}^M \in \mathbb{R}^{M+1}$.

SOLUTION for (3-4.a) → [3-4-1-0:fehs1.pdf](#) ▲

(3-4.b) ☐ (30 min.) In the file `fractionalheatequation.h` complete the implementation of the C++ function

```
template <typename SOURCEFN,
          typename RECORDER =
            std::function<void (const Eigen::VectorXd &)>>
Eigen::VectorXd evlMOT(
    SOURCEFN &&f, unsigned int n, double T, unsigned int M,
    RECORDER rec = [] (const Eigen::Vector2d &mu_n) {});
```

that solves (3.4.7) using direct elimination applied to the block-triangular linear system of equations, also known as **marching on in time (MOT)**, see [Lecture → Eq. (3.1.2.16)]. The argument `f` passes the right hand side source function $f = f(x, t)$ and must feature an evaluation operator **double operator** `()` (`Eigen::Vector2d`, **double**) **const**, `n` is the spatial resolution, `T` the final time T , and `M` provides the number of timesteps.

The argument `rec` supplies an object, which must possess a member operator **void operator** `()` (`const Eigen::VectorXd &`). That object “is called” with the current vector $\vec{\mu}_n$ in each timestep. For instance, it can be used to store all the intermediate state vectors.

In your implementation make use of the class **SqrtsMplusA**.

SOLUTION for (3-4.b) → [3-4-2-0:fhes2.pdf](#) ▲

(3-4.c) ☐ (10 min.) For your implementation of `evlMOT()` count the number of

- sparse LU decompositions,
- solves of triangular linear systems,
- matrix × vector products

that are carried out in dependence of M .

SOLUTION for (3-4.c) → 3-4-3-0:fhes3.pdf ▲

(3-4.d) 🕒 (120 min.) In `fractionalheatequation.h` realize an *efficient* C++ function

```
template <typename SOURCEFN,
          typename RECORDER = std::function<void (const
              Eigen::VectorXd &)>>
Eigen::VectorXd evlTriangToeplitz (
    SOURCEFN &&f, unsigned int n, double T, unsigned int L,
    RECORDER rec = [] (const Eigen::Vector2d &mu_n) {});
```

that uses the FFT-based algorithm outlined in [Lecture → § 3.1.5.21] to solve (3.4.7). The arguments are the same as for `evlMOT()` from Sub-problem (3-4.b) except for L , which specifies the number M of timesteps through $M := 2^L - 1$.

HIDDEN HINT 1 for (3-4.d) → 3-4-4-0:fheh4.pdf

SOLUTION for (3-4.d) → 3-4-4-1:fhe5s.pdf ▲

From [Lecture → Ex. 3.6.0.1] we know that [Lecture → Eq. (3.6.0.3)] can equivalently be written as

$$\mathbf{G}(\partial_t)\vec{\mu} = \vec{\varphi}(t) \quad \text{with} \quad \mathbf{G}(s) := \sqrt{s}\mathbf{M} + \mathbf{A}, \quad (3.4.19)$$

which means that we can represent the causal solution $t \mapsto \vec{\mu}(t)$ as a convolution

$$\vec{\mu}(t) = (\mathbf{G}^{-1}(\partial_t)\vec{\varphi})(t) \quad \text{with} \quad \mathbf{G}^{-1}(s) := (\sqrt{s}\mathbf{M} + \mathbf{A})^{-1}, \quad s \in \mathbb{C}^+. \quad (3.4.20)$$



Use implicit-Euler CQ with transfer function \mathbf{G}^{-1} and uniform timestep $\tau > 0$ to approximate $\vec{\mu}|_{\mathcal{G}_\tau}$:

$$\vec{\mu}|_{\mathcal{G}_\tau} \approx \text{CQ}_\tau^{\text{IE}}(\mathbf{G}^{-1}) * \vec{\varphi}|_{\mathcal{G}_\tau}. \quad (3.4.21)$$

(3-4.e) 🕒 (120 min.) Based on (3.4.21) implement an *efficient* (FFT-based) C++ function

```
template <typename SOURCEFN,
          typename RECORDER = std::function<void (const
              Eigen::VectorXd &)>>
Eigen::VectorXd evlASAOCQ (
    SOURCEFN &&f, unsigned int n, double T, unsigned int L,
    RECORDER rec = [] (const Eigen::Vector2d &mu_n) {});
```

with the same parameters as `evlTriangToeplitz` from Sub-problem (3-4.d) (and the same purpose) that relies on the “all-steps-in-one” convolution quadrature algorithm of [Lecture → § 3.4.3.22] (with $N = M$ in the notation of [Lecture → § 3.4.3.22]).

SOLUTION for (3-4.e) → 3-4-5-0:fheXs.pdf ▲

(3-4.f) 🕒 (30 min.) [depends on Sub-problem (3-4.d) and Sub-problem (3-4.e)]

Measure the runtimes of the different methods `evlMOT()`, `evlASAOCQ()`, and `evlTriangToeplitz()` for a fixed rather coarse grid ($n = 4$) and different numbers of timesteps $M := 2^L$. Plot them in a single plot and interpret the result.

SOLUTION for (3-4.f) → 3-4-6-0:.pdf ▲

End Problem 3-4 , 330 min.

Chapter 4

(Algebraic) Multigrid Methods

Problem 4-1: Stationary Linear Iterations

In [Lecture → Section 4.1.3] we learned about stationary linear iteration methods for solving linear systems of equations approximately. In this problem, we look at two specimens, the Gauss-Seidel method, as introduced in [Lecture → § 4.1.3.2], and the Jacobi method, which will be described in this problem itself.

The C++ coding in this problem relies on EIGEN.

▷ problem name/problem code folder: StationaryLinearIterations in [Code repository](#)

Throughout this problem let $\mathbf{A}_n \in \mathbb{R}^{N \times N}$, for $N := M^2$, $M := n - 1$, $n \in \mathbb{N}$, be the Poisson matrix as given in [Lecture → Eq. (4.1.1.23)].

(4-1.a)  Implement a C++ function

```
Eigen::SparseMatrix<double> poissonMatrix(unsigned int n);
```


which returns the Poisson matrix \mathbf{A}_n in EIGEN's sparse matrix format.

SOLUTION for (4-1.a) → [4-1-1-0:slis1.pdf](#) ▲

First, we study the so-called **Jacobi method**, a *parallel subspace correction method* for the iterative solution of linear systems of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$, $\mathbf{A} \in \mathbb{R}^{N,N}$, $\vec{\varphi} \in \mathbb{R}^N$. One step of the Jacobi method is defined by

$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \omega \operatorname{diag}(\mathbf{A})^{-1} (\vec{\varphi} - \mathbf{A}\vec{\mu}^{(k)}), \quad (4.1.2)$$

where $\operatorname{diag}(\mathbf{A})$ is the diagonal of \mathbf{A} and $\omega > 0$ is a so-called **relaxation parameter**, to be chosen “suitably”.

(4-1.b)  (45 min.) Derive the precise **asymptotic rate of linear convergence** for the Jacobi method (4.1.2) as a function of n and $\omega \in [0, 1]$ when applied to a linear systems of equations $\mathbf{A}_n\vec{\mu} = \vec{\varphi}$ for the Poisson matrix \mathbf{A}_n .

Remember that that “asymptotic rate of linear convergence” of a stationary linear iteration is the spectral radius of the error propagation matrix [Lecture → Eq. (4.1.3.14)], that is, the modulus of its largest eigenvalue.

HIDDEN HINT 1 for (4-1.b) → [4-1-2-0:slih2.pdf](#)

SOLUTION for (4-1.b) → [4-1-2-1:slis2.pdf](#) ▲

(4-1.c) ☒ (30 min.) [depends on Sub-problem (4-1.a)]

What is the optimal value for the relaxation parameter $\omega > 0$ for the Jacobi method when applied to solve a linear system of equations $\mathbf{A}_n \vec{\mu} = \vec{\varphi}$ for the Poisson matrix \mathbf{A}_n .

Here, “optimal” means that it yields the best asymptotic rate of linear convergence.

SOLUTION for (4-1.c) → [4-1-3-0:.pdf](#)

▲

Now we turn our attention to the **Gauss-Seidel method** the simple successive subspace correction iteration introduced in [Lecture → § 4.1.3.2]. For the linear system of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$, $\mathbf{A} \in \mathbb{R}^{N,N}$, $\vec{\varphi} \in \mathbb{R}^N$, it is defined by

$$\vec{\mu}^{(k+1)} = \vec{\mu}^{(k)} + \text{tril}(\mathbf{A})^{-1}(\vec{\varphi} - \mathbf{A}\vec{\mu}^{(k)}), \quad k \in \mathbb{N}_0, \quad [\text{Lecture} \rightarrow \text{Eq. (4.1.3.7)}]$$

where $\text{tril}(\mathbf{A})$ is the lower triangular part of \mathbf{A} .

(4-1.d) ☒ (20 min.) Following [Lecture → Code 4.1.3.3], implement an efficient C++ function

```
void gaussSeidel(const Eigen::SparseMatrix<double> &A,
                const Eigen::VectorXd &phi, Eigen::VectorXd &mu,
                int maxItr = 1000, double TOL = 1.0E-06);
```

that realizes a Gauss-Seidel based iterative solver for the linear system of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$. Here,

-
- A passes the sparse system matrix \mathbf{A} ,
- phi is the right-hand-side vector $\vec{\varphi}$,
- mu provides both the initial guess and the variable to return the approximate solution,
- maxItr is the maximum number of iterations, and
- TOL is the iteration error tolerance for the termination criterion.

HIDDEN HINT 1 for (4-1.d) → [4-1-4-0:sligsh1.pdf](#)

SOLUTION for (4-1.d) → [4-1-4-1:slis2.pdf](#)

▲

(4-1.e) ☒ Implement a C++ function

```
void gaussSeidelRate(const int n, double c, double TOL = 1.0E-03);
```

which determines the asymptotic rate of linear convergence of the Gauss-Seidel method when applied to a linear system of equations with the system matrix

$$\mathbf{X} := \mathbf{A}_n + c\mathbf{I}_N, \quad c > 0, \quad N = (n-1)^2.$$

Here \mathbf{I}_N is the $N \times N$ identity matrix and $c > 0$ is a given “reaction coefficient”.

Tabulate the rates for $c = 0, 1, 10$ and $n = 2^l$, $l = 1, 2, \dots, 10$. What do you observe?

HIDDEN HINT 1 for (4-1.e) → [4-1-5-0:sligshr.pdf](#)

SOLUTION for (4-1.e) → [4-1-5-1:sligs2.pdf](#)

▲

(4-1.f) ☒ (45 min.) Now we consider the Gauss-Seidel stationary linear iteration

$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \text{tril}(\mathbf{A})^{-1}(\vec{\varphi} - \mathbf{A}\vec{\mu}^{(k)}), \quad k \in \mathbb{N}_0, \quad (4.1.14)$$

for a generic *symmetric positive definite* (s.p.d.) matrix $\mathbf{A} \in \mathbb{R}^{N,N}$. Show that the error recursion for iteration (4.1.14) is a **contraction** with respect to the energy norm $\|\cdot\|_A$ induced by \mathbf{A} :

$$\left\| \vec{\mu}^{(k+1)} - \vec{\mu}^* \right\|_A < q \left\| \vec{\mu}^{(k)} - \vec{\mu}^* \right\|_A \quad \text{for some } q \in [0, 1[. \quad (4.1.15)$$

The Gauss-Seidel iteration converges for any linear system of equations with s.p.d. coefficient matrix.

HIDDEN HINT 1 for (4-1.f) → [4-1-6-0:slizh1.pdf](#)

HIDDEN HINT 2 for (4-1.f) → [4-1-6-1:slizh2.pdf](#)

HIDDEN HINT 3 for (4-1.f) → [4-1-6-2:slizh3.pdf](#)

SOLUTION for (4-1.f) → [4-1-6-3:slizsol.pdf](#) ▲

End Problem 4-1 , 140 min.

Problem 4-2: Matrix-Free Multigrid for Scalar Dirichlet BVPs

This problem is about the (memory-)efficient implementation of the geometric multigrid iteration of [Lecture → Section 4.2.4] for the special setting of a linear scalar elliptic boundary value problem on the unit square discretized by means of bi-linear Lagrangian finite elements on a tensor product mesh.

Relies on [Lecture → Ex. 4.1.1.24] and [Lecture → Section 4.2.4]. Also assumes knowledge of [Lecture → Rem. 4.1.3.18]. Implementation in C++ is based on EIGEN.
 ▷ problem name/problem code folder: MFMG in [Code repository](#)

For a given constant $c \in \mathbb{R}$ we consider the Dirichlet boundary value problem

$$-\Delta u + cu = f \quad \text{in } \Omega :=]0,1[^2, \quad u = 0 \quad \text{on } \partial\Omega. \quad (4.2.1)$$

As in [Lecture → Ex. 4.1.1.24] we employ a Galerkin finite-element discretization by means of bi-linear Lagrangian finite elements on a tensor-product grid \mathcal{M}_L with $M+1$ cells in each direction, $M = 2^L - 1$, $L \in \mathbb{N}$. We call L the “level” of the finite element mesh. All occurring integrals are computed approximately by means of the local trapezoidal quadrature rule [Lecture → Eq. (4.1.1.27)].

(4-2.a) ☐ (30 min.) Compute the **element matrix** \mathbf{A}_K for a generic square $K := [0, h]^2$, using the standard local shape functions and numbering its vertices in the order $\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ as in [Lecture → Fig. 158].

SOLUTION for (4-2.a) → [4-2-1-0:mfmgs1.pdf](#) ▲

(4-2.b) ☐ (30 min.) Assume a lexicographic numbering of the interior grid nodes as in [Lecture → Ex. 4.1.1.24]. Let $\mathbf{A} \in \mathbb{R}^{N,N}$, $N := M^2$, stand for the finite-element Galerkin matrix. Find an $M \times M$ **tri-diagonal** matrix \mathbf{T} such that

$$\mathbf{A} = \mathbf{I}_M \otimes \mathbf{T} + \mathbf{T} \otimes \mathbf{I}_M. \quad (4.2.3)$$

Here, \otimes stands for the **Kronecker product of matrices** [NumCSE course → Def. 1.4.3.7] and \mathbf{I}_M is the $M \times M$ identity matrix.

SOLUTION for (4-2.b) → [4-2-2-0:mfmgs2.pdf](#) ▲

We denote by \mathcal{G}_ℓ the vector space of **grid functions** based on the interior nodes of \mathcal{M}_ℓ , that is, the space of functions

$$\left\{ h \begin{bmatrix} i \\ j \end{bmatrix} : i, j \in \{1, \dots, M\} \right\} \mapsto \mathbb{R}, \quad M := 2^\ell - 1, \quad h := \frac{1}{M+1}.$$

Throughout this problem we represent grid functions $\underline{\mu} \in \mathcal{G}_\ell$ as matrices of size $(M+2) \times (M+2)$ stored as objects of type **Eigen::MatrixXd**. We adopt the convention that, if \mathbf{M} is the matrix storing the grid function $\underline{\mu}$, then

$$\mathbf{M}(i, j) = \begin{cases} \mu_{ij} := \underline{\mu} \left(h \begin{bmatrix} i \\ j \end{bmatrix} \right), & \text{if } i, j \in \{1, \dots, M\}, \\ 0, & \text{if } i = 0, M+1 \vee j = 0, M+1, \end{cases} \quad i, j \in \{0, \dots, M+1\}, \quad (4.2.6)$$

where C++ indexing is used: we equip the matrix with a “zero rim”. Moreover, we write \mathbf{A}_ℓ for the Galerkin matrix on the grid \mathcal{M}_ℓ with M^2 interior nodes, $M := 2^\ell - 1$, and meshwidth $h := 2^{-\ell}$.

In this problem you are supposed to implement the following class


C++ code 4.2.7: Class implementing basic multigrid functionality Get it on GitLab (`mfmfg.h`).

```

2 using GridFunction = Eigen::MatrixXd;
3 class DirichletBVPMultiGridSolver {
4 public:
5     DirichletBVPMultiGridSolver(double c, unsigned int L)
6         : c_(c), L_(L) {}
7     [[nodiscard]] GridFunction applyGridOperator(unsigned int level,
8                                                  const GridFunction &mu) const;
9     [[nodiscard]] GridFunction directSolve(unsigned int level,
10                                            const GridFunction &phi) const;
11 void sweepGaussSeidel(unsigned int level, GridFunction &mu,
12                       const GridFunction &phi) const;
13 [[nodiscard]] GridFunction residual(unsigned int level,
14                                    const GridFunction &mu,
15                                    const GridFunction &phi) const;
16 [[nodiscard]] GridFunction prolongate(unsigned int level,
17                                      const GridFunction &gamma) const;
18 [[nodiscard]] GridFunction restrict(unsigned int level,
19                                    const GridFunction &rho) const;
20 GridFunction multigridIteration(const GridFunction &mu,
21                               const GridFunction &phi,
22                               unsigned int L0 = 2) const;
23
24 private:
25     const double c_; // Reaction coefficient
26     const unsigned int L_; // Level of the finest grid
27 };

```

The constructor's arguments provide the reaction coefficient $c \in \mathbb{R}$ and the level L of the tensor-product finite-element mesh. The member functions will be described in the following sub-problems.

(4-2.c)  (30 min.) In the file `mfmfg.cpp` implement the member function `applyGridOperator()`, which computes $\mathbf{A}_\ell \vec{\mu}$ (ℓ passed as `level`) as a **GridFunction** $\in \mathcal{G}_\ell$ when given its argument also in that **GridFunction** format.


SOLUTION for (4-2.c) → [4-2-3-0:mfmgs3.pdf](#) ▲

(4-2.d)  (60 min.) [depends on Sub-problem (4-2.b)]

In the file `mfmfg.cpp` implement the member function `directSolve()`, which employs EIGEN's built-in sparse elimination solver to compute $\mathbf{A}_\ell^{-1} \vec{\phi}$ as a **GridFunction** $\in \mathcal{G}_\ell$ when given its argument also in **GridFunction** format.

HIDDEN HINT 1 for (4-2.d) → [4-2-4-0:mfmgh4.pdf](#)

SOLUTION for (4-2.d) → [4-2-4-1:mfmgs4.pdf](#) ▲

(4-2.e)  (20 min.) In the file `mfmfg.cpp` realize the member function `sweepGaussSeidel()`, which updates the **GridFunction** passed through its `mu` argument through a single Gauss-Seidel step for $\mathbf{A}_\ell \vec{\mu}_\ell = \vec{\phi}_\ell$. Lexicographic ordering of the mesh nodes should be used. The parameter `phi` provides a **GridFunction** representation of the right-hand-side vector $\vec{\phi}$.

The `level` ($= \ell$) argument specifies that the target linear system should be that on the grid \mathcal{M}_ℓ with $(2^\ell - 1)^2$ interior nodes.

SOLUTION for (4-2.e) → [4-2-5-0:mfmgs5.pdf](#) ▲

(4-2.f)  (20 min.) In the file `mfmfg.cpp` write the member function `residual()`, which

computes the **residual** $\vec{\phi}_\ell - \mathbf{A}_\ell \vec{\mu}_\ell$ as a **GridFunction** on level ℓ . The arguments `mu`, `phi` are also supposed to represent **GridFunction**s on level ℓ (, which is passed as `level`).

SOLUTION for (4-2.f) → [4-2-6-0:mfmgs6.pdf](#) ▲

(4-2.g) 🕒 (30 min.) In the file `mfmng.cpp` complete the implementation of the member function `prolongate()`, which computes the **prolongation** of a **GridFunction** on level $\ell - 1$ to a **GridFunction** on level ℓ (`= level`).

HIDDEN HINT 1 for (4-2.g) → [4-2-7-0:mfmgh7.pdf](#)

SOLUTION for (4-2.g) → [4-2-7-1:mfmgs4.pdf](#) ▲

(4-2.h) 🕒 (15 min.) [depends on Sub-problem (4-2.g)]

In the file `mfmng.cpp` code the member function `restrict()`, which computes the **restriction** of a (residual-type) **GridFunction** from level ℓ (`= level`) to a **GridFunction** on level $\ell - 1$.

HIDDEN HINT 1 for (4-2.h) → [4-2-8-0:mfmgh8.pdf](#)

SOLUTION for (4-2.h) → [4-2-8-1:mfmgs8.pdf](#) ▲

(4-2.i) 🕒 (60 min.) The member function `multigridIteration()` of **DirichletBVPMultiGridSolver** implements a single **multigrid V-cycle** iteration for $\mathbf{A}_L \vec{\mu}_L = \vec{\phi}_L$ as outlined in [Lecture → Code 4.2.4.5] (with `max_n_steps = 1`). In contrast to that code use one Gauss-Seidel sweep for **pre-smoothing** and another one for **post-smoothing**.

The argument `phi` passes the right-hand-side vector as a **GridFunction** object, and `mu` contains the current guess for the solution (also in the format of a **GridFunction**). The `mu` argument is also used to return the updated guess. Finally, `L0` specifies the grid level on which a direct solver should be used.

HIDDEN HINT 1 for (4-2.i) → [4-2-9-0:mfmgh9.pdf](#)

SOLUTION for (4-2.i) → [4-2-9-1:mfmgs9.pdf](#) ▲

(4-2.j) 🕒 [depends on Sub-problem (4-2.i)]

What is the maximum amount of memory required by your implementation of the member function `multigridIteration()` as a function of the level L ? Give a rather precise formula. The memory consumption for the direct solver on level $L_0=2$ can be neglected.

SOLUTION for (4-2.j) → [4-2-10-0:mfmgs10.pdf](#) ▲

(4-2.k) 🕒 (30 min.) [depends on Sub-problem (4-2.i)]

In the file `mfmng.cpp` implement the C++ function

```
double estimateMGConvergenceRate(double c, unsigned int L, double
    tol = 1.0E-3);
```

that uses the power method of [Lecture → Code 4.1.3.21] to estimate the rate of linear convergence of the multigrid V-cycle implemented as member function `multigridIteration()` of **DirichletBVP-MultiGridSolver**. The argument `c` passes the reaction coefficient c , L the level of the finest mesh, and `tol` the relative tolerance, with which the rate should be computed.

Use the grid function with value `1` in all interior nodes as starting vector and $L_0=2$.

SOLUTION for (4-2.k) → [4-2-11-0:mfmgsAs.pdf](#) ▲

(4-2.l) 🕒 (20 min.) [depends on Sub-problem (4-2.k)]

Write a C++ function (in `mfmng.cpp`)

```
void tabulateMGConvergenceRate();
```

that prints a table of convergence rates (estimated to three digits) for reaction coefficient $c \in \{-40, -20, -10, -1, 0, 1, 10, 20, 40\}$ and levels $L \in \{6, 7, 8, 9, 10, 11\}$.

Describe the trends reflected in the data.

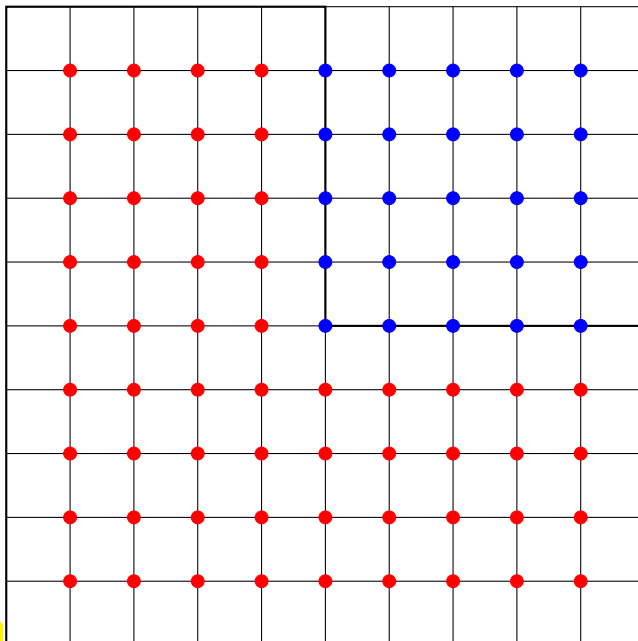
SOLUTION for (4-2.l) → [4-2-12-0:mfmgb.pdf](#)



(4-2.m) 🕒 (120 min.) [depends on Sub-problem (4-2.c)–Sub-problem (4-2.i)]

We are now interested in an implementation for the boundary value problem (4.2.1) on the **L-shaped domain** $\Omega :=]0, 1[^2 \setminus]\frac{1}{2}, 1[^2$.

The task is to make all member functions of **DirichletBVPMultiGridSolver** act appropriately on this different geometry. To that end, copy your existing codes into the file `mfmglshape.cpp` and extend the member functions of the original namespace to versions realizing the boundary value problem formulated on the **L-shaped domain**.



◁ Active interior nodes (●) and inactive interior nodes (●) of a tensor-product mesh of $]0, 1[^2$, when restricted to the L-shaped domain.



Confine all operations to active nodes, set all values in inactive nodes to zero.

In the double-loop implementation that underlies most member function this amounts to checking whether a pair of indices corresponds to an inactive node. In this case all updates/modifications of values are skipped.

Fig. 12

HIDDEN HINT 1 for (4-2.m) → [4-2-13-0:mfmgh.pdf](#)

SOLUTION for (4-2.m) → [4-2-13-1:.pdf](#)



End Problem 4-2, 465 min.

Problem 4-3: Galerkin Construction of Coarse-Grid Matrices

This problem is concerned with the algorithmic realization of the so-called **Galerkin construction** of the coarse-grid matrix, which is universally employed in algebraic multigrid methods, see [Lecture → § 4.3.1.2].

The implementations requested in this problem will rely on EIGEN. Knowledge about EIGEN's types and methods meant for dealing with sparse matrices is essential and you should refer to [NumCSE course → Section 2.7.2].

▷ problem name/problem code folder: GalerkinConstruction in [Code repository](#)

We consider the 2nd-order scalar elliptic model boundary value problem on the unit square:

$$-\Delta u = f \quad \text{in } \Omega :=]0,1[^2, \quad u = 0 \quad \text{on } \partial\Omega. \tag{4.3.1}$$

We perform standard Galerkin finite element discretization of (4.3.1) based on lowest-order Lagrangian finite elements [NumPDE course → Section 2.6]. We study two variants:

❶ We rely on a uniform tensor-product quadrilateral mesh \mathcal{M}_M with M cells in each direction, meshwidth $h := 1/M$.

As Galerkin trial and test space we rely on the space

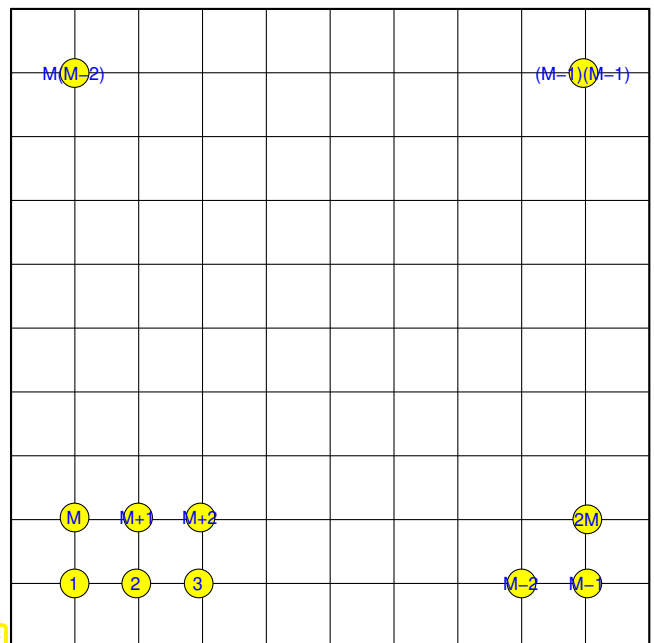
$$\mathcal{S}_{1,0}^0(\mathcal{M}_M) := \mathcal{S}_1^0(\mathcal{M}_M) \cap H_0^1(\Omega)$$

of piecewise *bi-linear* continuous functions on \mathcal{M}_M [NumPDE course → Ex. 2.6.2.1], which vanish on $\partial\Omega$.

As locally supported global basis functions we use the standard bi-linear tent functions [NumPDE course → Fig. 114] associated with the interior nodes of \mathcal{M} :

$$\blacktriangleright \quad \dim \mathcal{S}_{1,0}^0(\mathcal{M}_M) = N := (M - 1)^2.$$

We employ **lexicographic numbering** of the basis functions from left to right, bottom to top.



As in [Lecture → Ex. 4.1.1.24] in the assembly of the final finite-element linear system $\mathbf{A}\vec{\mu} = \vec{\varphi}$ we use the local trapezoidal quadrature rule [Lecture → Eq. (4.1.1.27)] for the evaluation of any integral over any cell of the mesh. This will render $\mathbf{A} \in \mathbb{R}^{N,N}$ the Poisson matrix [Lecture → Eq. (4.1.1.23)].

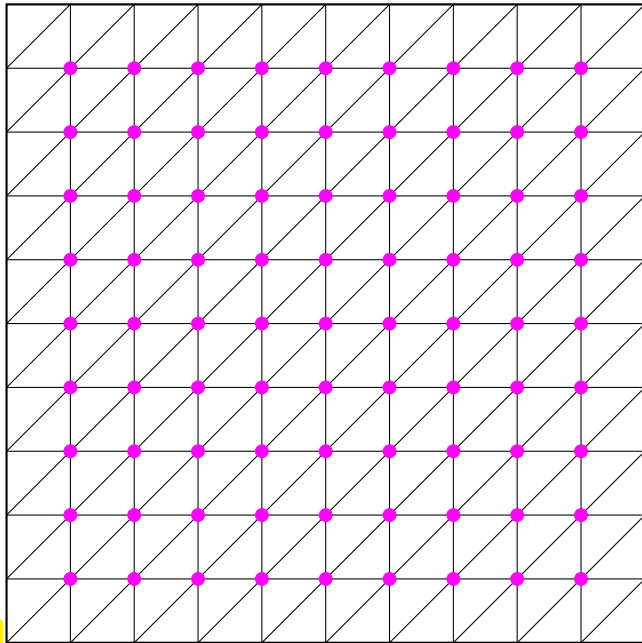


Fig. 14

② As in [Lecture → Ex. 4.1.1.22] we use a uniform tensor-product triangular mesh \mathcal{M}_M with M cells in each direction as sketched beside.

On \mathcal{M}_M we consider the lowest-order Lagrangian finite element space $\mathcal{S}_{1,0}^0(\mathcal{M}_M)$ of \mathcal{M} -piecewise linear finite-element functions that vanish on $\partial\Omega$???. Again, we have $\dim \mathcal{S}_{1,0}^0(\mathcal{M}_M) = (M-1)^2$ and can use the regular tent function basis [NumPDE course → Section 2.4.3].

As explained in [Lecture → Ex. 4.1.1.22] this will again yield the Poisson matrix as the (exact) Galerkin matrix.

The focus on coarse-grid correction in a two-grid setting [Lecture → ??]. As “fine grid” we use \mathcal{M}_M and as coarse grid \mathcal{M}_m , $m := M/2$, M assumed to be even. That is \mathcal{M}_M is spawned by the regular refinement of \mathcal{M}_m and the finite-element spaces are *nested*: $\mathcal{S}_{1,0}^0(\mathcal{M}_m) \subset \mathcal{S}_{1,0}^0(\mathcal{M}_M)$.

(4-3.a) 🕒 (45 min.) In the file `galerkinconstruction.cpp` implement the C++ function

```
Eigen::SparseMatrix<double, Eigen::RowMajor>
prolongationMatrix(unsigned int M, bool bilinear = true)
```

that computes the **prolongation matrix P** for discretization ①. The argument `M` passes the number of grid cells in each direction and the flag `bilinear` can be ignored at this point.

SOLUTION for (4-3.a) → [4-3-1-0:gcs1.pdf](#) ▲

(4-3.b) 🕒 (20 min.) [depends on Sub-problem (4-3.a)]

Now solve Sub-problem (4-3.a) for finite-element discretization ②. This case should also be dealt with by `prolongationMatrix()`, but with the flag `bilinear` set to **false**.

SOLUTION for (4-3.b) → [4-3-2-0:cos2.pdf](#) ▲

(4-3.c) 🕒 In the file `galerkinconstruction.cpp` write an efficient C++ function

```
Eigen::SparseMatrix<double> buildAH(
    const Eigen::SparseMatrix<double> &A,
    const Eigen::SparseMatrix<double, Eigen::RowMajor> &P);
```

that computes the result of the Galerkin construction

$$\mathbf{A}_H := \mathbf{P}^\top \mathbf{A} \mathbf{P}, \quad (4.3.7)$$

in sparse-matrix format when provided with the fine-grid (Galerkin) matrix **A** and the prolongation matrix **P**.

You must not invoke any of EIGEN's built-in functions for multiplying sparse matrices!

HIDDEN HINT 1 for (4-3.c) → [4-3-3-0:gacoh2a.pdf](#)

SOLUTION for (4-3.c) → [4-3-3-1:cos2a.pdf](#) ▲

(4-3.d) [depends on Sub-problem (4-3.a) & Sub-problem (4-3.b)]

The following code

C++ code 4.3.9: Testing of construction of A_H [Get it on GitLab \(galerkinconstruction.cpp\)](#)

```

2  const unsigned int N = 8;
3  const unsigned int n = N / 2;
4
5  const Eigen::SparseMatrix<double> Ah = poissonMatrix(N);
6  const Eigen::SparseMatrix<double> AH = poissonMatrix(n);
7  const auto P_L = prolongationMatrix(N, false);
8  const auto P_B = prolongationMatrix(N, true);
9  const Eigen::SparseMatrix<double> AH_L = buildAH(Ah, P_L);
10 const Eigen::SparseMatrix<double> AH_B = buildAH(Ah, P_B);
11
12 const double diff_L = (AH_L - AH).norm();
13 const double diff_B = (AH_B - AH).norm();
14 std::cout << "(AH_L - AH).norm() = " << diff_L
15           << ",\n(AH_B - AH).norm() = " << diff_B << std::endl;

```

outputs

```

1  (AH_L - AH).norm() = 0,
2  (AH_B - AH).norm() = 4

```

Give an explanation.

SOLUTION for (4-3.d) → [4-3-4-0:.pdf](#)

**(4-3.e)** [depends on Sub-problem (4-3.c)]

In `galerkinconstruction.cpp` you find the following code.

C++ code 4.3.10: EIGEN-based construction of coarse-grid matrix $A_H = P^T A P$
[Get it on GitLab \(galerkinconstruction.cpp\)](#)

```

2  Eigen::SparseMatrix<double> buildAH_eigen(
3      const Eigen::SparseMatrix<double> &A,
4      const Eigen::SparseMatrix<double, Eigen::RowMajor> &P) {
5      Eigen::SparseMatrix<double> AH;
6      AH = P.transpose() * A * P;
7      return AH;
8  }

```

It implements the Galerkin construction of a coarse-grid matrix enlisting EIGEN's multiplication of sparse matrices.

For finite-element discretization ① tabulate the measured runtimes of `buildAH_eigen()` and of your implementation of `buildAH()` with

- the the prolongation matrix P generated by `prolongationMatrix()`,
- and the fine-grid matrix obtained by calling the function `poissonMatrix()`,

and for $M = 2^L$, $L = 3, 4, \dots, 13$. To that end complete the code of the C++ function

```

template <typename SEQ>
void tabulateRuntimes(SEQ &&M_vals);

```

in the file `galerkinconstruction.h`. The type **SEQ** must allow the traversal of contents in a **for** loop. This function is called from `main()`. Adhere to the usual best practices of runtime measurements.

What do you observe and what conclusions do you draw from the data?

HIDDEN HINT 1 for (4-3.e) → [4-3-5-0:gfh7.pdf](#)

SOLUTION for (4-3.e) → [4-3-5-1:.pdf](#) ▲

End Problem 4-3, 65 min.

Problem 4-4: The Ruge-Stüben Algebraic Multigrid Method: A Case Study

For the special case of a “structured mesh” this problem deals with the construction of the C/F-splitting and of the prolongation matrix for the classical algebraic multigrid method first proposed by Ruge and Stüben in [RUS87] and presented in [Lecture → Section 4.3].

This problem assumes full command of the material of [Lecture → Section 4.3].

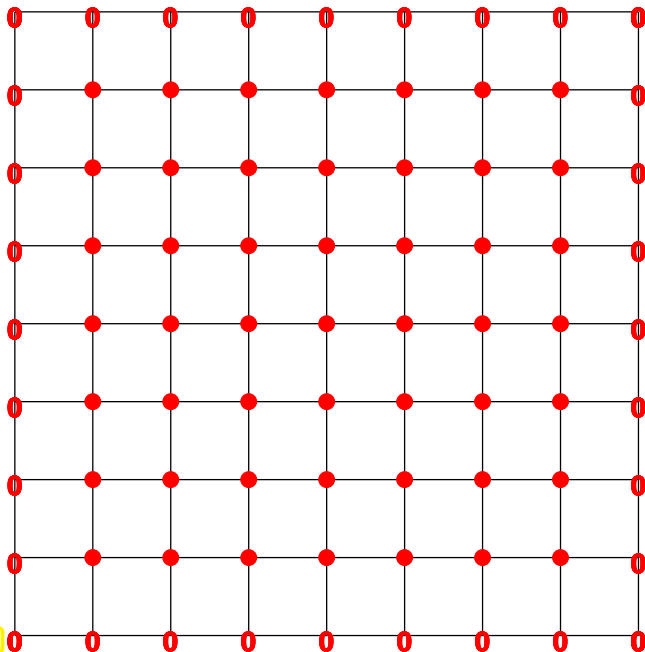


Fig. 17

In the first part of this project we consider a finite-difference linear system of equations on the interior nodes of an equidistant 8×8 tensor-product mesh of the unit square domain $\Omega =]0, 1[^2$.

The coefficient matrix is described by the **9-point stencil** [Lecture → § 4.1.1.31]

$$A \sim \begin{bmatrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & 1 & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{bmatrix}_h \tag{4.4.1}$$

Zero Dirichlet boundary conditions are imposed, that is, the nodes on $\partial\Omega$ carry the value 0.

(4-4.a) ☐ (30 min.) Characterize the sets $\mathcal{S}(i)$ and $\mathcal{S}(i)^*$ as introduced in [Lecture → § 4.3.3.1] for the central node i located at $\begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}$.

SOLUTION for (4-4.a) → 4-4-1-0:cfs0.pdf ▲

(4-4.b) ☐ (45 min.) “On paper” execute the algorithm from [Lecture → ??] for the construction of the C/F splitting in the set-up phase of the Ruge-Stüben algebraic multigrid methods. Use the parameters $\tau_C = \frac{1}{4}$ [Lecture → Def. 4.3.3.2] and $\delta = \frac{3}{2}$ [Lecture → § 4.3.3.8].

Start with the “central node” located at $\begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}$ as first C-node. Then, in the figures below, depict the current sets \mathcal{U} , \mathcal{C} , and \mathcal{F} at the end of the while-loop in [Lecture → ??] (Line 12). Mark C-nodes with magenta, F-nodes with cyan, and leave yet undecided nodes blank.

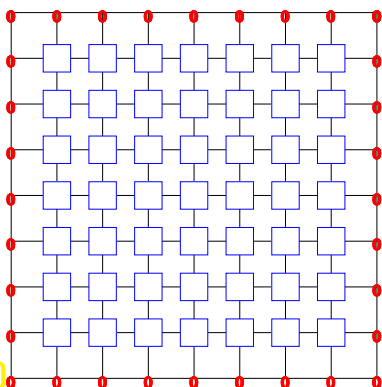


Fig. 19

While-loop executed once

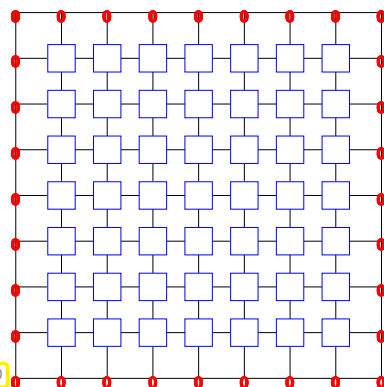


Fig. 20

While-loop executed twice

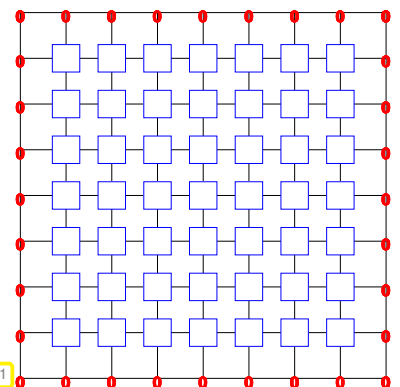
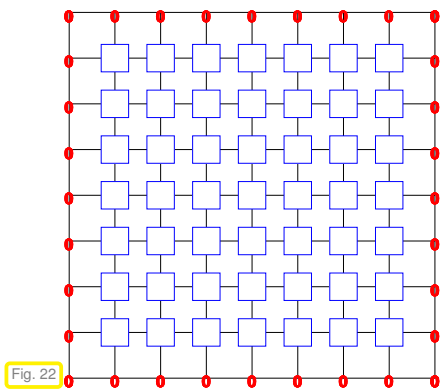
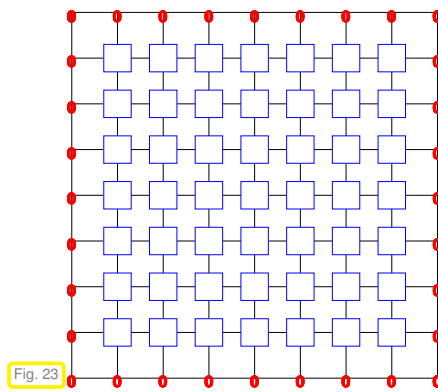
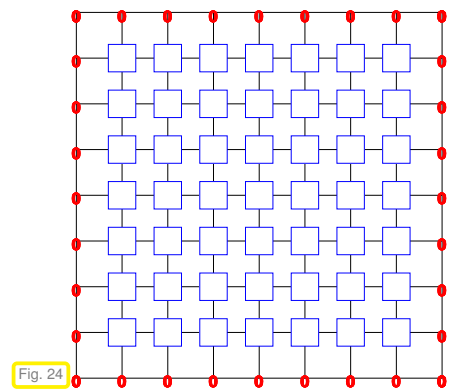
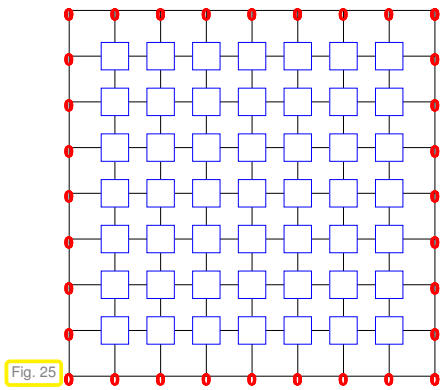
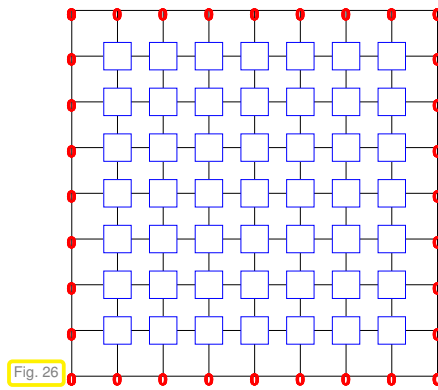
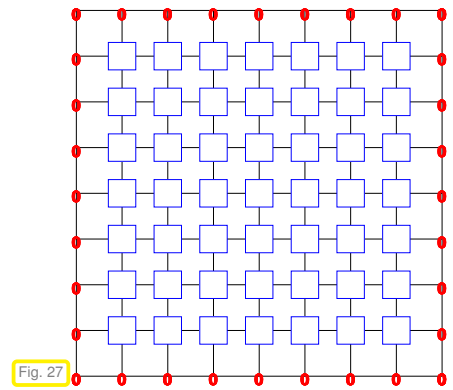


Fig. 21

While-loop executed thrice

While-loop executed $4 \times$ While-loop executed $5 \times$ While-loop executed $6 \times$ While-loop executed $7 \times$ While-loop executed $8 \times$ While-loop executed $9 \times$

SOLUTION for (4-4.b) → [4-4-2-0:cfs1.pdf](#) ▲

(4-4.c) ☹️ (45 min.) [depends on Sub-problem (4-5.a)]

The C/F-splitting built in Sub-problem (4-5.a) will relegate the node located at $\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2}+h \end{bmatrix}$, $h = \frac{1}{8}$, to an F-node. Based on [Lecture → Eq. (4.3.4.10)] compute the row of the prolongation matrix \mathbf{P} corresponding to that node.

Since the numbering of C-nodes has not been specified, a stencil description of that row of \mathbf{P} has to be given.

SOLUTION for (4-4.c) → [4-4-3-0:cfs4.pdf](#) ▲

(4-4.d) ☹️ (20 min.) [depends on Sub-problem (4-5.b)]

Sketch the sparsity pattern (“spy plot”) of the coarse grid matrix \mathbf{A}_H obtained from \mathbf{A} by the Galerkin construction based on the prolongation matrix constructed before.

SOLUTION for (4-4.d) → [4-4-4-0:cfs3.pdf](#) ▲

(4-4.e) ☹️ (45 min.) [depends on Sub-problem (4-5.b)]

Using lexicographic numbering of the C-nodes (“from bottom left to top right”) compute the diagonal entry $(\mathbf{A}_H)_{22}$.

SOLUTION for (4-4.e) → [4-4-5-0:.pdf](#) ▲

End Problem 4-4, 185 min.

Problem 4-5: The Ruge-Stüben Algebraic Multigrid Method: Implementation

[Lecture → Section 4.3] presented the classical algebraic multigrid method first proposed by Ruge and Stüben in [RUS87] and further refined later. This problem deals with the implementation of the various components of that methods, supplementing with concrete code the explanations given in [Lecture → Section 4.3].

This problem assumes full command of the material of [Lecture → Section 4.3]. Implementation will rely on EIGEN.

▷ problem name/problem code folder: CFSplit in [Code repository](#)

The following class is supposed to implement all the components of the Ruge-Stüben AMG method for solving the sparse linear system of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$, $\mathbf{A} \in \mathbb{R}^{N,N}$ s.p.d., as presented in [Lecture → Section 4.3].

C++ code 4.5.1: Definition of RugeStuebenAMG Get it on [GitLab](#) (cfsplit.h).

```

2  class RugeStuebenAMG {
3  public:
4  using nodeidx = unsigned int; // Number of a degree of freedom = node
5  enum NodeFlag : int { UNDECIDED = 0, FINE = 1, COARSE = -1 };
6  // Standard constructors
7  RugeStuebenAMG() = delete;
8  RugeStuebenAMG(const RugeStuebenAMG &) = delete;
9  RugeStuebenAMG(RugeStuebenAMG &&) = default;
10 RugeStuebenAMG &operator=(const RugeStuebenAMG &) = delete;
11 RugeStuebenAMG &operator=(RugeStuebenAMG &&) = default;
12 // Constructor performing the setup phase for the AMG method
13 template <typename RECORDER =
14         std::function<void(const Eigen::SparseMatrix<double> &,
15                           const Eigen::SparseMatrix<double> &,
16                           const std::vector<NodeFlag> &)>>
17 explicit RugeStuebenAMG(
18     const Eigen::SparseMatrix<double> &A, unsigned int min_mat_size = 10,
19     double tauC = 0.25, double sigma = 1.5,
20     RECORDER &&rec = [] (const Eigen::SparseMatrix<double> & /*AH*/,
21                         const Eigen::SparseMatrix<double> & /*P*/,
22                         const std::vector<NodeFlag> & /*flags*/ -> void {});
23 virtual ~RugeStuebenAMG() = default;
24
25 // Iteration operator for AMG
26 void iterate(const Eigen::VectorXd &phi, Eigen::VectorXd &mu) const;
27
28 Eigen::SparseMatrix<double> getP(unsigned int idx) { return Pmats_[idx]; }
29
30 private:
31 // Determine sets of strongly connected nodes
32 [[nodiscard]] std::pair<std::vector<std::vector<nodeidx>>,
33                       std::vector<std::vector<nodeidx>>>
34 getStrongConn(const Eigen::SparseMatrix<double> &A, double tauC) const;
35 // Set smoothable node to F-nodes
36 void setSmoothable(const Eigen::SparseMatrix<double> &A, double sigma,
37                  std::vector<NodeFlag> &nodeflags) const;
38 // Compute connectivity measure for all nodes
39 [[nodiscard]] std::vector<unsigned int> connectivityMeasure(
40     const std::vector<NodeFlag> &nodeflags,
41     const std::vector<std::vector<nodeidx>> &Sj_star) const;
42 // Set flags for nodes defining C/F splitting

```

```

43  [[nodiscard]] std::vector<NodeFlag> CFsplit(
44      const Eigen::SparseMatrix<double> &A,
45      const std::vector<std::vector<nodeidx>> &Sj_star) const;
46  // Compute the AMG prolongation matrix
47  [[nodiscard]] Eigen::SparseMatrix<double, Eigen::RowMajor> computeP(
48      const Eigen::SparseMatrix<double> &A,
49      const std::vector<std::vector<nodeidx>> &Si,
50      const std::vector<NodeFlag> &nodeflags) const;
51
52  // Finest level: level of linear system of equations
53  unsigned int L_;
54  // Sigma
55  double sigma_;
56  // tau
57  double tauC_;
58  // Doubly linked list (length L) of system matrices on all levels
59  std::vector<Eigen::SparseMatrix<double>> Amats_;
60  // Doubly linked list (length L-1) of prolongation matrices
61  std::vector<Eigen::SparseMatrix<double>> Pmats_;
62  // LU decomposition of matrix on coarsest level
63  Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>>
64  CoarseLU_;
65  };

```

The constructor takes care of the setup phase of Ruge-Stüben AMG, which involves

- to determine the coarse-fine (C/F) splitting of the degrees of freedom (“nodes”) [Lecture → Code 4.3.3.12] successively on all levels $\ell = 1, \dots, L$, L the total number of levels and also the level of the initial linear system of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$,
- to build the AMG prolongation matrices $\mathbf{P}_{\ell-1,\ell} = \mathbb{R}^{N_\ell, N_{\ell-1}}$ [Lecture → Section 4.3.4], $N_\ell \in \mathbb{N}$ the number of degrees of freedom/nodes on level ℓ , $\ell = 1, \dots, L$, $N_L := N$.
- to compute the system matrices \mathbf{A}_ℓ , $\ell = 1, \dots, L-1$ on coarser levels by means of the **Galerkin construction** [Lecture → ??]: $\mathbf{A}_{\ell-1} = \mathbf{P}_{\ell-1,\ell}^\top \mathbf{A}_\ell \mathbf{P}_{\ell-1,\ell}$.

(4-5.a) ☐ (15 min.) Study [Lecture → § 4.3.3.10] and understand how the C/F-splitting is build in [Lecture → Code 4.3.3.12], which adopts a strictly two-grid perspective. ▲

The member variables of the class **RugeStuebenAMG** have the following functions:

- L_+ gives the number $L \in \mathbb{N}$ of levels on which the AMG method is based. That number has the same meaning is as in the geometric multigrid method. Note that the numbering of levels starts from zero (coarsest level).
- $Amats_+$ is a vector of length L_++1 containing the matrices \mathbf{A}_ℓ , $\ell = 0, \dots, L$. $Amats_+[L_+]$ is the matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ passed to the constructor.
- $Pmats_+$ is a vector of length L_+ which contains the prolongation matrices $\mathbf{P}_{\ell,\ell+1}$, $\ell = 0, \dots, L-1$.
- $CoarseLU_+$ stores the precomputed sparse LU-decomposition of \mathbf{A}_0 , which is used for the direct solution of the LSE on the coarsest level. Refer to the [EIGEN documentation](#), where you can find the following sample code:

C++ code 4.5.2: Sample code for using EIGEN’s sparse LU decomposition

```

1  Eigen::VectorXd x(n), b(n);
2  Eigen::SparseMatrix<double> A;
3  Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> >

```

```

    solver;
4 // fill A and b;
5 // Compute the ordering permutation vector from the structural
  pattern of A
6 solver.analyzePattern(A);
7 // Compute the numerical factorization
8 solver.factorize(A);
9 //Use the factors to solve the linear system
10 x = solver.solve(b);

```

On each level ℓ the N_ℓ nodes are supposed to be indexed from 0 to $N_\ell - 1$. The sets \mathcal{C} and \mathcal{F} of C-nodes and F-nodes, respectively, are encoded by an object (x) of type `std::vector<NodeFlag>`, where **Node Flag** is a flag type for the selection status of a node:

$$\text{node } i \in \mathcal{C} \iff x[i] == \text{COARSE} \quad , \quad \text{node } i \in \mathcal{F} \iff x[i] == \text{FINE} .$$

The class **RugeStuebenAMG** should rely on the following simplified version of [Lecture → Code 4.3.3.12]:

Pseudocode 4.5.3: Construction of coarse-fine splitting based on matrix **A**

```

1  [C, F] = CFsplit(matrix A ∈ ℝN,N) {
2    F := {i ∈ {1,...,N} : |(A)ii| ≥ σ ∑k≠i |(A)ik|}; //
3
4    C := ∅; // no C-nodes yet
5    U := {1,...,N} \ F; // still unassigned nodes
6    while ( ∃j ∈ U : λ(j) ≠ 0 ) {
7      Compute λ(i) for all i ∈ U; // Connectivity measure [Lecture →
      Eq. (4.3.3.11)]
8      i ∈ argmaxj ∈ U{λ(j)}; // Most suitable C-node
9      C ← C ∪ {i}; // Grow set of C-nodes
10     F ← F ∪ S(i)*; // Eligible "receiver nodes" as F-nodes
11     U ← U \ ({i} ∪ S(i)*);
12   }
13   F ← F ∪ U; // Make all remaining nodes F-nodes
14   return { C, F };

```

The difference to [Lecture → Code 4.3.3.12] is that the connectivity measure is recomputed for all nodes in the inner loop. This is wasteful, because each execution of the loop body will only change the C/F-status of a few nodes. Consequently, only a few of the $\lambda(i)$ s will be affected.

(4-5.b) ☐ (15 min.) Code for this problem will sometimes involve loops over the non-zero entries in a row/column of a sparse matrix stored in an `Eigen::SparseMatrix<double>` object. Familiarize yourself with EIGEN's special iterators for traversing the non-zero entries in rows/columns of sparse matrices. To that end study [NumCSE course → Code 2.7.2.4] and the implementation of `buildAH()` in the file `cfsplit.cpp`. ▲

(4-5.c) ☐ (45 min.) In the file `cfsplit.cpp` complete the code of the member function

```

std::pair<
std::vector<std::vector<RugeStuebenAMG::nodeidx>>,
std::vector<std::vector<RugeStuebenAMG::nodeidx>>>
RugeStuebenAMG::getStrongConn (

```

```
const Eigen::SparseMatrix<double> &A,
double tauC) const;
```

- The parameter `A` passes the sparse symmetric system matrix $\mathbf{A} \in \mathbb{R}^{n,n}$.
- The argument `tauC` specifies the parameter τ_C in [Lecture → Def. 4.3.3.2] of strong (negative) coupling.
- The first returned object, call it `Si`, is an array of index vectors, for which `Si[i]` provides the index set $\mathcal{S}(i)$ for node i as defined in [Lecture → § 4.3.3.1] (C++ indexing).
- The second returned object corresponds to the index sets $\mathcal{S}(j)^*$ encoded in the same way as the $\mathcal{S}(i)$ s in `Si`.

SOLUTION for (4-5.c) → [4-5-3-0:cfs3.pdf](#) ▲

(4-5.d) 🕒 (30 min.) In the file `cfsplit.cpp` code the member function

```
void RugeStuebenAMG::setSmoothable(
const Eigen::SparseMatrix<double> &A,
double sigma,
std::vector<NodeFlag> &nodeflags) const;
```

This function is expected to set the status flags of smooth able nodes to `FINE`. A node i is regarded as smoothable, if

$$|(\mathbf{A})_{i,i}| \geq \sigma \sum_{k \neq i} |(\mathbf{A})_{i,k}|, \quad \sigma > 1, \text{ e.g. } \sigma = \frac{3}{2}, \quad i \in \{1, \dots, n\}. \quad [\text{Lecture} \rightarrow \text{Eq. (4.3.3.9)}]$$

(Here, linear algebra indexing is used.)

- The argument `A` passes the sparse symmetric matrix $\mathbf{A} \in \mathbb{R}^{n,n}$.
- The argument `delta` gives the parameter σ in [Lecture → Eq. (4.3.3.9)].
- The variable `nodeflags` returns the updated array of status flags for all nodes. We expect `nodeflags.size() == A.cols()`. Status flags of non-smoothable nodes should not be touched.

SOLUTION for (4-5.d) → [4-5-4-0:cfs4.pdf](#) ▲

(4-5.e) 🕒 (30 min.) [depends on Sub-problem (4-5.c)]

Supplement the missing code for the member function

```
std::vector<unsigned int>
RugeStuebenAMG::connectivityMeasure(
const std::vector<NodeFlag> &nodeflags,
const std::vector<std::vector<nodeidx>> &Sj_star) const;
```

This function is supposed to return the **connectivity measures** (linear algebra indexing here!)

$$\lambda(i) := \#(\mathcal{S}(i)^* \cap \mathcal{U}) + 2 \cdot \#(\mathcal{S}(i)^* \cap \mathcal{F}), \quad i \in \{1, \dots, N\}, \quad [\text{Lecture} \rightarrow \text{Eq. (4.3.3.11)}]$$

for all the nodes in the index set $\{1, \dots, \text{nodeflags.size}()\}$ (LA indexing). The argument `nodeflags` supplies the status flags for all nodes, and `Sj_star` provides the sets $\mathcal{S}(j)^*$ according to [Lecture → § 4.3.3.1].

SOLUTION for (4-5.e) → [4-5-5-0:cfs5.pdf](#) ▲

(4-5.f) 🕒 (45 min.) [depends on Sub-problem (4-5.c), Sub-problem (4-5.d), Sub-problem (4-5.e)]

Taking the cue from Code 4.5.3 implement the member function

```
std::vector<RugeStuebenAMG::NodeFlag>
RugeStuebenAMG::CFsplit (
  const Eigen::SparseMatrix<double> &A,
  const std::vector<std::vector<nodeidx>> &Sj_star);
```

It is supposed to compute the C/F-splitting of the degrees of freedom/nodes and encode it in a flag array that is returned.

- The parameter A provides the underlying sparse matrix A
- The argument Sj_star gives the sets $\mathcal{S}(j)^*$ in an array, as it is returned by the member function `getStrongConn()`.

Of course, also the other member functions implemented so far, should be used.

SOLUTION for (4-5.f) → [4-5-6-0:cfs6.pdf](#) ▲

(4-5.g) 🕒 (60 min.) In the file `cfsplit.cpp` complete the code of the member function

```
Eigen::SparseMatrix<double, Eigen::RowMajor>
RugeStuebenAMG::computeP (
  const Eigen::SparseMatrix<double> &A,
  const std::vector<std::vector<nodeidx>> &Si,
  const std::vector<NodeFlag> &nodeflags) const;
```

Its purpose is to compute the sparse prolongation matrix $P \in \mathbb{R}^{N_\ell, N_{\ell-1}}$ based on

- the sparse (symmetric) system matrix $A \in \mathbb{R}^{N_\ell, N_\ell}$ supplied through the parameter A ,
- sets $\mathcal{S}(i)$ of strongly connected nodes stored in Si for every degree of freedom $i \in \{0, \dots, N_\ell - 1\}$ (C++ indexing),
- and a given C/F-splitting of the degrees of freedom specified through the flag array `nodeflags` of length N_ℓ .

The formulas defining the entries of P are

$$(P)_{i,j} = \begin{cases} 1 & \text{, if } i \text{ is the } j\text{-th index in } \mathcal{C}, \\ 0 & \text{for all other indices } i \in \mathcal{C}, \end{cases} \quad [\text{Lecture} \rightarrow \text{Eq. (4.3.1.9)}]$$

for the self-coupling of C-nodes, and ($i \in \mathcal{F}$)

$$(P\vec{\eta})_i := -\frac{1}{(A)_{i,i}} \left(\alpha_i \left(\sum_{j \in \mathcal{S}(i) \cap \mathcal{C}} (A)_{i,j}^- (\vec{\eta})_j + \sum_{j \in \mathcal{S}(i) \cap \mathcal{F}} \frac{(A)_{i,j}^-}{(A)_{j,j}} \cdot \hat{\alpha}_j \cdot \sum_{k \in \mathcal{S}(j) \cap \mathcal{C}} (A)_{j,k} (\vec{\eta})_k \right) + \hat{\beta}_i \sum_{j \in \mathcal{C} \cap \mathcal{S}(i)} (A)_{i,j}^+ (\eta)_j \right), \quad [\text{Lecture} \rightarrow \text{Eq. (4.3.4.10)}]$$

$$\alpha_i := \frac{\sum_{k \neq i} (A)_{i,k}^-}{\sum_{k \in \mathcal{S}(i)} (A)_{i,k}^-}, \quad \hat{\alpha}_j := \frac{\sum_{k \neq j} (A)_{j,k}}{\sum_{k \in \mathcal{S}(j) \cap \mathcal{C}} (A)_{j,k}}, \quad \hat{\beta}_i := \frac{\sum_{k \neq i} (A)_{i,k}^+}{\sum_{j \in \mathcal{C} \cap \mathcal{S}(i)} (A)_{i,j}^+},$$

for the $C \rightarrow F$ prolongation weights:

$$\begin{aligned}
 (\mathbf{P})_{ij}^+ &= -\frac{\alpha_i(\mathbf{A})_{ij}^- + \hat{\beta}_i(\mathbf{A})_{ij}^+}{(\mathbf{A})_{ii}}, \text{ if } j \in \mathcal{S}(i) \cap \mathcal{C}, \\
 (\mathbf{P})_{ik}^+ &= -\frac{\alpha_i \hat{\alpha}_j(\mathbf{A})_{ij}^- (\mathbf{A})_{jk}}{(\mathbf{A})_{ii}(\mathbf{A})_{jj}}, \text{ if } k \in \mathcal{S}(j) \cap \mathcal{C} \text{ for some } j \in \mathcal{S}(i) \cap \mathcal{F}.
 \end{aligned}
 \tag{4.5.10}$$

Here, the indices j and k mean the index number of the corresponding C-node. Also note that these are update formulas for the entries of the prolongation matrix, which is to be initialized with zero.

SOLUTION for (4-5.g) \rightarrow 4-5-7-0:cfs6a.pdf ▲

(4-5.h) 📄 (60 min.) [depends on Sub-problem (4-5.c), Sub-problem (4-5.g), Sub-problem (4-5.c)]

In the file `cfsplit.h` write the code for the non-default constructor of **RugeStuebenAMG**:

```

template <typename RECORDER =
std::function<void (const Eigen::SparseMatrix<double> &,
const Eigen::SparseMatrix<double> &,
const std::vector<NodeFlag> &>>
explicit RugeStuebenAMG (
const Eigen::SparseMatrix<double> &A,
unsigned int min_mat_size = 10,
double tauC = 0.25, double sigma = 1.5,
RECORDER &&rec = [] (
const Eigen::SparseMatrix<double> & /*AH*/,
const Eigen::SparseMatrix<double> & /*P*/,
const std::vector<NodeFlag> & /*flags*/) -> void {});

```

The constructor has to initialize all data members of **RugeStuebenAMG** by

- starting from the matrix $\mathbf{A} = \mathbf{A}_L$ on the finest level and working its way up to the coarsest level $\ell = 0$,
- calling `RugeStuebenAMG::CFsplit()` and `RugeStuebenAMG::computeP()` on each level $\ell > 0$ to obtain the prolongation matrices $\mathbf{P}_{\ell-1,\ell}$,
- and then carrying out the Galerkin construction $\mathbf{A}_{\ell-1} = \mathbf{P}_{\ell-1,\ell}^T \mathbf{A}_\ell \mathbf{P}_{\ell-1,\ell}$.

Of course, the total number L of coarse levels is not known in advance. We stop advancing to even coarser levels, when the number of C-nodes on the current level drops below `min_mat_size`. Then we perform the sparse LU-decomposition of the obtained coarse-grid matrix.

The parameters `tauC` and `sigma` correspond to the arguments of `RugeStuebenAMG::getStrongConn()` and `RugeStuebenAMG::setSmoothable()`, respectively, with the same name.

In `rec` one can pass a functor object, which, on each level $\ell > 0$, is given the newly constructed CF-splitting (through a flag array), the prolongation matrix $\mathbf{P}_{\ell-1,\ell}$ and the coarse-grid matrix $\mathbf{A}_{\ell-1}$.

To facilitate the implementation, the Galerkin construction can rely on the function

```

Eigen::SparseMatrix<double> buildAH (
const Eigen::SparseMatrix<double> &A,
const Eigen::SparseMatrix<double, Eigen::RowMajor> &P);

```

which returns $\mathbf{P}^T \mathbf{A} \mathbf{P}$ as a sparse matrix. This function is borrowed from Problem 4-3.

SOLUTION for (4-5.h) → [4-5-8-0:cfs7.pdf](#) ▲

(4-5.i) 🕒 (30 min.) [depends on Sub-problem (4-5.h)]

In `cfsplit.cpp` supplement the missing lines of the member function

```
void RugeStuebenAMG::iterate (
  const Eigen::VectorXd &phi,
  Eigen::VectorXd &mu) const;
```

that carries out one step of the AMG V-cycle for the linear system of equations $A\vec{\mu} = \vec{\phi}$. The parameter `mu` passes the initial guess and returns the next iterate. A single Gauss-Seidel pre- and post-smoothing step should be used as in [Lecture → Code 4.3.1.4].

SOLUTION for (4-5.i) → [4-5-9-0:.pdf](#) ▲

(4-5.j) 🕒 (30 min.) [depends on Sub-problem (4-5.i)]

Taking the cue from [Lecture → Code 4.1.3.22] in `cfsplit.cpp` implement a C++ function

```
double cvgRateAMG(const Eigen::SparseMatrix<double> &A, double
  tol);
```

that determines the empiric rate of linear convergence for the Ruge-Stüben AMG method provided by the class **RugeStuebenAMG** for the system matrix passed in `A`. Use the default parameters for the constructor of **RugeStuebenAMG**.

SOLUTION for (4-5.j) → [4-5-10-0:.pdf](#) ▲

End Problem 4-5 , 360 min.

SOLUTION of (2-1.a):

We draw a vector $\in [0, 1]^2$ from a uniform distribution, which can be done using EIGEN's `Random()` initialization function. Then we check the minimal distance requirement and discard the random vector in case it is violated.

C++ code 2.1.3: Random initialization of star positions Get it on [GitLab \(gravitationalforces.cpp\)](#).

```
2  std::vector<Eigen::Vector2d> initStarPositions(unsigned int n, double mindist) {
3      assertm(mindist > 0, "Minimal distance must be positive");
4      const double max_md = 1.0 / (2.0 * std::sqrt(n)); // Maximal minimal distance
5      if (mindist > max_md) {
6          mindist = max_md;
7      }
8      // Vector for returning positions
9      std::vector<Eigen::Vector2d> pos;
10     // Fill position vector with random positions taking into account
11     // minimal
12     // distance requirement
13     do {
14         Eigen::Matrix<double, 2, Eigen::Dynamic> randpos =
15             0.5 * (Eigen::Matrix<double, 2, Eigen::Dynamic>::Random(2, n) +
16                 Eigen::Matrix<double, 2, Eigen::Dynamic>::Constant(2, n, 1.0));
17         // Add columns of random matrix one by one checking minimal distance
18         for (int i = 0; (i < n) && (pos.size() < n); ++i) {
19             int m = 0;
20             bool good = true;
21             while ((m < pos.size()) and
22                 (good = (pos[m] - randpos.col(i)).norm() > mindist)) {
23                 ++m;
24             }
25             if (good) {
26                 // Accept next position vector
27                 pos.emplace_back(randpos.col(i));
28             }
29             // It can happen that positions are rejected. In this case we have
30             // to try
31             // more.
32         }
33     } while (pos.size() < n);
34     return pos;
35 }
```

SOLUTION of (2-1.b):

A double-loop implementation of (2.1.2) is straightforward.

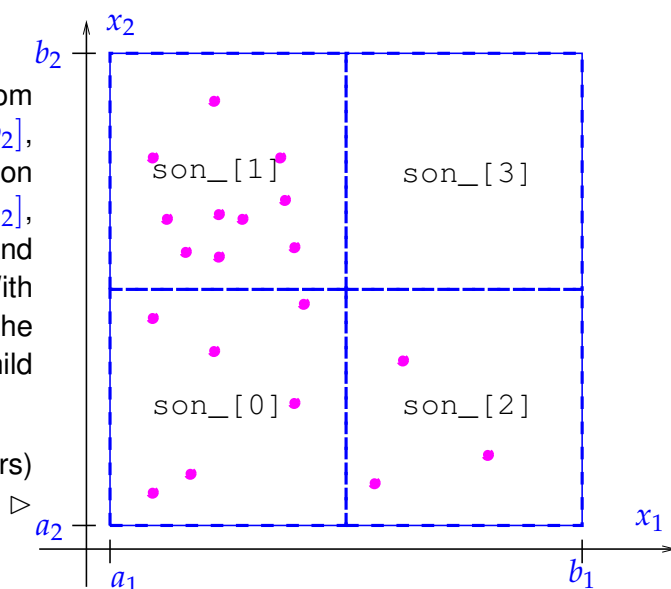
C++ code 2.1.4: Exact direct computation of force on every star
Get it on [GitLab \(gravitationalforces.cpp\)](https://gitlab.com/gravitationalforces).

```
2  std::vector<Eigen::Vector2d> computeForces_direct(  
3      const std::vector<Eigen::Vector2d> &masspositions,  
4      const std::vector<double> &masses) {  
5      const unsigned int n = masspositions.size();  
6      assertm(n == masses.size(),  
7          "Mismatch of sizes of masspositions and masses vectors");  
8      // Forces will be stored in this array  
9      std::vector<Eigen::Vector2d> forces(n);  
10     Eigen::Vector2d acc;  
11     for (unsigned int j = 0; j < n; j++) {  
12         // Compute force on j-th star  
13         acc.setZero();  
14         for (unsigned int i = 0; i < n; i++) {  
15             if (i != j) {  
16                 const Eigen::Vector2d diff_masspositions =  
17                     masspositions[i] - masspositions[j];  
18                 acc += diff_masspositions * masses[i] /  
19                     std::pow(diff_masspositions.norm(), 3);  
20             }  
21         }  
22         forces[j] = acc * masses[j] / (4 * M_PI);  
23     }  
24     return forces;  
25 }
```

SOLUTION of (2-1.c):

The construction is done *recursively*. Starting from a sub-cluster with bounding box $[a_1, b_1] \times [a_2, b_2]$, we sort its stars into four subsets depending on whether they lie in the boxes $[a_1, m_1] \times [a_2, m_2]$, $[a_1, m_1] \times [m_2, b_2]$, $[m_1, b_1] \times [a_2, m_2]$, and $[m_1, b_1] \times [m_2, b_2]$, where $m_i := \frac{1}{2}(a_i + b_i)$. With these subsets, if they are not empty, we call the constructor of **StarQuadTreeNode** to create child nodes.

In the situation sketched beside (• marks stars)
son_[3] == **nullptr**.



Throughout we may set the bounding boxes of sub-clusters to squares with side lengths 2^{-L} for a quadtree node on level $L \in \mathbb{N}_0$. The bounding boxes of the child nodes can simply be chosen as squares of half the size as indicated in Fig. 1.

C++ code 2.1.6: Constructor of **StarQuadTree::StarQuadTreeNode** Get it on [GitLab \(gravitationalforces.cpp\)](https://gitlab.com/gravitationalforces/cpp).

```
2 StarQuadTree::StarQuadTreeNode::StarQuadTreeNode(
3     std::vector<unsigned int> star_idx, Eigen::Matrix2d bbox,
4     StarQuadTree &tree)
5     : star_idx_(star_idx), bbox_(bbox), mass(0.0), center({0, 0}) {
6     assertm(star_idx_.size() > 0, "Can't create a node without stars...");
7     // This is a sub-optimal implementation!
8     // A better algorithm initializes the total masses
9     // and center of gravity of son cluster first and then computes those
10    // quantities for the parent cluster from this information. This
11    // saves
12    // looping through the stars of a cluster and summing up their masses
13    // and
14    // (weighted) position vectors.
15
16    // Total mass by summation
17    for (unsigned int i = 0; i < star_idx_.size(); i++) {
18        const Eigen::Vector2d starpos(tree.starpos_[star_idx_[i]]);
19        mass += tree.starmasses_[star_idx_[i]];
20        center += tree.starmasses_[star_idx_[i]] * starpos;
21        assertm(((starpos[0] >= bbox(0, 0)) and (starpos[0] <= bbox(0, 1)) and
22                (starpos[1] >= bbox(1, 0)) and (starpos[1] <= bbox(1, 1))),
23                "Star out of bounding box!");
24    }
25
26    // Center of gravity for cluster of star associated with current node
27    center = center / mass;
28
29    // In this implementation leaf nodes contain a single star, whose
30    // position
31    // also agrees with the center of the node cluster.
```

```

28 if (star_idx_.size() == 1) {
29     tree.no_leaves_++;
30     return;
31 }
32 tree.no_clusters_++;
33 // Sub-boxes obtained by halving bounding box in each direction
34 const double m1 = 0.5 * (bbox_(0, 0) + bbox_(0, 1));
35 const double m2 = 0.5 * (bbox_(1, 0) + bbox_(1, 1));
36 std::array<Eigen::Matrix2d, 4> sub_boxes;
37 sub_boxes[0] << bbox_(0, 0), m1, bbox_(1, 0), m2;
38 sub_boxes[1] << bbox_(0, 0), m1, m2, bbox_(1, 1);
39 sub_boxes[2] << m1, bbox_(0, 1), bbox_(1, 0), m2;
40 sub_boxes[3] << m1, bbox_(0, 1), m2, bbox_(1, 1);
41 // Index sets for son clusters
42 std::array<std::vector<unsigned int>, 4> sub_indices;
43
44 // Check, which son box the stars lie in, cf. [Lecture → ??]
45 for (unsigned int i = 0; i < star_idx_.size(); i++) {
46     if (tree.starpos_[star_idx_[i]][0] < m1) { // left part
47         if (tree.starpos_[star_idx_[i]][1] < m2) { // left bottom
48             sub_indices[0].push_back(star_idx_[i]);
49         } else { // left top
50             sub_indices[1].push_back(star_idx_[i]);
51         }
52     } else { // right part
53         if (tree.starpos_[star_idx_[i]][1] < m2) { // right bottom
54             sub_indices[2].push_back(star_idx_[i]);
55         } else { // right top
56             sub_indices[3].push_back(star_idx_[i]);
57         }
58     }
59 }
60 // Recursion: son nodes are created, if non-empty
61 for (int i : {0, 1, 2, 3}) {
62     if (!sub_indices[i].empty())
63         sons_[i] = std::move(std::make_unique<StarQuadTree::StarQuadTreeNode>(
64             sub_indices[i], sub_boxes[i], tree));
65 }
66 }

```

SOLUTION of (2-1.d):

C++ code 2.1.9: Implementation of member function StarQuadTreeClustering::isAdmissible(). [Get it on GitLab \(gravitationalforces.cpp\)](#)

```
2 bool StarQuadTreeClustering::isAdmissible(const StarQuadTreeNode &node,
3                                           Eigen::Vector2d p, double eta) const {
4     // Implements admissibility condition \ref{eq:admstar}
5     bool admissible;
6     // Diameter of bounding box is the distance of its opposite corners
7     const double diam = (node.bbox_.col(0) - node.bbox_.col(1)).norm();
8     // To compute the distance of a point from an axis-aligned box we have to
9     // distinguish nine different cases, which can conveniently be done by first
10    // introducing a function computing distances in 1D.
11    auto intvdist = [](double a, double b, double x) -> double {
12        if (b < a) std::swap(a, b);
13        if (x < a) return (a - x);
14        if (x > b) return (x - b);
15        return 0.0;
16    };
17    const double dx = intvdist(node.bbox_(0, 0), node.bbox_(0, 1), p[0]);
18    const double dy = intvdist(node.bbox_(1, 0), node.bbox_(1, 1), p[1]);
19    const double dist = Eigen::Vector2d(dx, dy).norm();
20    admissible =
21        (dist > eta * diam); // Admissibility condition \ref{eq:admstar}
22    return admissible;
23 }
```


SOLUTION of (2-1.e):

To evaluate (??) traverse the quadtree from the root down to the leaves. Whenever a node is admissible with respect to x^j add the force due to its equivalent star. If a node is not admissible, continue with its children.

C++ code 2.1.11: Implementation of member function StarQuadTreeClustering::forceOnStar(). [Get it on GitLab \(gravitationalforces.cpp\)](#).

```
2 Eigen::Vector2d StarQuadTreeClustering::forceOnStar(unsigned int j,
3                                     double eta) const {
4     Eigen::Vector2d acc; // For summation of force
5     acc.setZero();
6     // Trick: Recursive lambda function capturing the whole object
7     std::function<void(const StarQuadTreeNode *)> traverse =
8     [&](const StarQuadTreeNode *node) {
9         if (node == nullptr)
10            return; // In case if the cluster has no stars in it
11            // Since leaf clusters contain only a single star we can treat
12            // them in the same way as admissible clusters.
13            if (isAdmissible(*node, starpos_[j], eta) or (node->isLeaf())) {
14                const Eigen::Vector2d diff_masspositions = node->center - starpos_[j];
15                const double dist = diff_masspositions.norm();
16                if (dist > 1.0E-10) {
17                    acc += diff_masspositions * node->mass / pow(dist, 3);
18                }
19            } else { // traverse further, if current sub-cluster is not
20                // admissible
21                traverse(node->sons_[0].get());
22                traverse(node->sons_[1].get());
23                traverse(node->sons_[2].get());
24                traverse(node->sons_[3].get());
25            }
26        };
27        // Start of recursion
28        traverse(this->root_.get());
29        // Multiply with forefactor in (??)
30        acc = (acc * starmasses_[j] / (4 * M_PI));
31        return acc;
32    }
```

Note that a lambda function within a member function of a class has access to all class variables and member functions, once **this** is captured. This can be done explicitly (`[this]`) or through the **implicit capture** `[&]`.

SOLUTION of (2-1.f):

The implementation is straightforward using `StarQuadTreeClustering::forceOnStar()`.

C++ code 2.1.12: Implementation of `forceError()`
Get it on [GitLab](#) ([gravitationalforces.cpp](#)).

```
2  std::vector<double> forceError(const StarQuadTreeClustering &qt,
3                               const std::vector<double> &etas) {
4      std::vector<double> error(etas.size()); // For returning errors
5      std::vector<Eigen::Vector2d> exact_forces{
6          computeForces_direct(qt.starpos_, qt.starmasses_)};
7
8      std::vector<double> normed_errors(qt.n);
9      // Compute errors for different values of the admissibility parameter
10     for (int eta_i = 0; eta_i < etas.size(); eta_i++) {
11         for (unsigned int j = 0; j < qt.n; j++) {
12             const Eigen::Vector2d force_j = qt.forceOnStar(j, etas[eta_i]);
13             normed_errors[j] = (exact_forces[j] - force_j).norm();
14         }
15         error[eta_i] =
16             *std::max_element(normed_errors.begin(), normed_errors.end());
17     }
18     return error;
19 }
```

The larger η the smaller the clustering error.

This is not surprising, because the larger η the more stringent the admissibility condition for clusters, the fewer “equivalent stars” are used and the more pairwise interactions are taken into account.

For very large η , eventually, no cluster would be admissible and the algorithm would just compute the pairwise interactions.

η	error
1	16.4
1.25	9.89
1.5	7.00
1.75	5.06
2	2.94
2.25	2.77
2.5	2.14
2.75	1.52
3	1.07

HINT 1 for (2-1.g): This code snippet shows the use of the C++ chrono library for measuring runtimes:

C++ code 2.1.13: Measuring runtime of a part of a C++ code.
Get it on [GitLab \(gravitationalforces.cpp\)](#).

```
2 std::pair<double, double> measureRuntimes(unsigned int n, unsigned int n_runs) {
3     assertm((n > 1), "At least two stars required!");
4     // Initialize star positions
5     std::vector<Eigen::Vector2d> pos = GravitationalForces::initStarPositions(n);
6     // All stars have equal (unit) mass
7     std::vector<double> mass(n, 1.0);
8     double ms_exact =
9         std::numeric_limits<double>::max(); // Time measured for exact
10        evaluation
11    double ms_cluster = std::numeric_limits<
12        double>::max(); // Time taken for clustering-based evaluation
13    // Build quad tree of stars
14    StarQuadTreeClustering qt(pos, mass);
15    // Admissibility parameter
16    const double eta = 1.5;
17
18    // Runtime for exact computation of forces with effort  $O(n^2)$ 
19    std::vector<Eigen::Vector2d> forces(n);
20    for (int r = 0; r < n_runs; ++r) {
21        auto t1_exact = std::chrono::high_resolution_clock::now();
22        forces = computeForces_direct(qt.starpos_, qt.starmasses_);
23        auto t2_exact = std::chrono::high_resolution_clock::now();
24        /* Getting number of milliseconds as a double. */
25        std::chrono::duration<double, std::milli> ms_double = (t2_exact - t1_exact);
26        ms_exact = std::min(ms_exact, ms_double.count());
27    }
28    std::cout << "n = " << n << " : runtime computeForces_direct= " << ms_exact
29        << "ms\n";
30
31    // Runtime for cluster-based approximate evaluation, cost  $O(n \log n)$ 
32    for (int r = 0; r < n_runs; ++r) {
33        auto t1_cluster = std::chrono::high_resolution_clock::now();
34        for (unsigned int j = 0; j < qt.n; j++) {
35            forces[j] = qt.forceOnStar(j, eta);
36        }
37        auto t2_cluster = std::chrono::high_resolution_clock::now();
38        /* Getting number of milliseconds as a double. */
39        std::chrono::duration<double, std::milli> ms_double =
40            (t2_cluster - t1_cluster);
41        ms_cluster = std::max(ms_cluster, ms_double.count());
42    }
43    std::cout << "n = " << n << " : runtime forceOnStar[eta=" << eta
44        << "] = " << ms_cluster << "ms\n";
45    return {ms_exact, ms_cluster};
46 }
```



For runtime measurements it is essential to compile your code in *Release Mode*: configure your CMake build system accordingly!

SOLUTION of (2-1.g):

C++ code 2.1.14: Function `measureRuntimes()` Get it on  [GitLab \(gravitationalforces.cpp\)](#).

```

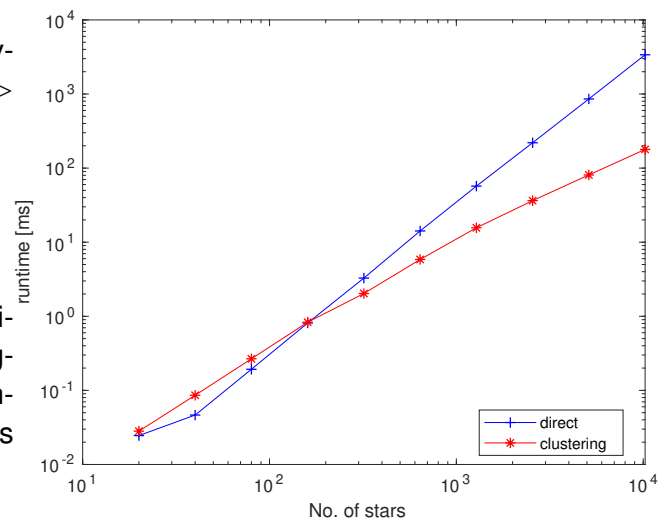
2  std::pair<double, double> measureRuntimes(unsigned int n, unsigned int n_runs) {
3      assertm((n > 1), "At least two stars required!");
4      // Initialize star positions
5      std::vector<Eigen::Vector2d> pos = GravitationalForces::initStarPositions(n);
6      // All stars have equal (unit) mass
7      std::vector<double> mass(n, 1.0);
8      double ms_exact =
9          std::numeric_limits<double>::max(); // Time measured for exact
          evaluation
10     double ms_cluster = std::numeric_limits<
11         double>::max(); // Time taken for clustering-based evaluation
12     // Build quad tree of stars
13     StarQuadTreeClustering qt(pos, mass);
14     // Admissibility parameter
15     const double eta = 1.5;
16
17     // Runtime for exact computation of forces with effort  $O(n^2)$ 
18     std::vector<Eigen::Vector2d> forces(n);
19     for (int r = 0; r < n_runs; ++r) {
20         auto t1_exact = std::chrono::high_resolution_clock::now();
21         forces = computeForces_direct(qt.starpos_, qt.starmasses_);
22         auto t2_exact = std::chrono::high_resolution_clock::now();
23         /* Getting number of milliseconds as a double. */
24         std::chrono::duration<double, std::milli> ms_double = (t2_exact - t1_exact);
25         ms_exact = std::min(ms_exact, ms_double.count());
26     }
27     std::cout << "n = " << n << " : runtime computeForces_direct= " << ms_exact
28         << "ms\n";
29
30     // Runtime for cluster-based approximate evaluation, cost  $O(n \log n)$ 
31     for (int r = 0; r < n_runs; ++r) {
32         auto t1_cluster = std::chrono::high_resolution_clock::now();
33         for (unsigned int j = 0; j < qt.n; j++) {
34             forces[j] = qt.forceOnStar(j, eta);
35         }
36         auto t2_cluster = std::chrono::high_resolution_clock::now();
37         /* Getting number of milliseconds as a double. */
38         std::chrono::duration<double, std::milli> ms_double =
39             (t2_cluster - t1_cluster);
40         ms_cluster = std::max(ms_cluster, ms_double.count());
41     }
42     std::cout << "n = " << n << " : runtime forceOnStar[eta=" << eta
43         << "] = " << ms_cluster << "ms\n";
44     return {ms_exact, ms_cluster};
45 }

```

Plot of runtimes of the two ways to compute all gravitational forces on the stars. ▽

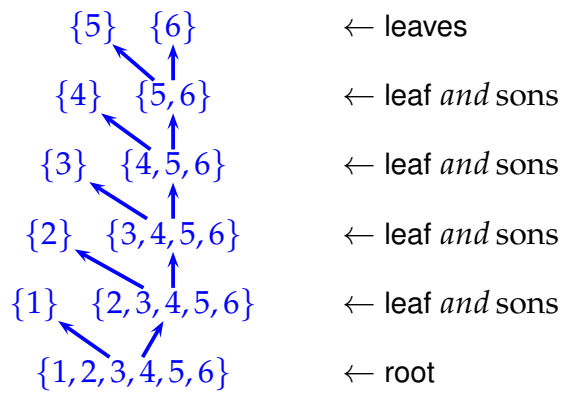
CPU : Intel(R) Core(TM) i7-8550U @ 1.8GHz
OS : Linux (kernel 6.0.11)
Compiler: GCC 12.2.1
Options : -O3 -DNDEBUG

We see that for large number of stars the approximate force evaluation by clustering techniques is significantly faster than the direct pairwise force computation. The runtimes reflect the $O(n \log n)$ versus $O(n^2)$ advantage.



SOLUTION of (2-2.a):

Do the geometric splitting according to (2.2.1) by “paper and pencil” and observe that one son will always be assigned only one point.



SOLUTION of (2-2.b):

We end up with an *empty far-field set* \mathbb{F}_{far} of cluster pairs!

You have to consider all possible pairs of nodes appearing in each level of the cluster tree of e.g. Sub-problem (2-2.a), given $n = 6$. The admissibility condition excludes leaves, so the only possible pair in level i (ordering the levels from bottom to top, i.e. from the root to the leaves) is $\{i, \dots, n\}$ with itself. The distance between the bounding boxes of this pair is therefore 0, meaning that the pair is not admissible.

Hence, there are no far-field cluster pairs.

SOLUTION of (2-3.a):

By the iterated product rule

$$N'_q(x) = \sum_{k=1}^q \prod_{\substack{j=1 \\ j \neq k}}^q (x - t_j), \quad x \in \mathbb{R}. \quad (2.3.5)$$

When $x = t_i$, $i = 1, \dots, q$, all summands except that for $k = i$ vanish and we get

$$N'_q(t_i) = \prod_{\substack{j=1 \\ j \neq i}}^q (t_i - t_j) = \lambda_i^{-1}. \quad (2.3.6)$$

Note that $N'_q(t_i) \neq 0$, because t_i is a simple zero of N_q .

HINT 1 for (2-3.b): The nodal polynomial for the set $\{t_j\}_{j=1}^q$ of Chebychev nodes from (2.3.1) is just the (scaled) Chebychev polynomial of degree q :

$$N_q = 2^{1-q} T_q \quad \forall q \in \mathbb{N} ! \quad (2.3.8)$$

┘

SOLUTION of (2-3.b):

Straightforward computation verifies $T_q(t_j) = 0$, which means $N_q = \xi T_q$ for some $\xi \in \mathbb{R}$. We recall the 3-term recurrence for Chebycehv polynomials:

Theorem [NumCSE course \rightarrow ??]. **3-term recursion for Chebychev polynomials**

The function T_n defined in (??) satisfy the **3-term recursion**

$$T_{n+1}(t) = 2t T_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1 \quad , \quad T_1(t) = t \quad , \quad n \in \mathbb{N} \quad . \quad [\text{NumCSE course} \rightarrow \text{??}]$$

This implies that T_q has leading coefficient 2^{q-1} , whereas that for N_q is equal to 1. This fixes the scaling factor $\tilde{\xi} = 2^{1-q}$ and proves the hint.

Next, using

$$\arccos'(x) = -\frac{1}{\sqrt{1-x^2}} \quad , \quad -1 < x < 1 \quad ,$$

and then the chain rule we get

$$T_q'(x) = -q \sin(q \arccos(x)) \frac{-1}{\sqrt{1-x^2}} \quad , \quad -1 < x \leq 1 \quad , \quad (2.3.9)$$

$$\blacktriangleright \quad T_q'(t_i) = q \underbrace{\sin\left(\frac{(2i-1)\pi}{2}\right)}_{(-1)^{i-1}} \frac{1}{\sin\left(\frac{2i-1}{2q} \pi\right)} \quad .$$

Combined with (2.3.4) this yields the assertion.

SOLUTION of (2-3.c):

We take the cue from [[NumCSE course](#) → ??].

C++ code 2.3.10: Implementation of `chebInterpEval1D()`

```
2  template <typename FUNCTOR>
3  std::vector<double> chebInterpEval1D(unsigned int q, FUNCTOR f,
4                                     std::vector<double> &x) {
5      // Initialize Chebychev nodes, compute barycentric weights
6      //  $\lambda_i$  (up to  $q$ -dependent scaling) and sample the function
7      std::vector<double> lambda(q); // barycentric weights
8      std::vector<double> t(q);      // Chebychev nodes
9      std::vector<double> y(q);      // Sampled function values
10     int sgn = 1;
11     for (int k = 0; k < q; ++k, sgn *= -1) {
12         t[k] =
13             std::cos((2.0 * (k + 1) - 1.0) / (2 * q) * M_PI); // (2.3.1)
14         lambda[k] =
15             std::pow(2, q - 1) / q * sgn *
16             std::sin((2.0 * (k + 1) - 1.0) / (2 * q) * M_PI); // (2.3.7)
17         y[k] = f(t[k]); //  $y_k$ 
18     }
19     // Loop over all evaluation points  $x_i$ 
20     const std::vector<double>::size_type N = x.size();
21     std::vector<double> res(N, 0.0); // Result vector
22     for (int k = 0; k < N; ++k) {
23         // Denominator in the barycentric interpolation formula
24         double den = 0.0;
25         bool nonode = true;
26         for (int j = 0; j < q; ++j) {
27             if (x[k] == t[j]) {
28                 // Avoid division by zero. In this case testing exact equality
29                 // of floating point numbers is safe, because the point of this test
30                 // is not to avoid amplification of roundoff errors, but really
31                 // preventing a division by zero exception.
32                 res[k] = y[j];
33                 nonode = false;
34                 break;
35             }
36             const double tmp = lambda[j] / (x[k] - t[j]);
37             res[k] += y[j] * tmp;
38             den += tmp;
39         }
40         if (nonode) {
41             res[k] /= den;
42         }
43     }
44     return res;
45 }
```

SOLUTION of (2-3.d):

We recall from [Lecture → ??] that we have to compute

$$(I \otimes I)(\left\{ \begin{bmatrix} \xi \\ \eta \end{bmatrix} \mapsto \mathbf{f}(\xi, \eta) \right\})(x_1^i, x_2^i), \quad \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix} = \mathbf{x}^i,$$

where I is the 1D Chebychev interpolation operator on $[-1, 1]$, which takes a function $[-1, 1] \rightarrow \mathbb{R}$ to an uni-variate polynomial $\mathbb{R} \rightarrow \mathbb{R}$. The interpretation of this formula is

$$(I \otimes I)(\left\{ \begin{bmatrix} \xi \\ \eta \end{bmatrix} \mapsto \mathbf{f}(\xi, \eta) \right\})(x, y) = I(\left\{ \xi \mapsto I(\left\{ \eta \mapsto f(\xi, \eta) \right\})(y) \right\})(x), \quad x, y \in \mathbb{R}, \quad (2.3.11)$$

Next, recall the barycentric representation [Lecture → ??] of the Chebychev interpolation operator, t_i the Chebychev nodes from (2.3.1),

$$I(\left\{ \xi \mapsto f(\xi) \right\})(x) = \sum_{i=1}^q \frac{\lambda_i}{x - t_i} f(t_i) \cdot \left(\sum_{i=1}^q \frac{\lambda_i}{x - t_i} \right)^{-1}, \quad x \neq t_\ell, \quad (2.3.12)$$

where the barycentric weights are given by (2.3.7). Combining (2.3.11) and (2.3.12) yields for fixed $x, y \in \mathbb{R}$

$$\begin{aligned} (I \otimes I)(\left\{ \begin{bmatrix} \xi \\ \eta \end{bmatrix} \mapsto \mathbf{f}(\xi, \eta) \right\})(x, y) \\ &= \sum_{j=1}^q \frac{\lambda_j}{x - t_j} \sum_{i=1}^q \frac{\lambda_i}{y - t_i} f(t_j, t_i) \cdot \left(\sum_{i=1}^q \frac{\lambda_i}{y - t_i} \right)^{-1} \cdot \left(\sum_{j=1}^q \frac{\lambda_j}{x - t_j} \right)^{-1} \\ &= \left(\sum_{i=1}^q w_y^i \right)^{-1} \left(\sum_{j=1}^q w_x^j \right)^{-1} \sum_{j=1}^q \sum_{i=1}^q w_x^j w_y^i f(t_j, t_i), \end{aligned} \quad (2.3.13)$$

with weights

$$w_x^j := \frac{\lambda_j}{x - t_j}, \quad w_y^i := \frac{\lambda_i}{y - t_i}.$$

This suggests the following algorithm:

1. Precompute the barycentric weights $\lambda_i, i = 1, \dots, q$.
2. Run through the sequence of evaluation points $(\mathbf{x}^\ell)_\ell$ and inside this loop
3. compute the weights w_x^i and $w_y^i, i = 1, \dots, q$, which depend on \mathbf{x}^ℓ and
4. evaluate the sums $s_x := \sum_{j=1}^q w_x^j$ and $s_y := \sum_{i=1}^q w_y^i$,
5. and finally evaluate $p(\mathbf{x}^\ell) := s_x^{-1} s_y^{-1} \sum_j \sum_i w_x^j w_y^i f(t_j, t_i)$.

Remark. A more efficient implementation is possible, if the evaluation points are located on a tensor-product grid.

Note that as in Code 2.3.14 the situation that a component of \mathbf{x}^i coincides with a Chebychev node has to be handled in order to avoid division by zero.

C++ code 2.3.14: Implementation of `chebInterpEval2D()`

```
2  template <typename FUNCTOR>
3  std::vector<double> chebInterpEval2D(unsigned int q, FUNCTOR f,
4                                     const std::vector<Eigen::Vector2d> &x) {
5      // Initialize Chebychev nodes, compute barycentric weights
6      //  $\lambda_i$  (up to  $q$ -dependent scaling) and sample the function
7      std::vector<double> lambda(q); // barycentric weights
8      std::vector<double> t(q);     // Chebychev nodes
9      int sgn = 1;
10     for (int k = 0; k < q; ++k, sgn *= -1) {
11         t[k] = std::cos((2.0 * (k + 1) - 1.0) / (2 * q) * M_PI);
12         lambda[k] = std::pow(2, q - 1) / q * sgn *
13                 std::sin((2.0 * (k + 1) - 1.0) / (2 * q) * M_PI);
14     }
15     Eigen::MatrixXd y(q, q); // Sampled function values
16     for (int k = 0; k < q; ++k) {
17         for (int l = 0; l <= k; ++l) {
18             y(k, l) = f(t[k], t[l]);
19             y(l, k) = f(t[l], t[k]);
20         }
21     }
22     // Loop over all evaluation points
23     const std::vector<double>::size_type N = x.size();
24     std::vector<double> res(N, 0.0); // Result vector
25     std::vector<double> wx(q);     // Weights  $w_x^i$ 
26     std::vector<double> wy(q);     // Weights  $w_y^i$ 
27     int nodex; // Store index of  $t_x$  in case of division by zero
28     int nodey; // Store index of  $t_y$  in case of division by zero
29     for (int k = 0; k < N; ++k) {
30         double sx = 0;
31         double sy = 0;
32         bool nonodex = true;
33         bool nonodey = true;
34
35         for (int j = 0; j < q; ++j) {
36             if (nonodex) {
37                 // In the case of division by zero
38                 if (std::abs(x[k][0] - t[j]) < 1e-9) {
39                     nonodex = false;
40                     nodex = j; // Keep track of the index
41                     if (!nonodey) { // Early termination in the case of second
42                                     // division
43                                     // by zero
44                                     break;
45                                 }
46                     wx[j] = lambda[j] / (x[k][0] - t[j]);
47                     sx += wx[j];
48                 }
49             }
50             // Only enter if no division by zero occurred
51             if (nonodey) {
52                 // In the case of division by zero
53                 if (std::abs(x[k][1] - t[j]) < 1e-9) {
54                     nonodey = false;
55                     nodey = j; // Keep track of the index
56                     if (!nonodex) { // Early termination in the case of second
```

```

57         division // by zero
58         break;
59     }
60     } else {
61         wy[j] = lambda[j] / (x[k][1] - t[j]);
62         sy += wy[j];
63     }
64 }
65 }
66 // The case of 2 division by zero
67 if (!nonodex && !nonodey) {
68     res[k] = y(nodex, nodey);
69 }
70 // The case of a division by zero along the x Axis
71 else if (!nonodex) {
72     for (int j = 0; j < q; ++j) {
73         res[k] += y(nodex, j) * wy[j];
74     }
75     res[k] /= sy;
76 }
77 // The case of a division by zero along the y Axis
78 else if (!nonodey) {
79     for (int j = 0; j < q; ++j) {
80         res[k] += y(j, nodey) * wx[j];
81     }
82     res[k] /= sx;
83 }
84 // No division by zero occurred
85 else {
86     for (int j = 0; j < q; ++j) {
87         for (int i = 0; i < q; ++i) {
88             res[k] += y(j, i) * wx[j] * wy[i];
89         }
90     }
91     res[k] /= sy * sx;
92 }
93 }
94 return res;
95 }

```

SOLUTION of (2-3.e):

From [NumCSE course → ??] we recall the **affine transformation** from the reference interval $[-1, 1]$ to a general interval $[a, b]$:

$$\Phi : [-1, 1] \rightarrow [a, b] \quad , \quad \Phi(\hat{x}) := \frac{1}{2}(b-a)\hat{x} + \frac{1}{2}(a+b) . \quad (2.3.15)$$

This can be generalized to two dimensions:

$$\Phi : \begin{cases} [-1, 1]^2 & \rightarrow [a_1, b_1] \times [a_2, b_2] \\ \hat{x} & \mapsto \frac{1}{2} \begin{bmatrix} b_1 - a_1 & 0 \\ 0 & b_2 - a_2 \end{bmatrix} \hat{x} + \frac{1}{2} \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix} . \end{cases}$$

This mapping induces a corresponding affine pullback of functions $f : [a_1, b_1] \times [a_2, b_2] \rightarrow \mathbb{R}$:

$$(\Phi^* f)(\hat{x}) := f(\Phi(\hat{x})) , \quad \hat{x} \in [-1, 1]^2 . \quad (2.3.16)$$

Given an interpolation operator on the reference square,

$$\hat{\Gamma} : C^0([-1, 1]^2) \rightarrow \mathcal{P}_{q-1}(\mathbb{R}^2) , \quad (2.3.17)$$

we can then define a corresponding interpolation operator on $[a_1, b_1] \times [a_2, b_2]$:

$$(If)(x) := \hat{\Gamma}(\Phi^* f)(\Phi^{-1}(x)) , \quad x \in \mathbb{R}^2 . \quad (2.3.18)$$

This can be realized as follows:

- Wrap f into a lambda function, which represents $\Phi^* f$.
- Map the evaluation points x^i under Φ^{-1} :

$$\Phi^{-1}(x) = \begin{bmatrix} \frac{2}{b_1 - a_1} & 0 \\ 0 & \frac{2}{b_2 - a_2} \end{bmatrix} \left(x - \frac{1}{2} \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \end{bmatrix} \right) . \quad (2.3.19)$$

- Call `ChebInterpEval2D()` with the wrapped function and the mapped evaluation points. It will return the desired result.

C++ code 2.3.20: Implementation of `genChebInterpEval2D()`

```
2 template <typename FUNCTOR>
3 std::vector<double> genChebInterpEval2D(unsigned int q, FUNCTOR f,
4 Eigen::Vector2d a, Eigen::Vector2d b,
5 const std::vector<Eigen::Vector2d> &x) {
6 const std::vector<double>::size_type N = x.size();
7 std::vector<double> res = std::vector<double>(N, 0.0); // Result vector
8
9 // Transformation from [-1,1]^2 to
10 // [a1,b1] x [a2,b2]
11 auto phif = [f, a, b](double x1, double x2) {
12     const double tmp1 = 0.5 * ((b[0] - a[0]) * x1 + a[0] + b[0]);
13     const double tmp2 = 0.5 * ((b[1] - a[1]) * x2 + a[1] + b[1]);
14     return f(tmp1, tmp2);
15 };
```

```
16 // Inverse transformation  $\Phi^{-1}$ 
17 auto phiinv = [a, b](const Eigen::Vector2d &x) {
18     const double x1 = 2. / (b[0] - a[0]) * (x[0] - 0.5 * (a[0] + b[0]));
19     const double x2 = 2. / (b[1] - a[1]) * (x[1] - 0.5 * (a[1] + b[1]));
20     return (Eigen::Vector2d() << x1, x2).finished();
21 };
22 // Vector of transformed evaluation points
23 std::vector<Eigen::Vector2d> phiinvx(N);
24 for (unsigned int k = 0; k < N; ++k) {
25     phiinvx[k] = phiinv(x[k]); // Affine transformation
26 }
27 // Interpolation on  $[-1,1]^2$ 
28 res = chebInterpEval2D(q, phif, phiinvx);
29 return res;
30 }
```

SOLUTION of (2-3.f):

Solving this problem is left to the brightest students in the course.

HINT 1 for (2-4.j): A simple implementation of `approxErrorLLR()` can rely on `mvLLRPartMat()` from Sub-problem (2-4.i).

SOLUTION of (2-4.j):

Write n for the dimension of the square matrices \mathbf{M} and $\widetilde{\mathbf{M}}$ and $\vec{\epsilon}_j$ for the j -th Cartesian coordinate vector in \mathbb{R}^n . Then

$$\text{err}^2 = \frac{1}{n^2} \sum_{j=1}^n \left\| \mathbf{M} \vec{\epsilon}_j - \widetilde{\mathbf{M}} \vec{\epsilon}_j \right\|_2^2,$$

and the matrix×vector multiplication $\widetilde{\mathbf{M}} \vec{\epsilon}_j$ can be realized by means of `mvLLRPartMat()`.

C++ code 2.4.22: Computation of approximation error →GITLAB

```
2 template <typename KERNEL>
3 std::pair<double, double> approxErrorLLR(BiDirChebPartMat1D<KERNEL> &llrcM) {
4     const size_t n = llrcM.rows();
5     const size_t m = llrcM.cols();
6     Eigen::MatrixXd M(n, m); // Exact kernel collocation matrix
7     Eigen::MatrixXd Mt(n, m); // Compressed matrix as dense matrix
8     // Not a particularly memory-efficient implementation!
9     const std::vector<HMAT::Point<1>> &row_pts{(llrcM.rowT->root)->pts};
10    const std::vector<HMAT::Point<1>> &col_pts{(llrcM.colT->root)->pts};
11    for (int i = 0; i < n; ++i) {
12        for (int j = 0; j < m; ++j) {
13            M(i, j) = llrcM.G(row_pts[i].x[0], col_pts[j].x[0]);
14        }
15    }
16    // Compute compressed matrix as dense matrix
17    Eigen::VectorXd x(m);
18    x.setZero();
19    for (int j = 0; j < m; ++j) {
20        x[j] = 1.0;
21        if (j > 0) x[j - 1] = 0.0;
22        Mt.col(j) = mvLLRPartMat(llrcM, x);
23    }
24    return {std::sqrt((M - Mt).squaredNorm() / (n * m)),
25            std::sqrt(M.squaredNorm() / (n * m))};
26 }
```

HINT 1 for (2-4.k): Take into account [NumCSE course → ??]. Testing whether $\text{err} = 0$ is pointless, because err is the result of a numerical computation that is inevitably affected by roundoff errors. Test, whether

$$\text{err} \leq \cdot \text{EPS} \|\mathbf{M}\|_F, \quad \text{EPS} \approx \text{machine precision} .$$

┘

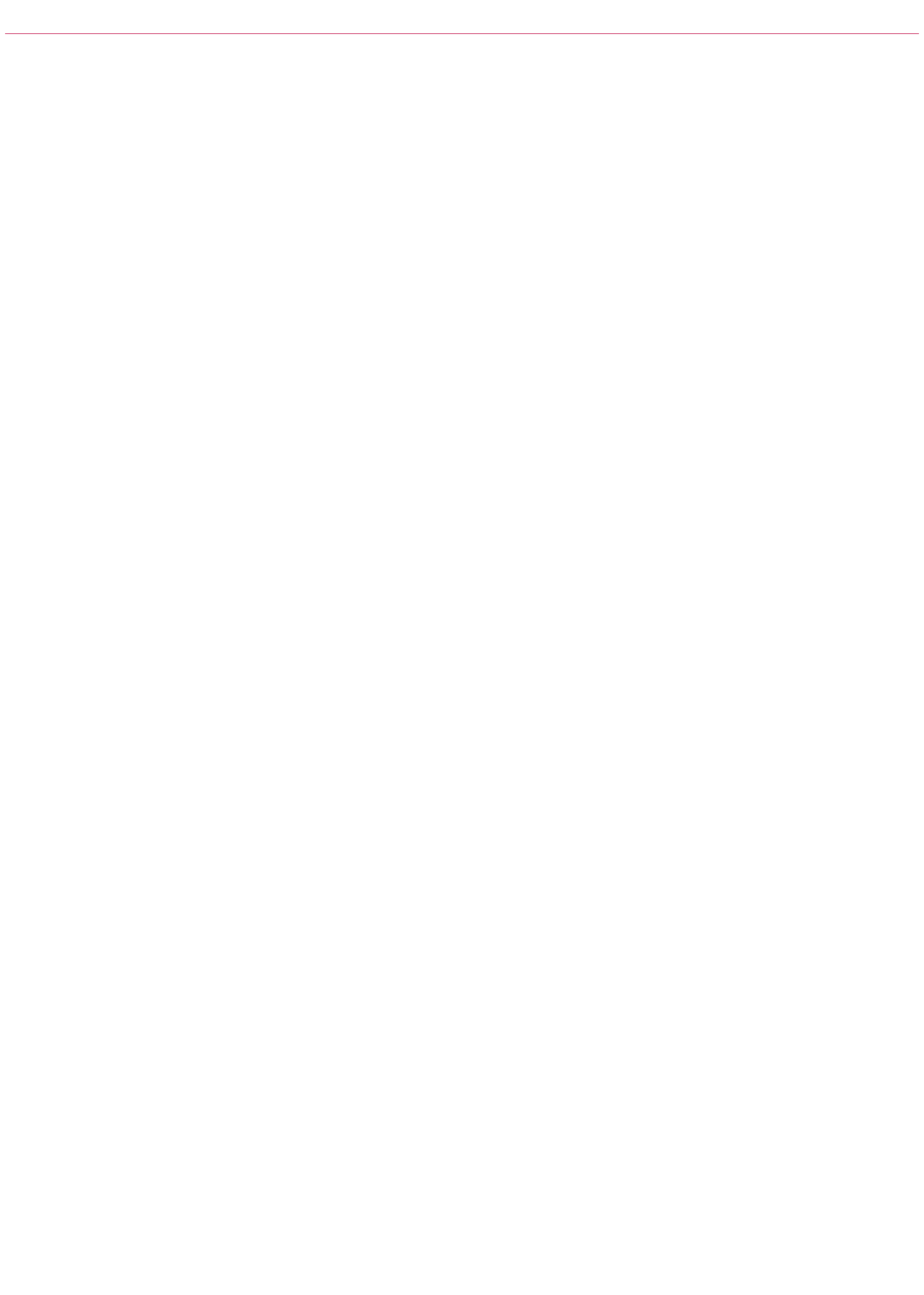
SOLUTION of (2-4.k):

C++ code 2.4.23: validateLLR(): Validation of local low-rank compression code
→[GITLAB](#)

```

2  bool validateLLR(unsigned int q, double tol, double eta) {
3      const int npts = 64; // Number of collocation points
4      // Initialize collocation points (the same in x/y-direction)
5      std::vector<HMAT::Point<1>> pts;
6      for (int n = 0; n < npts; n++) {
7          HMAT::Point<1> p;
8          p.idx = n;
9          p.x[0] = static_cast<double>(n) / (npts - 1);
10         pts.push_back(p);
11     }
12     // Allocate cluster tree object (the same for both directions)
13     auto T_row = std::make_shared<
14         KernMatLLRApprox::LLRClusterTree<KernMatLLRApprox::InterpNode<1>>>(q);
15     T_row->init(pts);
16     auto T_col = std::make_shared<
17         KernMatLLRApprox::LLRClusterTree<KernMatLLRApprox::InterpNode<1>>>(q);
18     T_col->init(pts);
19
20     // Loop over polynomials up to degree q and check whether for a
21     // polynomial
22     // kernel the approximation error will vanish.
23     for (unsigned int k = 0; k < q; ++k) {
24         for (unsigned int l = 0; l < q; ++l) {
25             struct PolynomialKernel {
26                 explicit PolynomialKernel() : deg_x_(0), deg_y_(0) {}
27                 PolynomialKernel(unsigned int deg_x, unsigned int deg_y)
28                     : deg_x_(deg_x), deg_y_(deg_y) {}
29                 double operator()(double x, double y) const {
30                     return std::pow(x, deg_x_) * std::pow(y, deg_y_);
31                 }
32                 unsigned int deg_x_;
33                 unsigned int deg_y_;
34             } G(k, l);
35             // Initialize local low-rank compressed matrix data structure
36             //
37             KernMatLLRApprox::BiDirChebPartMat1D<PolynomialKernel> Mt(T_row, T_col, G,
38                 q, eta);
39             std::cout << "validateLLR: BlockPartition: " << Mt.ffb_cnt
40                 << " far field blocks, " << Mt.nfb_cnt << " near-field blocks"
41                 << std::endl;
42             auto [error, norm] = approxErrorLLR<PolynomialKernel>(Mt);
43             std::cout << "Kernel (x,y) -> x^" << k << " * y^" << l
44                 << ": rel error = " << error / norm << std::endl;
45             if (error > tol * norm) {
46                 return false;
47             }
48         }
49     }
50     return true;
51 }

```



SOLUTION of (2-4.1):

C++ code 2.4.25: validateLLR(): Validation of local low-rank compression code
→[GITLAB](#)

```

2  struct LogKernel {
3      LogKernel() = default;
4      double operator()(double x, double y) const {
5          const double d = std::abs(x - y);
6          return ((d > (std::numeric_limits<double>::epsilon() * std::abs(x + y)))
7                  ? (-std::log(d))
8                  : 0.0);
9      }
10 };
11
12 void tabulateConvergenceLLR(std::vector<unsigned int> &&n_vec,
13                             std::vector<unsigned int> &&q_vec, double eta) {
14     LogKernel G;
15     Eigen::MatrixXd relerr(n_vec.size(), q_vec.size());
16     for (int n_idx = 0; n_idx < n_vec.size(); ++n_idx) {
17         for (int q_idx = 0; q_idx < q_vec.size(); ++q_idx) {
18             // Create equidistant points
19             std::vector<HMAT::Point<1>> pts;
20             for (int pt_idx = 0; pt_idx < n_vec[n_idx]; pt_idx++) {
21                 HMAT::Point<1> p;
22                 p.idx = pt_idx;
23                 p.x[0] = static_cast<double>(pt_idx) / (n_vec[n_idx] - 1);
24                 pts.push_back(p);
25             }
26             // Allocate cluster tree objects (the same for both directions)
27             auto T_row = std::make_shared<
28                 KernMatLLRAprox::LLRClusterTree<KernMatLLRAprox::InterpNode<1>>>(
29                 q_vec[q_idx]);
30             T_row->init(pts);
31             auto T_col = std::make_shared<
32                 KernMatLLRAprox::LLRClusterTree<KernMatLLRAprox::InterpNode<1>>>(
33                 q_vec[q_idx]);
34             T_col->init(pts);
35             // Build local low-rank compressed matrix
36             KernMatLLRAprox::BiDirChebPartMat1D<LogKernel> Mt(T_row, T_col, G,
37                 q_vec[q_idx], eta);
38             std::cout << "tabulateConvergenceLLR: q=" << q_vec[q_idx]
39                 << ", n_pts = " << n_vec[n_idx]
40                 << " : BlockPartition: " << Mt.ffb_cnt << " far field blocks, "
41                 << Mt.nfb_cnt << " near-field blocks" << std::endl;
42             auto [error, norm] = approxErrorLLR<LogKernel>(Mt);
43             std::cout << "abs err = " << error << ", rel err = " << error / norm
44                 << std::endl;
45             relerr(n_idx, q_idx) = error / norm;
46         }
47     }
48
49     std::cout << "Relative errors of local low-rank matrix approximation"
50         << std::endl;
51     std::cout << "n\\q ";
52     for (int q_idx = 0; q_idx < q_vec.size(); ++q_idx) {

```

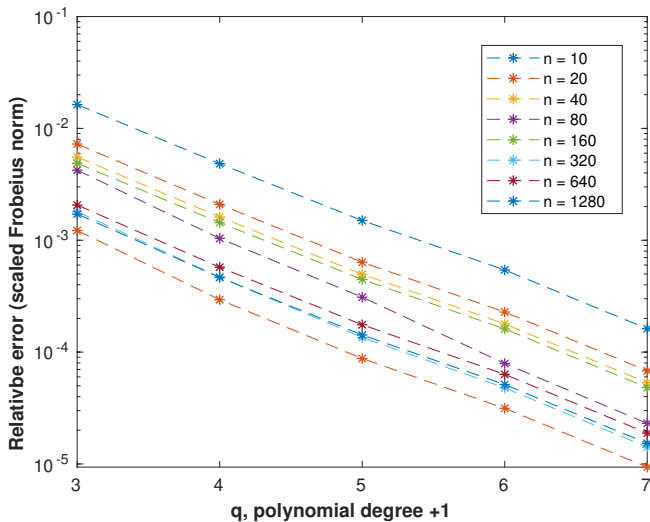
```

53     std::cout << std::setw(15) << q_vec[q_idx];
54 }
55 std::cout << std::endl;
56 for (int n_idx = 0; n_idx < n_vec.size(); ++n_idx) {
57     std::cout << std::setw(15) << n_vec[n_idx];
58     for (int q_idx = 0; q_idx < q_vec.size(); ++q_idx) {
59         std::cout << std::setw(15) << relerr(n_idx, q_idx);
60     }
61     std::cout << std::endl;
62 }
63 }

```

Error table produced by the code:

n \ q	3	4	5	6	7
10	0.0163202	0.00483677	0.00150076	0.000542133	0.000161866
20	0.00725101	0.00209287	0.000635818	0.000228238	6.79452e-05
40	0.00555359	0.00161781	0.000496389	0.000178737	5.32868e-05
60	0.00423128	0.00103931	0.000309776	7.91086e-05	2.30836e-05
80	0.00488024	0.00143791	0.000446219	0.000161221	4.8142e-05
160	0.00180033	0.000467991	0.000135282	4.78552e-05	1.41545e-05
320	0.00206326	0.000574123	0.000175616	6.32425e-05	1.88628e-05
640	0.0017143	0.000464783	0.00014227	5.12812e-05	1.53061e-05
1280	0.00122254	0.000294242	8.76713e-05	3.14384e-05	9.37263e-06



◁ In a lin-log plot the approximation errors trace out almost straight lines.

We see empiric **exponential convergence** of the approximation error as a function of the polynomial degree q of bi-directional Chebychev interpolation in the far-field. This is in agreement with the considerations of [Lecture → ??].

We also observe that the (scaled) approximation error does not depend strongly on the matrix size.

HINT 1 for (2-4.m): This code snippet shows the use of the C++ chrono library for measuring runtimes:

C++ code 2.4.26: Measuring runtime of a part of a C++ code.

```
2 void runtimeMeasuredDemo(void) {
3     auto t1 = std::chrono::high_resolution_clock::now();
4     double s = 0.0;
5     for (long int i = 0; i < 10000000; ++i) {
6         s += 1.0 / std::sqrt((double)i);
7     }
8     auto t2 = std::chrono::high_resolution_clock::now();
9     /* Getting number of milliseconds as a double. */
10    std::chrono::duration<double, std::milli> ms_double = t2 - t1;
11    std::cout << "Runtime = " << ms_double.count() << "ms\n";
12 }
```



For runtime measurements it is essential to compile your code in *Release Mode*: configure your CMake build system accordingly!

C++ code 2.4.27: runtimeMatVec () : runtime measurements for matrix×vector →GITLAB

```

2 void runtimeMatVec(std::vector<unsigned int> &n_vec, unsigned int n_runs,
3                 unsigned int q, double eta) {
4     std::cout << "runtimeMatVec(n_runs = " << n_runs << ", q = " << q
5                 << ", eta = " << eta << ")" << std::endl;
6     LogKernel G;
7     unsigned int n_cnt = n_vec.size();
8     for (unsigned int n : n_vec) {
9         // Create equidistant points
10        std::vector<HMAT::Point<1>> pts;
11        for (int pt_idx = 0; pt_idx < n; pt_idx++) {
12            HMAT::Point<1> p;
13            p.idx = pt_idx;
14            p.x[0] = static_cast<double>(pt_idx) / (n - 1);
15            pts.push_back(p);
16        }
17        // Allocate cluster tree objects (the same for both directions)
18        auto T_row = std::make_shared<
19            KernMatLLRApprox::LLRClusterTree<KernMatLLRApprox::InterpNode<1>>>(q);
20        T_row->init(pts);
21        auto T_col = std::make_shared<
22            KernMatLLRApprox::LLRClusterTree<KernMatLLRApprox::InterpNode<1>>>(q);
23        T_col->init(pts);
24        // Build local low-rank compressed matrix
25        KernMatLLRApprox::BiDirChebPartMat1D<LogKernel> Mt(T_row, T_col, G, q, eta);
26        const size_t nrows = Mt.rows();
27        const size_t ncols = Mt.cols();
28        Eigen::VectorXd x = Eigen::VectorXd::Constant(ncols, 1.0);
29        Eigen::VectorXd y(nrows);
30        // Average runtimes over n_run runs
31        double ms_time = std::numeric_limits<double>::max();
32        for (int r = 0; r < n_runs; ++r) {
33            auto t1 = std::chrono::high_resolution_clock::now();
34            y = mvLLRPartMat(Mt, x);
35            auto t2 = std::chrono::high_resolution_clock::now();
36            /* Getting number of milliseconds as a double. */
37            std::chrono::duration<double, std::milli> ms_double = (t2 - t1);
38            ms_time = std::min(ms_time, ms_double.count());
39        }
40        std::cout << "n = " << n << ": " << ms_time << " ms, #nfb = " << Mt.nfb_cnt
41                << ", #ffb = " << Mt.ffb_cnt << std::endl;
42    }
43 }

```

We observe that a doubling of the matrix size n leads to a *fourfold* increase of the computing time, which hints at an asymptotic computational complexity $O(n^2)$ for $n \rightarrow \infty$, which does not match the expected $O(n \log n)$ asymptotic effort.

However, as expected, $\#F_{\text{far}}$ and $\#F_{\text{near}}$ increase only (nearly) linearly with n .

n	time	$\#F_{\text{far}}$	$\#F_{\text{near}}$
1024	3.83465 ms	2048	2504
2048	13.9696 ms	4096	5058
4096	53.0404 ms	8192	10172
8192	215.414 ms	16384	20406
16384	875.852 ms	32768	40880
32768	3483.78 ms	65536	81834
65536	14299.7 ms	131072	163748
131072	65480.2 ms	262144	327582
262144	282803	524288	655256

SOLUTION of (2-4.b):

- It suffices to test for (ii) and (iii), when we accept that the index set \mathbb{I} is defined by $T.root \rightarrow \mathbb{I}()$. These two requirements can be checked “locally” for every node that has children.
- When retrieving the index set $\mathcal{I}(w)$ of a node w (via $w.\mathbb{I}()$) and of those of its children v_1 and v_2 , we first sort them using `std::sort()`.
- Finally we can traverse the sorted sequences in parallel and simultaneously verify [Lecture \rightarrow ??] and [Lecture \rightarrow ??].

An alternative approach relies on a mapping $v : \{1, \dots, \max \mathbb{I}\} \rightarrow \{1, \dots, \#\mathbb{I}\}$, which returns for each index its position in the index array. For each node w with children v_1 and v_2 we do the following:

1. Initialize an vector c of integers of length $\#\mathbb{I}$ with zero.
2. $c[i] = 1$, if $i \in v(\mathcal{I}(w))$.
3. $c[i] += 1$, if $i \in v(\mathcal{I}(v_1))$.
4. $c[i] += 1$, if $i \in v(\mathcal{I}(v_2))$.
5. If $c[i] \notin \{0, 2\}$ for some index, then the test has failed.

This is to be embedded into a recursion.

SOLUTION of (2-4.d):

1. We first extract the index sets \mathbb{I}_x and \mathbb{I}_y (sequences of non-negative integers in this case) from the root of the cluster trees `*partmat.rowT` and `*partmat.colT`. Those define the column and row index sets for the matrix.
2. We determine the cardinalities $n := \#\mathbb{I}_x \in \mathbb{N}$ and $m := \#\mathbb{I}_y \in \mathbb{N}$ of these index sets and introduce bijective mappings $v_x : \{1, \dots, n\} \rightarrow \mathbb{I}_x$ and $v_y : \{1, \dots, m\} \rightarrow \mathbb{I}_y$.
3. We initialize an integer $m \times n$ -matrix C with zero.
4. For every index block B in both the near field and the far field do

$$C(k, \ell) += 1, \quad \text{if } (k, \ell) \in v_x(\mathcal{I}(B.nx)) \times v_y(\mathcal{I}(B.ny)).$$

5. If $C(k, \ell) \neq 1$ for some $k \in \{1, \dots, m\}$, $\ell \in \{1, \dots, n\}$, then the test has failed.

C++ code 2.4.3: Implementation of `checkMatrixPartition()` → [GITLAB](#)

```

2  template <typename NODE>
3  bool checkMatrixPartition (
4      const HMAT::BlockPartition<NODE, HMAT::IndexBlock<NODE>,
5          HMAT::IndexBlock<NODE>> &partmat) {
6  assertm((partmat.rowT and partmat.colT), "Missing trees!");
7  // Index sets I, J underlying the row/col cluster trees
8  const std::vector<std::size_t> idxset_row{partmat.rowT->root->l()};
9  const std::vector<std::size_t> idxset_col{partmat.colT->root->l()};
10 // Array for remapping indices to integers from 0 to
11 // #I/J
12 const std::size_t maxidx_row =
13     *std::max_element(idxset_row.begin(), idxset_row.end());
14 const std::size_t maxidx_col =
15     *std::max_element(idxset_col.begin(), idxset_col.end());
16 // A bit of a gamble: indices ≈ #I, #J
17 assertm((maxidx_row < 2 * idxset_row.size()) and
18         (maxidx_col < 2 * idxset_col.size()),
19         "Maximal index too large");
20 std::vector<int> remap_idx_row(maxidx_row + 1, -1);
21 std::vector<int> remap_idx_col(maxidx_col + 1, -1);
22 for (int k = 0; k < idxset_row.size(); ++k) remap_idx_row[idxset_row[k]] = k;
23 for (int k = 0; k < idxset_col.size(); ++k) remap_idx_col[idxset_col[k]] = k;
24 // Matrix for keeping track of occurrences of matrix entries
25 Eigen::MatrixXi C(idxset_row.size(), idxset_col.size());
26 C.setZero();
27
28 // Run through all far-field blocks
29 for (const auto &ffb : partmat.farField) {
30     for (std::size_t row_idx : ffb.i_idx) {
31         for (std::size_t col_idx : ffb.j_idx) {
32             if ((row_idx > maxidx_row) or (col_idx > maxidx_col)) return false;
33             const int pos_row = remap_idx_row[row_idx];
34             const int pos_col = remap_idx_col[col_idx];
35             if ((pos_row < 0) or (pos_col < 0)) return false;
36             C(pos_row, pos_col) += 1;
37         }
38     }

```

```
39 }
40 // Run through all near-field blocks: samme code
41 for (const auto &nfb : partmat.nearField) {
42     for (std::size_t row_idx : nfb.i_idx) {
43         for (std::size_t col_idx : nfb.j_idx) {
44             if ((row_idx > maxidx_row) or (col_idx > maxidx_col)) return false;
45             const int pos_row = remap_idx_row[row_idx];
46             const int pos_col = remap_idx_col[col_idx];
47             if ((pos_row < 0) or (pos_col < 0)) return false;
48             C(pos_row, pos_col) += 1;
49         }
50     }
51 }
52 // Check whether matrix contains entries != 1
53 return (C.array() == 1).all();
54 }
```

HINT 1 for (2-4.e):

- Remember the formula for the q Chebychev nodes on a general interval $[a, b]$:

$$t_j := a + \frac{1}{2}(b - a) \left(\cos\left(\frac{2j-1}{2q}\pi\right) + 1 \right), \quad j = 1, \dots, q. \quad [\text{Lecture} \rightarrow ??]$$

- Useful is the barycentric representation of the Lagrange polynomials for Chebychev interpolation

$$L_\ell(x) = \begin{cases} \frac{\lambda_\ell}{x - t_\ell} \cdot \left(\sum_{i=1}^q \frac{\lambda_i}{x - t_i} \right)^{-1}, & x \neq t_k, \quad k = 1, \dots, q, \\ 1, & x = t_\ell, \\ 0, & x = t_k, \quad k \neq \ell, \end{cases} \quad [\text{Lecture} \rightarrow ??]$$

with barycentric weights from Eq. (2.3.7) rescaled to fit the general interval $[a, b]$:

$$\lambda_i = 2^{q-1} \frac{\hat{\lambda}_i}{(b-a)^{q-1}}, \quad \hat{\lambda}_i := \frac{2^{q-1}}{q} (-1)^{i-1} \sin\left(\frac{2i-1}{2q}\pi\right), \quad i = 1, \dots, q. \quad (2.4.6)$$

┘

SOLUTION of (2-4.e):

1. Retrieve the bounding box of the current cluster w , which will be an interval $[a, b]$.
2. Compute the barycentric weights λ_i from (2.4.6).
3. We compute \mathbf{V}_w row-by-row looping over the collocation points held by the current cluster. Those can be accessed through the data member `CtNode::pts`, see [Lecture → ??].
4. For the collocation point ξ_j we evaluate the denominator in [Lecture → ??]:

$$\tau := \sum_{i=1}^q \frac{\lambda_i}{\xi_j - t_i} \quad \text{if } \xi_j \neq t_k.$$

The special cases from [Lecture → ??] must also be treated.

5. Then set

$$(\mathbf{V}_w)_{j,\ell} = L_\ell(\xi_j) = \begin{cases} \frac{1}{\tau} \frac{\lambda_\ell}{\xi_j - t_\ell} & , \text{ if } \xi_j \notin \{t_k\}, \\ \delta_{\ell,j} & \text{else,} \end{cases} \quad \begin{array}{l} \ell = 1, \dots, q, \\ j = 1, \dots, \#\mathcal{I}(w). \end{array}$$

C++ code 2.4.7: Implementation of `InterpNode::initV()` →GITLAB

```
2 template <int DIM>
3 void InterpNode<DIM>::initV () {
4     static_assert(DIM == 1, "Implemented only for 1D");
5     // Retrieve bounding box, explanation for the this pointer
6     const HMAT::BBox<DIM> bbox(this->pts);
7     // Find interval corresponding to the bounding box of the current
8     // cluster
9     const double a = bbox.minc[0];
10    const double b = bbox.maxc[0];
11    // Compute Chebychev nodes t_i and
12    // barycentric weights lambda_i for interval [a,b]
13    Eigen::VectorXd lambda(q); // barycentric weights
14    Eigen::VectorXd t(q); // Chebychev nodes
15    const double fac = std::pow(4.0 / (b - a), q - 1) / q;
16    int sgn = 1;
17    for (int i = 0; i < q; i++, sgn *= -1) {
18        const double arg = (2.0 * i + 1.0) / (2 * q) * M_PI;
19        t[i] = a + 0.5 * (b - a) * (std::cos(arg) + 1.0);
20        lambda[i] = fac * sgn * std::sin(arg); // (2.4.6)
21    }
22    // Number of collocation points in cluster
23    const unsigned int nlw = (this->pts).size();
24    // Traverse collocation points (in the interval [a,b])
25    Eigen::VectorXd tx_diff(q); // t_i - xi
26    for (int j = 0; j < nlw; ++j) {
27        // Coordinate of current collocation point
28        const double xi = ((this->pts)[j]).x[0];
29        double tau = 0.0; // tau := sum_{i=1}^q lambda_i / (xi - t_i)
30        bool on_node = false;
31        for (int i = 0; i < q; i++) {
32            tx_diff[i] = xi - t[i];
33            // Avoid division by zero
```



```
33     if (tx_diff[i] == 0.0) {
34         on_node = true;
35         V.row(j).setZero();
36         V(j, i) = 1.0; //  $(\mathbf{V})_{j,:} = \mathbf{e}_i^T$  when hitting a node
37         break; // The  $j$ -th row of  $\mathbf{V}$  is complete already
38     } else {
39         const double txdl = lambda[i] / tx_diff[i];
40         V(j, i) = txdl;
41         tau += txdl;
42     }
43 }
44 if (!on_node) V.row(j) /= tau;
45 }
46 }
```

SOLUTION of (2-4.f):

The data members G and q are just copies of constructor arguments. As discussed in [Lecture → ??], if the **BiDirChebInterpBlock** object represents the block $v \times w$, v, w two clusters, the data member C has to hold the matrix

$$C|_{v \times w} := \left[G(t_v^k, t_w^\ell) \right]_{k, \ell=1, \dots, q} \in \mathbb{R}^{q, q}, \quad (2.4.9)$$

where t_v^k, t_w^ℓ are the Chebychev nodes in v and w , respectively.

How are these Chebychev nodes defined? Each block $v \times w$ of the matrix partition comes with a pair of bounding boxes belonging to the involved clusters v and w . In the current one-dimensional setting let us write $[a, b]$ and $[c, d]$ ($a < b, c < d$) for these bounding boxes. They can be fetched via the `getBBox()` member function of **Node**. Then we can resort to [Lecture → ??] and find

$$t_v^k := a + \frac{1}{2}(b - a) \left(\cos\left(\frac{2k - 1}{2q}\pi\right) + 1 \right), \quad k = 1, \dots, q, \quad (2.4.10a)$$

$$t_w^\ell := c + \frac{1}{2}(d - c) \left(\cos\left(\frac{2\ell - 1}{2q}\pi\right) + 1 \right), \quad \ell = 1, \dots, q. \quad (2.4.10b)$$

Then, in light of (2.4.9) a double-loop initialization of $C_{v \times w}$ is straightforward.

C++ code 2.4.11: Constructor of **BiDirChebInterpBlock** →GITLAB

```
2 template <class NODE, typename KERNEL>
3 BiDirChebInterpBlock<NODE, KERNEL>::BiDirChebInterpBlock(
4     NODE &_nx, NODE &_ny, KERNEL _Gfun, std::size_t _q)
5     : HMAT::IndexBlock<NODE>(_nx, _ny), G(std::move(_Gfun)), q(_q), C(_q, _q) {
6     static_assert(NODE::dim == 1, "Only implemented in 1D");
7     // Obtain bounding boxes, here intervals
8     std::array<HMAT::BBox<1>, 2> bboxes = {_nx.getBBox(), _ny.getBBox()};
9     std::array<Eigen::Vector2d, 2> intv;
10    for (int d = 0; d < 2; ++d) {
11        intv[d] = Eigen::Vector2d(bboxes[d].minc[0], bboxes[d].maxc[0]);
12    }
13    // Compute Chebychev nodes for nodal intervals, [Lecture → ??]
14    Eigen::MatrixXd t(2, q);
15    for (int j = 0; j < q; ++j) {
16        // Chebychev nodes on standard interval
17        const double cosval = std::cos((2.0 * j + 1.0) / (2 * q) * M_PI);
18        for (int d = 0; d < 2; ++d) {
19            // Code for (2.4.10a) & (2.4.10b)
20            t(d, j) = intv[d][0] + 0.5 * (intv[d][1] - intv[d][0]) * (cosval + 1.0);
21        }
22    }
23    // Fill matrix C: (C)k,ℓ = G(tk, tℓ), see (2.4.9)
24    for (int k = 0; k < q; ++k) {
25        for (int j = 0; j < q; ++j) {
26            C(k, j) = G(t(0, k), t(1, j));
27        }
28    }
29 }
```

SOLUTION of (2-4.g):

A near-field block $v \times w$ just represents a sub-matrix of the kernel collocation matrix. So, using Ass. 2.4.4,

$$\mathbf{M}_{\text{loc}} \leftrightarrow \left[G(\xi_i, \xi_j) \right]_{\substack{i \in \mathcal{I}(v) \\ j \in \mathcal{I}(w)}} \in \mathbb{R}^{\#\mathcal{I}(v), \#\mathcal{I}(w)}. \quad (2.4.13)$$

The collocation points ξ_i can be accessed through the `pts` data member of the nodes `nx` and `ny`, see [Lecture → ??].

C++ code 2.4.14: Constructor of **NearFieldBlock** → **GITLAB**

```
2 template <class NODE, typename KERNEL>
3 NearFieldBlock<NODE, KERNEL>::NearFieldBlock(NODE &_nx, NODE &_ny, KERNEL _Gfun)
4   : HMAT::IndexBlock<NODE>(_nx, _ny),
5     G(std::move(_Gfun)),
6     Mloc(_nx.pts.size(), _ny.pts.size()) {
7   static_assert(NODE::dim == 1, "Only implemented in 1D");
8   // Direct initialization of near field kernel collocation matrix
9   for (int i = 0; i < _nx.pts.size(); ++i) {
10     for (int j = 0; j < _ny.pts.size(); ++j) {
11       Mloc(i, j) = G((_nx.pts[i]).x[0], (_ny.pts[j]).x[0]);
12     }
13   }
14 }
```

SOLUTION of (2-4.h):

You may use two **hash maps** `hmapI` and `hmapJ` with

- key type **const NODE *** and
- value type **unsigned int** (initialized with zero).

We run through all the block of the matrix partitioning \mathbb{F} . Whenever a cluster pair $(v, w) \in \mathcal{T}_I \times \mathcal{T}_J$ occurs in either the far field or the near field, then we do `hmapI[v] += 1` and `hmapJ[w] += 1`. In the end we find the maximum of all values stored in `hmapI` and `hmapJ`, which yields `spm(F)`.

A hash map data structure is provided by the C++ standard library through the data type `std::unordered_map`.

C++ code 2.4.19: Implementation of `computeSparsityMeasure()` →GITLAB

```
2 template <typename NODE, typename FFB, typename NFB>
3 unsigned int computeSparsityMeasure(
4     const HMAT::BlockPartition<NODE, FFB, NFB> &blockpart,
5     std::ostream *out = nullptr) {
6     assertm((blockpart.rowT and blockpart.colT), "Missing trees!");
7     // Set up hash maps for nodes of both trees
8     using nf_node_t = typename NFB::node_t;
9     using hashmap_t = std::unordered_map<const nf_node_t *, int>;
10    using keyval_t = typename hashmap_t::value_type;
11    hashmap_t nodemap_row;
12    hashmap_t nodemap_col;
13    // Maximal node counts for row and column clusters
14    int xnode_maxcnt = 0;
15    int ynode_maxcnt = 0;
16    // Run through all far-field blocks and also determine maximal cluster
17    // count
18    for (const auto &ffb : blockpart.farField) {
19        // If the current block is based on a cluster, increment that
20        // cluster's
21        // count in the hash map
22        if (nodemap_row.find(&ffb.nx) == nodemap_row.end()) {
23            nodemap_row[&ffb.nx] = 1;
24            xnode_maxcnt = std::max(xnode_maxcnt, 1);
25        } else {
26            xnode_maxcnt = std::max(xnode_maxcnt, nodemap_row[&ffb.nx] += 1);
27        }
28        if (nodemap_col.find(&ffb.ny) == nodemap_col.end()) {
29            nodemap_col[&ffb.ny] = 1;
30            ynode_maxcnt = std::max(ynode_maxcnt, 1);
31        } else {
32            ynode_maxcnt = std::max(ynode_maxcnt, nodemap_col[&ffb.ny] += 1);
33        }
34    }
35    // Run through all near-field blocks
36    for (const auto &nfb : blockpart.nearField) {
37        // If the current block is based on a cluster, increment that
38        // cluster's
39        // count in the hash map
40        if (nodemap_row.find(&nfb.nx) == nodemap_row.end()) {
41            nodemap_row[&nfb.nx] = 1;
```

```

39     xnode_maxcnt = std::max(xnode_maxcnt, 1);
40 } else {
41     xnode_maxcnt = std::max(xnode_maxcnt, nodemap_row[&nfb.nx] += 1);
42 }
43 if (nodemap_col.find(&nfb.ny) == nodemap_col.end()) {
44     nodemap_col[&nfb.ny] = 1;
45     ynode_maxcnt = std::max(ynode_maxcnt, 1);
46
47 } else {
48     ynode_maxcnt = std::max(ynode_maxcnt, nodemap_col[&nfb.ny] += 1);
49 }
50 }
51 // Print node statistics if requested
52 if (out) {
53     auto output = [](const hashmap_t &nodemap, std::ostream *out) -> void {
54         if (out) {
55             for (const keyval_t kv : nodemap) {
56                 const nf_node_t *node_p = kv.first;
57                 const int node_count = kv.second;
58                 const std::vector<size_t> idxs{node_p->l()};
59                 (*out) << "Node with idx set = ";
60                 for (size_t idx : idxs) {
61                     (*out) << idx << ' ';
62                 }
63                 (*out) << " occurs " << node_count << " times " << std::endl;
64             }
65         }
66     };
67     (*out) << "ROW CLUSTER TREE" << std::endl;
68     output(nodemap_row, out);
69     (*out) << "COLUMN CLUSTER TREE" << std::endl;
70     output(nodemap_col, out);
71 }
72 return std::max(xnode_maxcnt, ynode_maxcnt);
73 }

```

Remark. The index-based design discussed in ?? would lead to a simpler code for `computeSparsityM`

HINT 1 for (2-4.i): You may use the data members `clust_sect_vec` of **HMAT::CtNode** and `clust_omega` of **KernMatLLRAprox::InterpNode** to store temporary data in the clusters of the cluster trees underlying `llrcmat`. └

SOLUTION of (2-4.i):

We adapt the algorithm outlined in [Lecture → ??].

C++ code 2.4.20: mvLLRPartMat () : matrix×vector for compressed matrix →GITLAB

```
2  template <class NODE, typename FFB, typename NFB>
3  Eigen::VectorXd mvLLRPartMat(BiDirChebBlockPartition<NODE, FFB, NFB> &llrcmat,
4                               const Eigen::VectorXd &x) {
5      using ff_node_t = typename FFB::node_t;
6      using nf_node_t = typename NFB::node_t;
7      // Verify requirement of contiguous indices
8      const std::vector<size_t> col_idxes = (llrcmat.colT->root)->l();
9      assertm((*std::max_element(col_idxes.begin(), col_idxes.end()) ==
10             col_idxes.size() - 1),
11             "col_idxes not contiguous");
12      const std::vector<size_t> row_idxes = (llrcmat.rowT->root)->l();
13      assertm((*std::max_element(row_idxes.begin(), row_idxes.end()) ==
14             row_idxes.size() - 1),
15             "row_idxes not contiguous");
16      assertm(x.size() == col_idxes.size(), "Wrong size of vector x");
17      Eigen::VectorXd y(row_idxes.size()); // Return value
18      y.setZero();
19      // Pass (I) of [Lecture → ??]: Reduce to cluster for column tree,
20      // computation of
21      //  $\vec{\omega}_w := \mathbf{V}_w^T \mathbf{R}_w \vec{\mu} \in \mathbb{R}^{\#I(w)}$ 
22      std::function<void(NODE * col_node)> comp_omega_rec =
23          [&](NODE *col_node) -> void {
24              if (col_node) {
25                  // Obtain indices held by the cluster
26                  const std::vector<size_t> idxs = col_node->l();
27                  for (int j = 0; j < idxs.size(); ++j) {
28                      // Restriction of argument vector to column cluster
29                      (col_node->clust_sect_vec)[j] = x[idxs[j]];
30                      // Compute  $\vec{\omega}_w$ 
31                      col_node->clust_omega =
32                          (col_node->V).transpose() * col_node->clust_sect_vec;
33                  }
34                  // Recursion: visit entire cluster tree
35                  comp_omega_rec(col_node->sons[0]);
36                  comp_omega_rec(col_node->sons[1]);
37              }
38          };
39      comp_omega_rec(llrcmat.colT->root);
40
41      // Final sentence in [Lecture → ??]: Clear local storage of row tree
42      std::function<void(NODE * ipnode)> clear_vec_rec = [&](NODE *ipnode) -> void {
43          if (ipnode) {
44              ipnode->clust_sect_vec.setZero();
45              ipnode->clust_omega.setZero();
46              clear_vec_rec(ipnode->sons[0]);
47              clear_vec_rec(ipnode->sons[1]);
48          }
49      };
50      clear_vec_rec(llrcmat.rowT->root);
```

```

51 // Pass (II) in [Lecture → ??]: Do block-based computations
52 // Visit far-field blocks:
53 for (auto &ffb : llrcmat.farField) {
54     // Far-field block: accumulate  $C_{v \times w} \vec{\omega}_w$ 
55     ff_node_t &row_node = ffb.nx;
56     const ff_node_t &col_node = ffb.ny;
57     row_node.clust_omega += ffb.C * col_node.clust_omega;
58 }
59 for (auto &nfb : llrcmat.nearField) {
60     // Near-field block: use restricted kernel collocatin matrix  $M|_{v \times w}$ 
61     nf_node_t &row_node = nfb.nx;
62     const nf_node_t &col_node = nfb.ny;
63     row_node.clust_sect_vec += nfb.Mloc * col_node.clust_sect_vec;
64 }
65
66 // Pass (III) of [Lecture → ??]: Traverse row tree and assemble
67 // return vector
68 std::function<void(NODE * row_node)> ass_y_rec = [&](NODE *row_node) -> void {
69     if (row_node) {
70         const std::vector<size_t> idxs = row_node->l();
71         //  $x|_v + = U_v \vec{\zeta}_v + \vec{\phi}_v$ 
72         row_node->clust_sect_vec += row_node->V * row_node->clust_omega;
73         // Expand from cluster v
74         for (int j = 0; j < idxs.size(); ++j) {
75             y[idxs[j]] += row_node->clust_sect_vec[j];
76         }
77         // Recursion through row cluster tree
78         ass_y_rec(row_node->sons[0]);
79         ass_y_rec(row_node->sons[1]);
80     }
81 };
82 ass_y_rec(llrcmat.rowT->root);
83 return y;
84 }

```

Remark. The index-based approach discussed in ?? could bring significant benefits to the implementation of `mvLLRPartMat()`. In particular it could help curb memory fragmentation.

SOLUTION of (2-5.j):

We expect *exponential convergence* of $\text{err}(q)$ for $q \rightarrow \infty$.

HINT 1 for (2-5.k): This code snippet shows the use of the C++ `chrono` library for measuring runtimes:

C++ code 2.5.16: Measuring runtime of a part of a C++ code.
Get it on [🐱 GitLab \(gravitationalforces.cpp\)](#).

```
2 void runtimeMeasuredDemo(void) {
3     auto t1 = std::chrono::high_resolution_clock::now();
4     double s = 0.0;
5     for (long int i = 0; i < 10000000; ++i) {
6         s += 1.0 / std::sqrt((double)i);
7     }
8     auto t2 = std::chrono::high_resolution_clock::now();
9     /* Getting number of milliseconds as a double. */
10    std::chrono::duration<double, std::milli> ms_double = t2 - t1;
11    std::cout << "Runtime = " << ms_double.count() << "ms\n";
12 }
```



For runtime measurements it is essential to compile your code in *Release Mode*: configure your CMake build system accordingly!

SOLUTION of (2-5.k):

HINT 1 for (2-5.b): Note that

$$p(\ell/n) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i \frac{j\ell}{n}), \quad \ell \in \{0, \dots, n-1\},$$

which, in light of

Definition [NumCSE course → ??]. **Discrete Fourier transform (DFT)**

The linear map $\text{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\text{DFT}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform (DFT)**, i.e. for $[c_0, \dots, c_{n-1}] := \text{DFT}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} = \sum_{j=0}^{n-1} y_j \exp(-2\pi i \frac{kj}{n}), \quad k = 0, \dots, n-1. \quad [\text{NumCSE course} \rightarrow ??]$$

means that

$$[p(\ell/n)]_{\ell=0}^{n-1} = \text{DFT}_n \left([\gamma_j]_{j=0}^{n-1} \right). \quad (2.5.5)$$

SOLUTION of (2-5.b):

According to (2.5.5) all we need to do is to invoke a DFT of the coefficient vector, see [[NumCSE course](#) → ??].

C++ code 2.5.6: Evaluation of a trigonometric polynomial at equidistant points →[GITLAB](#)

```
2 Eigen::VectorXcd evalTrigPolyEquid(const Eigen::VectorXcd &coeffs) {
3     //
4     // TODO
5     //
6 }
```

Remark. If you need the value of p only at a single point, do not use DFT, but the plain and simple Horner scheme [[NumCSE course](#) → ??].

HINT 1 for (2-5.c): The cardinal basis $\{b_k\}_{k=0}^{n-1} \subset \mathcal{P}_n^T$ of \mathcal{P}_n^T is uniquely characterized by the formula

$$b_j^{(k/n)} = \delta_{k,j} \Leftrightarrow p(x) = \sum_{k=0}^{n-1} p^{(k/n)} b_j(x), \quad x \in \mathbb{R}, \quad \forall p \in \mathcal{P}_n^T. \quad (2.5.7)$$

┘

HINT 2 for (2-5.c): Use the formula [NumCSE course → ??] for the inverse DFT to express the coefficients γ_j of

$$p(x) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x), \quad x \in [0, 1[. \quad (2.5.3)$$

in terms of the point values $f_k := p(k/n), k = 0, \dots, n-1$. ┘

SOLUTION of (2-5.c):

The interpolation problem in \mathcal{P}_n^T can be solved by means of DFT: let us try to find $p \in \mathcal{P}_n^T$ satisfying the interpolation conditions $p(k/n) = f_k$, $k = 0, \dots, n-1$, for given $f_k \in \mathbb{C}$. Inserting the “monomial representation” (2.5.3) of p this leads to the linear system of equations for the unknown coefficients γ_j

$$\sum_{j=0}^{n-1} \gamma_j \exp(2\pi i j \frac{k}{n}) = f_k, \quad k = 0, \dots, n-1.$$

\Downarrow

$$\text{DFT}_n \vec{\gamma} = \vec{\varphi} := [f_k]_{k=0}^{n-1}.$$

We have an explicit formula for DFT_n^{-1} , which yields

$$\gamma_j := \frac{1}{n} \sum_{k=0}^{n-1} f_k \exp(2\pi i \frac{kj}{n}), \quad j = 0, \dots, n-1.$$

Hence, for $f_k = \delta_{k,\ell}$ for some $\ell \in \{0, \dots, n-1\}$, we find

$$\gamma_j = \frac{1}{n} \exp(2\pi i \frac{\ell j}{n}), \quad j = 0, \dots, n-1.$$

So the ℓ -th cardinal basis function is

$$\begin{aligned} b_\ell(x) &= \frac{1}{n} \sum_{j=0}^{n-1} \exp(2\pi i \frac{\ell j}{n}) \exp(-2\pi i j x) = \frac{1}{n} \sum_{j=0}^{n-1} \exp(2\pi i (\frac{\ell}{n} - x)j) \\ &= \frac{1}{n} \frac{\exp(2\pi i n x) - 1}{\exp(2\pi i (\frac{\ell}{n} - x)) - 1}, \quad x \in [0, 1[, \quad x \notin k/n + \mathbb{Z}. \end{aligned} \tag{2.5.8}$$

where we used the formula for a geometric sum and $\exp(2\pi i j) = 1$ for all $j \in \mathbb{Z}$.

Obviously, $b_\ell(k/n) = 0$ for all $k \in \{0, \dots, n-1\} \setminus \{\ell\}$. Applying L'Hospital's rule for $x \rightarrow \ell/n$, shows that $b_\ell(\ell/n) = 1$. This verifies that b_ℓ from (2.5.8) is the desired trigonometric cardinal basis polynomial.

SOLUTION of (2-5.d):

The implementation is straightforward using a “vectorized **Horner scheme**”, cf. [NumCSE course → ??].
For a single $x \in [0, 1]$ we can write

$$p(x) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x) \tag{2.5.9}$$
$$= \gamma_0 + z \cdot (\gamma_1 + z \cdot (\gamma_2 + z \cdot (\gamma_3 + \dots + \gamma_{n-1} z) \dots)) , \quad z := \exp(-2\pi i x) .$$

The vectorized version is obtained by replacing z with $[\exp(-2\pi i x_\ell)]_{\ell=0}^{m-1}$, and regarding \cdot as the componentwise product of vectors and $+$ as adding the same scalar to every component of a vector.

The asymptotic computational effort will be $O(mn)$ for $n, m \rightarrow \infty$.

C++ code 2.5.10: Exact evaluation of a trigonometric polynomial in many points →GITLAB

```
2 Eigen::VectorXcd evalTrigPoly(const Eigen::VectorXcd &gamma,
3                               const Eigen::VectorXd &x) {
4     //
5     // TODO
6     //
7 }
```

SOLUTION of (2-5.e):

We use the cardinal basis representation of $x \mapsto p(x)$

$$p(x) = \sum_{j=0}^{n-1} p(j/n) b_j(x), \quad x \in \mathbb{R},$$

and the formula (2.5.8):

$$\blacktriangleright \quad p(x) = \sum_{j=0}^{n-1} p(j/n) \frac{1}{n} \frac{\exp(2\pi i n x) - 1}{\exp(2\pi i (\frac{k}{n} - x)) - 1}, \quad x \in]0, 1].$$

To avoid division by zero we have to require $x \notin \{k/n: k = 0, \dots, n-1\}$. By the assumption made this requirement is met by the x_ℓ and we conclude

$$p(x_\ell) = \frac{1}{n} \sum_{j=0}^{n-1} p(j/n) \frac{\exp(2\pi i n x_\ell) - 1}{\exp(2\pi i (\frac{k}{n} - x_\ell)) - 1},$$

from which we infer by the elementary operations of linear algebra

$$\zeta_\ell = \frac{1}{n} (\exp(2\pi i n x_\ell) - 1), \quad G(x, y) := \frac{1}{\exp(2\pi i (y - x)) - 1}, \quad x, y \in [0, 1[, \quad x \neq y.$$

HINT 1 for (2-5.f): Consider G 1-periodically extended to $\{(x, y) \in \mathbb{R}^2 : x - y \notin \mathbb{Z}\}$.

SOLUTION of (2-5.f):

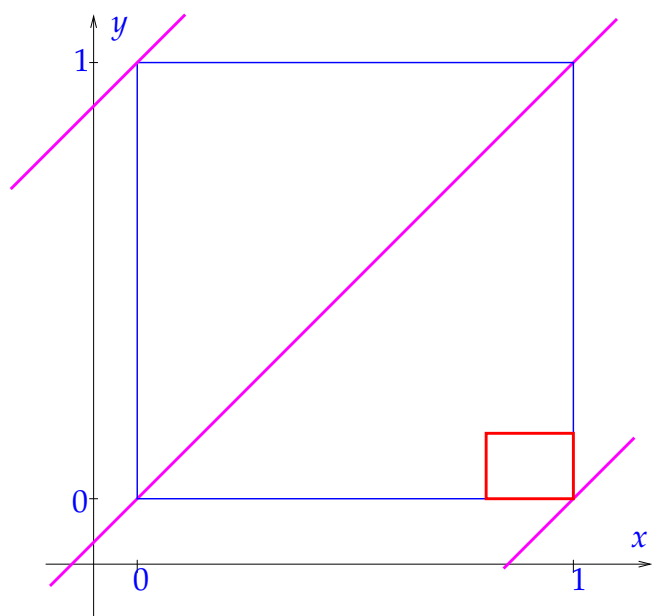
For $z \approx 0$ we have $\exp(z) - 1 \approx z$. Hence close to the “diagonal” $\{(x, y) \in \mathbb{R}^2 : x = y\}$ we have

$$G(x, y) \approx \frac{1}{2\pi i(x - y)}, \quad x \neq y.$$

It seems that G is just another **asymptotically smooth** kernel function [Lecture \rightarrow ??], a class, for which we found a suitable geometric admissibility condition [Lecture \rightarrow ??] in [Lecture \rightarrow ??]. Two clusters are regarded as admissible if the rectangle formed by their bounding boxes has a distance from the “line of singularity” proportional to its size, see [Lecture \rightarrow ??]. Uniform exponential convergence of Chebychev interpolation on far-field block was then observed in [Lecture \rightarrow ??].



Due to the periodicity of the complex exponential function the current kernel function G has more lines of singularity apart from the “main diagonal”.



More precisely, these lines of singularity form the set

$$S := \{(x, y) \in \mathbb{R}^2 : x - y \in \mathbb{Z}\}.$$

◁ So except for the “main diagonal”, two more lines in S will impact the geometric admissibility condition even if all products of bounding boxes are within the unit square $[0, 1]^2$. All relevant lines of singularity are marked with — in the figure.

◻ $\hat{=}$ absolutely inadmissible cluster pair



In the admissibility condition [Lecture \rightarrow ??] replace the distance **dist** of one-dimensional bounding boxes with their **1-periodic distance** dist_p .

$$\text{dist}_p(X, Y) := \min\{\text{dist}(X; Y), \text{dist}(X + 1; Y), \text{dist}(X - 1; Y)\}, \quad X, Y \subset [0, 1]. \quad (2.5.12)$$

So the final admissibility condition for two index sets $I \subset \{0, \dots, m - 1\}$ and $J \subset \{0, \dots, n - 1\}$ will be

$$\text{adm}(I, J) := \left\{ \frac{\max\{\text{diam}(\text{box}_x(I)), \text{diam}(\text{box}_y(J))\}}{2 \text{dist}_p(\text{box}_x(I), \text{box}_y(J))} \leq \eta_0 \right\}, \quad \eta_0 > 0. \quad (2.5.13)$$

The notations are the same as in [Lecture \rightarrow ??].

SOLUTION of (2-5.g):

We can start from the standard implementation of the `adm()` member function from [Lecture → ??]. All we need to do is to replace the regular distance function with dist_p as defined in (2.5.12),

SOLUTION of (2-5.h):

The requirement concerning the asymptotic computational effort severely constrains our options. We employ *local linearization*: we define the set

$$\mathcal{N} := \left\{ x \in [0, 1] : |x - k/n| \leq \sqrt{\tau}/2n \text{ and } |p'(k/n)(x - k/n)| \leq \tau |p(k/n)| + \tau^2 \text{ for some } k \in \{0, \dots, n-1\} \right\}.$$

All x_ℓ s belonging to \mathcal{N} will then be flagged as “replacable” with the nearest value k/n .

The numbers $p(k/n), p'(k/n)$ can both be computed with asymptotic computational cost $O(n \log n)$ for $n \rightarrow \infty$ by calling `evalTrigPolyEquid()` implemented in Sub-problem (2-5.b), because

$$p(x) = \sum_{j=0}^{n-1} \gamma_j \exp(-2\pi i j x) \quad \blacktriangleright \quad p'(x) = -2\pi i \sum_{j=1}^{n-1} j \gamma_j \exp(-2\pi i j x).$$

Thus, with a total effort of $O(n \log n)$ for $n \rightarrow \infty$, for every $k = 0, \dots, n-1$ we can compute that interval of \mathcal{N} enclosing k/n . Then we run through the x_ℓ s, find the nearest site k/n and determine whether x_ℓ lies inside the associated \mathcal{N} -interval.

C++ code 2.5.14: Checking for evaluation points causing problems to local low-rank compression. →GITLAB

```
2 std::vector<unsigned int> checkX(const Eigen::VectorXcd &gamma,
3                               const Eigen::VectorXd &x, double tau) {
4     //
5     // TODO
6     //
7 }
```

SOLUTION of (2-5.i):

The implementation is based on (2.5.11) and local low-rank compression of the kernel matrix $\mathbf{M} := [G(x_\ell, k/n)]_{\ell=0, \dots, m-1; k=0, \dots, n-1}$, using cluster trees. We rely on the data structures developed in Problem 2-4. It comprises the following steps.

- (i) Call `evalTrigPolyEquid()` to compute $\vec{\phi} := [p(k/n)]_{k=0}^{n-1}$ with an asymptotic computational effort of $n \log n$ for $n \rightarrow \infty$.
- (ii) Invoke `checkX` from Sub-problem (2-5.h) with a tolerance of $\tau = 10 \text{ EPS}$, `EPS` the machine precision, to determine those x_ℓ s, for which $p(x_\ell)$ can safely be replaced with a component of $\vec{\phi}$. Remove those x_ℓ s from the set of evaluation points.
- (iii) Compute the vector $\vec{\zeta} := [n^{-1}(\exp(2\pi i x_\ell))]_{\ell=0}^{m-1}$.
- (iv) Build binary cluster tree objects `equitree` and `xtree` of type `HMAT::ClusterTree<HMAT::InterpNode<1>, KERNEL>` for the point sets $\{k/n : k = 0, \dots, n-1\}$ and $\{w_\ell : \ell = 0, \dots, m-1\}$.
- (v) Based on the two cluster trees, a dedicated `struct` representing the kernel function `G` construct an object `llrcompmat` of type

```
template <typename KERNEL>
using BiDirChebPartMat1D_ETP = HMAT::BlockPartitionPeriodic<
HMAT::InterpNode<1>, HMAT::BiDirChebInterpBlock<HMAT::CtNode<1>,
    KERNEL>,
HMAT::NearFieldBlock<HMAT::CtNode<1>, KERNEL>>;
```

Use the admissibility parameter $\eta_0 = 0.5$.

- (vi) Invoke the function `mvLLRPartMat()` implemented in Sub-problem (2-4.i) to compute $\widetilde{\mathbf{M}}\vec{\phi}$, where $\widetilde{\mathbf{M}}$ is a the local low-rank approximation of \mathbf{M} provided by `llrcompmat`.
- (vii) Scale the result with the components of $\vec{\zeta}$ and return the result vector (after taking into account removed evaluation points).

C++ code 2.5.15: Local low-rank compression for fast evaluation of a trigonometric polynomial → [GITLAB](#)

```
2 Eigen::VectorXcd evalTrigPolyApprox(const Eigen::VectorXcd &gamma,
3                                     const Eigen::VectorXd &x, unsigned int q) {
4     //
5     // TODO
6     //
7 }
```

HINT 1 for (2-6.a): Use suitable addition theorems for the trigonometric functions.

SOLUTION of (2-6.a):

Using the addition formula for \sin we get

$$\sin(\xi_i - \xi_j) = \sin(\xi_i) \cos(\xi_j) - \cos(\xi_i) \sin(\xi_j) \quad , \quad i \in \{1, \dots, n\} .$$

This means that the matrix \mathbf{K}_1 has the representation

$$\mathbf{K}_1 = \begin{bmatrix} \sin(\xi_1) \\ \vdots \\ \sin(\xi_i) \\ \vdots \\ \sin(\xi_n) \end{bmatrix} \begin{bmatrix} \cos(\xi_1) & \dots & \cos(\xi_j) & \dots & \cos(\xi_n) \end{bmatrix} - \begin{bmatrix} \cos(\xi_1) \\ \vdots \\ \cos(\xi_i) \\ \vdots \\ \cos(\xi_n) \end{bmatrix} \begin{bmatrix} \sin(\xi_1) & \dots & \sin(\xi_j) & \dots & \sin(\xi_n) \end{bmatrix} . \quad (2.6.2)$$

The rank of \mathbf{K}_1 , which is the dimension of the row or columns space, is clearly 2, because it is the sum of two rank-1 matrices (tensor products of vectors).

The same considerations carry over to \mathbf{K}_2 , where we resort to the identity

$$\cos(\xi_i - \xi_j - \frac{1}{2n}) = \cos(\xi_i) \cos(\xi_j + \frac{1}{2n}) + \sin(\xi_i) \sin(\xi_j + \frac{1}{2n}) \quad , \quad i \in \{1, \dots, n\} . \quad (2.6.3)$$

By the same very same arguments as above, the rank of \mathbf{K}_2 turns out to be 2.

SOLUTION of (2-6.b):

The algorithm is explained in [Lecture → ??], see [Lecture → ??].

C++-code 2.6.4: Solution of (2-6.b)

```

2  std::pair<Eigen::MatrixXd, Eigen::MatrixXd> low_rank_merge(
3      const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
4      const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2) {
5      assert(A1.cols() == B1.cols() && A2.cols() == B2.cols() &&
6          A1.cols() == A2.cols() && "All no.s of cols should be equal to q");
7
8      size_t m = B1.rows();
9      size_t n = B1.cols();
10     // Find low-rank factors of B1 and B2 using QR decomposition as in
11     // (2.4.2.25) ??
12     Eigen::HouseholderQR<Eigen::MatrixXd> QR1 = B1.householderQr();
13     Eigen::HouseholderQR<Eigen::MatrixXd> QR2 = B2.householderQr();
14
15     // Build the thin matrix R
16     Eigen::MatrixXd R1 = Eigen::MatrixXd::Identity(std::min(m, n), m) *
17         QR1.matrixQR().triangularView<Eigen::Upper>();
18     Eigen::MatrixXd R2 = Eigen::MatrixXd::Identity(std::min(m, n), m) *
19         QR2.matrixQR().triangularView<Eigen::Upper>();
20
21     // About QR decomposition with Eigen:
22     // If  $B_1: m \times n$ , then  $Q_1: m \times m$  and  $R_1: m \times n$ .
23     // If  $m > n$ , however, the extra columns of  $Q_1$  and extra rows of  $R_1$ 
24     // are not needed. Matlab returns this "economy-size" format calling
25     // "qr(A, 0)", which does not compute these extra entries. With the
26     // code above,
27     // Eigen is smart enough not to compute the discarded vectors.
28
29     // Build  $\hat{Z}$  from  $A_i$  and  $R_i$ 
30     Eigen::MatrixXd Z(A1.rows(), R1.rows() + R2.rows());
31     Z << A1 * R1.transpose(), A2 * R2.transpose();
32
33     // Compute SVD of  $\hat{Z}$  as in (2.4.2.25) ??
34     Eigen::JacobiSVD<Eigen::MatrixXd> SVD(
35         Z, Eigen::ComputeThinU | Eigen::ComputeThinV);
36     Eigen::VectorXd s = SVD.singularValues();
37
38     // Only consider first q singular values
39     Eigen::MatrixXd S;
40     S.setZero(A1.cols(), A1.cols());
41     S.diagonal() = s.head(A1.cols());
42
43     // Only consider first q columns of U and V
44     Eigen::MatrixXd U = SVD.matrixU().leftCols(A1.cols());
45     Eigen::MatrixXd V = SVD.matrixV().leftCols(A1.cols());
46
47     // Split V to be compatible with Q1 and Q2 in compressed format
48     Eigen::MatrixXd V1 =
49         Eigen::MatrixXd::Identity(m, std::min(m, n)) * V.topRows(std::min(m, n));
50     Eigen::MatrixXd V2 = Eigen::MatrixXd::Identity(m, std::min(m, n)) *
51         V.bottomRows(std::min(m, n));

```

```

51 // About SVD decomposition with Eigen:
52 // With Eigen::JacobiSVD you can ask for thin U or V to be computed.
53 // In case of a rectangular  $m \times n$  matrix,
54 // with  $j$  the smaller value among  $m$  and  $n$ ,
55 // there can only be at most  $j$  singular values.
56 // The remaining columns of U and V do not correspond
57 // to actual singular vectors and are not computed in thin format.
58
59 // Compute  $\tilde{A}$  as in (??)
60 Eigen::MatrixXd Atilde = U * S;
61
62 // Compute  $\tilde{B}$  while avoiding recovering  $Q$  as a dense matrix
63 Eigen::MatrixXd Btilde1 = QR1.householderQ() * V1;
64 Eigen::MatrixXd Btilde2 = QR2.householderQ() * V2;
65 Eigen::MatrixXd Btilde(Btilde1.rows() + Btilde2.rows(), Btilde1.cols());
66 Btilde << Btilde1, Btilde2;
67
68 // Return factors of  $\tilde{Z}$ 
69 return {Atilde, Btilde};
70 }

```

Get it on  GitLab (lowrankmerge.cpp).

HINT 1 for (2-6.c): The factors $\mathbf{A}_1, \mathbf{B}_1 \in \mathbb{R}^{n,2}$ and $\mathbf{A}_2, \mathbf{B}_2 \in \mathbb{R}^{n,2}$ for \mathbf{K}_1 and \mathbf{K}_2 can be deduced from the representations (2.6.2) and (2.6.3) by the formula

$$\mathbf{v}_1 \mathbf{u}_1^\top + \mathbf{v}_2 \mathbf{u}_2^\top = [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} \mathbf{u}_1^\top \\ \mathbf{u}_2^\top \end{bmatrix},$$

which is a special case of [Lecture \rightarrow ??].

SOLUTION of (2-6.c):

C++-code 2.6.5: Solution of (2-6.c)

```

2  std::pair<double, double> test_low_rank_merge(size_t n) {
3      double nf = static_cast<double>(n); // convert data type
4      // Build  $K_1$ ,  $K_2$  and  $Z$  defined in Sub-problem (2-6.a)
5      Eigen::MatrixXd X1(n, n), X2(n, n);
6      for (int i = 0.; i < n; ++i) {
7          for (int j = 0.; j < n; ++j) {
8              X1(i, j) = std::sin((i - j) / nf);
9              X2(i, j) = std::cos((i - j - 0.5) / nf);
10         }
11     }
12     Eigen::MatrixXd Z(n, 2 * n);
13     Z << X1, X2;
14
15     // Build  $A_1$ ,  $B_1$ ,  $A_2$ ,  $B_2$  using Sub-problem (2-6.a)
16     Eigen::MatrixXd A1(n, 2), B1(n, 2), A2(n, 2), B2(n, 2);
17     for (int i = 0.; i < n; ++i) {
18         A1(i, 0) = std::sin(i / nf);
19         A1(i, 1) = std::cos(i / nf);
20         B1(i, 0) = std::cos(i / nf);
21         B1(i, 1) = -std::sin(i / nf);
22         A2(i, 0) = std::cos(i / nf);
23         A2(i, 1) = std::sin(i / nf);
24         B2(i, 0) = std::cos((i + 0.5) / nf);
25         B2(i, 1) = std::sin((i + 0.5) / nf);
26     }
27
28     // Call the function implemented in ??prb:lrn:subprb:2>
29     std::pair<Eigen::MatrixXd, Eigen::MatrixXd> AB =
30         low_rank_merge(A1, B1, A2, B2);
31
32     // Compute low-rank approximation error
33     Eigen::MatrixXd Ztilde = AB.first * AB.second.transpose();
34     Eigen::MatrixXd diff = Z - Ztilde;
35
36     double err_Frob = diff.norm() / n; // scaled Frobenius norm
37     double err_max = diff.cwiseAbs().maxCoeff(); // maximum norm
38
39     // Return scaled Frobenius norm and maximum norm of approximation
40     // error
41     return {err_Frob, err_max};
42 }

```


Get it on  GitLab ([lowrankmerge.cpp](#)).

Table of low-rank approximation errors:

n = 8	Frobenius norm = 9.532e-16	Max norm = 1.332e-15
n = 16	Frobenius norm = 4.263e-16	Max norm = 6.661e-16
n = 32	Frobenius norm = 6.501e-16	Max norm = 1.332e-15
n = 64	Frobenius norm = 3.152e-16	Max norm = 1.332e-15
n = 128	Frobenius norm = 7.333e-16	Max norm = 1.443e-15
n = 256	Frobenius norm = 4.503e-16	Max norm = 2.887e-15
n = 512	Frobenius norm = 6.534e-16	Max norm = 2.502e-15
n = 1024	Frobenius norm = 9.582e-16	Max norm = 4.496e-15
n = 2048	Frobenius norm = 1.078e-15	Max norm = 2.909e-14
n = 4096	Frobenius norm = 1.982e-15	Max norm = 2.004e-14

Here low-rank compression just incurs errors of about the same order of magnitude as machine precision. The reason is that $[\mathbf{K}_1 \ \mathbf{K}_2]$ still has rank 2, because \mathbf{A}_1 and \mathbf{A}_2 have the same column space!

SOLUTION of (2-6.d):

C++-code 2.6.6: Solution of (2-6.d)

```

2  std::pair<Eigen::MatrixXd, Eigen::MatrixXd> adap_rank_merge(
3      const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
4      const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2, double rtol,
5      double atol) {
6      assert(A1.cols() == B1.cols() && A2.cols() == B2.cols() &&
7          A1.cols() == A2.cols() && "All no.s of cols should be equal to q");
8
9      size_t m = B1.rows();
10     size_t n = B1.cols();
11     // Find low-rank factors of B1 and B2 using QR decomposition as in (2.4.2.25)
12     \cref{par:mergetrunc}
13     Eigen::HouseholderQR<Eigen::MatrixXd> QR1 = B1.householderQr();
14     Eigen::HouseholderQR<Eigen::MatrixXd> QR2 = B2.householderQr();
15
16     // Build the thin matrix R
17     Eigen::MatrixXd R1 = Eigen::MatrixXd::Identity(std::min(m, n), m) *
18         QR1.matrixQR().triangularView<Eigen::Upper>();
19     Eigen::MatrixXd R2 = Eigen::MatrixXd::Identity(std::min(m, n), m) *
20         QR2.matrixQR().triangularView<Eigen::Upper>();
21
22     // Construct  $\hat{Z}$  from  $A_i$  and  $R_i$ 
23     Eigen::MatrixXd Z(A1.rows(), R1.rows() + R2.rows());
24     Z << A1 * R1.transpose(), A2 * R2.transpose();
25
26     // Compute SVD of  $\hat{Z}$ 
27     Eigen::JacobiSVD<Eigen::MatrixXd> SVD(
28         Z, Eigen::ComputeThinU | Eigen::ComputeThinV);
29     Eigen::VectorXd s = SVD.singularValues();
30
31     // Find singular values larger than atol and rtol as in \eqref{eq:adaptrunc}
32     unsigned p = s.size();
33     for (unsigned q = 1; q < s.size(); ++q) {
34         //  $q \in \{1, \dots, p-1\} : \sigma_q \leq \text{rtol} \cdot \sigma_0$ 
35         if ((s(q) <= s(0) * rtol) or (s(q) <= atol)) {
36             p = q;
37             break;
38         }
39     }
40
41     // Only consider the largest p singular values
42     Eigen::MatrixXd S;
43     S.setZero(p, p);
44     S.diagonal() = s.head(p);
45
46     // Only consider the first p columns of U and V
47     Eigen::MatrixXd U = SVD.matrixU().leftCols(p);
48     Eigen::MatrixXd V = SVD.matrixV().leftCols(p);
49
50     // Split V to be compatible with Q1 and Q2 in compressed format
51     Eigen::MatrixXd V1 =
52         Eigen::MatrixXd::Identity(m, std::min(m, n)) * V.topRows(std::min(m, n));
53     Eigen::MatrixXd V2 = Eigen::MatrixXd::Identity(m, std::min(m, n)) *

```

```

53         V.bottomRows(std::min(m, n));
54
55     // Compute  $\tilde{A}$  as in \eqref{eq:lrfac1}
56     Eigen::MatrixXd Atilde = U * S;
57
58     // Compute  $\tilde{B}$  while avoiding recovering  $Q$  as a dense matrix
59     Eigen::MatrixXd Btilde1 = QR1.householderQ() * V1;
60     Eigen::MatrixXd Btilde2 = QR2.householderQ() * V2;
61     Eigen::MatrixXd Btilde(Btilde1.rows() + Btilde2.rows(), Btilde1.cols());
62     Btilde << Btilde1, Btilde2;
63
64     // Return factors of  $\tilde{Z}$ 
65     return {Atilde, Btilde};
66 }

```

Get it on  GitLab (lowrankmerge.cpp).

SOLUTION of (2-6.e):

C++-code 2.6.7: Solution of Sub-problem (2-6.e)

```
2  std::pair<double, size_t> test_adap_rank_merge(size_t n, double rtol) {
3  // Make sure that trigonometric functions receive float arguments
4  // "Hidden integer arithmetic" is a trap of C/C++
5  const double nf = static_cast<double>(n);
6  // Build  $K_1$ ,  $K_2$  and  $Z$  defined in ??
7  Eigen::MatrixXd X1(n, n), X2(n, n);
8  for (int i = 0.; i < n; ++i) {
9      for (int j = 0.; j < n; ++j) {
10         X1(i, j) = std::sin((i - j) / nf);
11         X2(i, j) = std::cos((i - j - 0.5) / nf);
12     }
13 }
14 Eigen::MatrixXd Z(n, 2 * n);
15 Z << X1, X2;
16
17 // Build  $A_1$ ,  $B_1$ ,  $A_2$ ,  $B_2$  using Sub-problem (2-6.a)
18 Eigen::MatrixXd A1(n, 2), B1(n, 2), A2(n, 2), B2(n, 2);
19 for (int i = 0.; i < n; ++i) {
20     A1(i, 0) = std::sin(i / nf);
21     A1(i, 1) = std::cos(i / nf);
22     B1(i, 0) = std::cos(i / nf);
23     B1(i, 1) = -std::sin(i / nf);
24     A2(i, 0) = std::cos(i / nf);
25     A2(i, 1) = std::sin(i / nf);
26     B2(i, 0) = std::cos((i + 0.5) / nf);
27     B2(i, 1) = std::sin((i + 0.5) / nf);
28 }
29
30 // Call the function from Sub-problem (2-6.d)
31 std::pair<Eigen::MatrixXd, Eigen::MatrixXd> AB =
32     adap_rank_merge(A1, B1, A2, B2, rtol, __DBL_MIN__);
33
34 // Compute low-rank approximation error
35 Eigen::MatrixXd Ztilde = AB.first * AB.second.transpose();
36 Eigen::MatrixXd diff = Z - Ztilde;
37
38 double err_Frob = diff.norm() / n; // scaled Frobenius norm
39
40 // Return scaled Frobenius norm of approximation error and
41 // the rank required to achieve the relative tolerance rtol
42 return {err_Frob, AB.first.cols()};
43 }
```

Get it on  GitLab (lowrankmerge.cpp).

Fixed $rtol = 10^{-4}$:

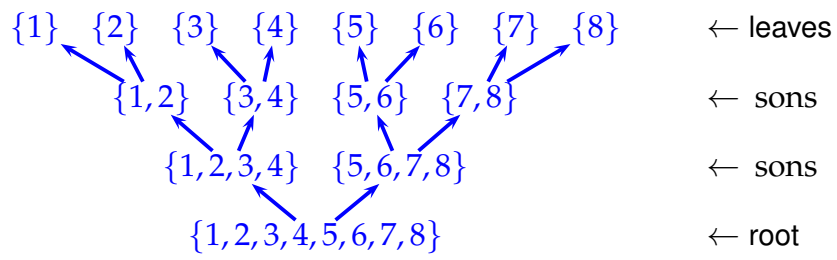
```
n = 8      Frobenius norm = 9.532e-16 Rank = 2
n = 16     Frobenius norm = 4.263e-16 Rank = 2
n = 32     Frobenius norm = 6.501e-16 Rank = 2
n = 64     Frobenius norm = 3.152e-16 Rank = 2
n = 128    Frobenius norm = 7.333e-16 Rank = 2
n = 256    Frobenius norm = 4.503e-16 Rank = 2
n = 512    Frobenius norm = 6.534e-16 Rank = 2
n = 1024   Frobenius norm = 9.582e-16 Rank = 2
n = 2048   Frobenius norm = 1.078e-15 Rank = 2
n = 4096   Frobenius norm = 1.982e-15 Rank = 2
```

Fixed $n = 500$:

```
rtol = 1.000e-01  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-02  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-03  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-04  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-05  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-06  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-07  Frobenius norm = 1.188e-15 Rank = 2
rtol = 1.000e-08  Frobenius norm = 1.188e-15 Rank = 2
```

Of course, since $\text{rank}([\mathbf{K}_1 \ \mathbf{K}_2]) = 2$ we do not expect the adaptive approach to make much of a difference. For any reasonably small tolerance truncation at rank = 2 will satisfy the accuracy requirements.

SOLUTION of (2-7.a):



HINT 1 for (2-7.j): As usual, let us consider the partition of a hierarchical matrix $\mathbf{H} \in \mathcal{S}_p$:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix}$$

In the case of an LU-decomposition, both factors \mathbf{L} and \mathbf{U} are also \mathcal{H} -matrices:

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ 0 & \mathbf{U}_{22} \end{bmatrix}$$

SOLUTION of (2-7.j):

As in [Lecture → ??] we start from the matrix partitions:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} \mathbf{L}_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ 0 & \mathbf{U}_{22} \end{bmatrix}. \quad (2.7.12)$$

All the matrices are \mathcal{H} -matrices $\in \mathcal{S}_p$. In particular this means that all non-zero off-diagonal blocks are rank- q matrices available through their rank- q factors.

We recall the equations from [Lecture → ??], which define the non-zero blocks of \mathbf{L} and \mathbf{U} :

- ① Find $\mathbf{L}_{11}, \mathbf{U}_{11} \in \mathbb{R}^{k,k}$: $\mathbf{L}_{11} \cdot \mathbf{U}_{11} = \mathbf{H}_{11}$ $\hat{=}$ LU-decomposition \triangleright recursion,
- ② Find $\mathbf{U}_{12} \in \mathbb{R}^{k,l}$: $\mathbf{L}_{11} \mathbf{U}_{12} = \mathbf{H}_{12}$ $\hat{=}$ forward elimination,
Find $\mathbf{L}_{21} \in \mathbb{R}^{l,k}$: $\mathbf{L}_{21} \mathbf{U}_{11} = \mathbf{H}_{21}$ $\hat{=}$ forward elimination,
- ③ Find $\mathbf{L}_{22}, \mathbf{U}_{22} \in \mathbb{R}^{l,l}$: $\mathbf{L}_{22} \cdot \mathbf{U}_{22} = \mathbf{H}_{22} - \mathbf{L}_{21} \mathbf{U}_{12}$ $\hat{=}$ LU-decomposition \triangleright recursion.

However, since $\mathbf{H} \in \mathcal{S}_p$, the step ② (forward eliminations) is simplified, because \mathbf{H}_{12} and \mathbf{H}_{21} are square rank- q matrices, which can be expressed as $\mathbf{A}_{12}^H (\mathbf{B}_{12}^H)^\top$ and $\mathbf{A}_{21}^H (\mathbf{B}_{21}^H)^\top$, respectively. We can readily apply the insight from [Lecture → ??].

Hence, if we factorize \mathbf{U}_{12} and \mathbf{L}_{21} into $\mathbf{A}_{12}^U (\mathbf{B}_{12}^U)^\top$ and $\mathbf{A}_{21}^L (\mathbf{B}_{21}^L)^\top$, those factors can be computed by forward eliminations:

- ② Find $\mathbf{U}_{12} \in \mathbb{R}^{k,l}$: $\mathbf{L}_{11} \mathbf{A}_{12}^U (\mathbf{B}_{12}^U)^\top = \mathbf{V} \mathbf{A}_{12}^H (\mathbf{B}_{12}^H)^\top$ $\hat{=}$ forward elimination,
Find $\mathbf{L}_{21} \in \mathbb{R}^{l,k}$: $\mathbf{A}_{21}^L (\mathbf{B}_{21}^L)^\top \mathbf{U}_{11} = \mathbf{A}_{21}^H (\mathbf{B}_{21}^H)^\top$ $\hat{=}$ forward elimination.

By imposing $\mathbf{B}_{12}^U \equiv \mathbf{B}_{12}^H$ and $\mathbf{A}_{21}^L \equiv \mathbf{A}_{21}^H$, we only need to find two matrices $\in \mathbb{R}^{2^{p-1}, q}$. This will prove useful for the solution of the next Sub-problem (2-7.k).

HINT 1 for (2-7.k): Look up [Lecture → ??].



HINT 2 for (2-7.k): Similarly to `ssmlrm()` (see Sub-problem (2-7.i)), you will have to rely on a function from the lecture, namely `hmat_forw_elim()` from [Lecture → ??].

HINT 3 for (2-7.k): You can use again the fact that, if you denote the columns of \mathbf{A} and \mathbf{B} as \mathbf{a}_i and \mathbf{b}_i , then $\mathbf{AB}^\top = \sum_{i=1}^q \mathbf{a}_i \mathbf{b}_i^\top$. ┘

SOLUTION of (2-7.k):

Given the hint, we have that

$$\begin{aligned} \mathbf{L}(\mathbf{A}\mathbf{B}^\top) &= \mathbf{U}\mathbf{V}^\top \implies \\ \mathbf{L} \sum_{i=1}^q \mathbf{a}_i \mathbf{b}_i^\top &= \sum_{i=1}^q \mathbf{u}_i \mathbf{v}_i^\top \implies \\ \sum_{i=1}^q (\mathbf{L}\mathbf{a}_i) \mathbf{b}_i^\top &= \sum_{i=1}^q \mathbf{u}_i \mathbf{v}_i^\top \end{aligned}$$

If we impose that $\mathbf{b}_i \equiv \mathbf{v}$, in a similar way to the far-field case of `hmat_triag_solve()` from [Lecture → ??], and we consider a componentwise equality, we need to solve q problems of the form [Lecture → ??], i.e. solving triangular linear systems with an \mathcal{H} -coefficient matrix. This can be done by means of function `hmat_forw_elim()` from [Lecture → ??].

Pseudocode 2.7.13: Solving a triangular linear system with $\mathbf{H} \in \mathcal{S}_p$ and $\mathbf{U}\mathbf{V}^\top$, $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n,q}$

```
1  [A, B ∈ ℝn,q] ← ssmat_triag_solve( $\mathcal{H}$ -matrix  $\mathbf{L} \in \mathcal{S}_p$ , matrix  $\mathbf{U} \in \mathbb{R}^{n,q}$ ,  
    $\mathbf{V} \in \mathbb{R}^{n,q}$ )  
2  {  
3    [A, B] := output matrices  
4    for ( $i = 1 \rightarrow q$ ) {  
5       $\mathbf{a}_i = \text{hmat\_forw\_elim}(\mathbf{L}, \mathbf{u}_i)$ ; // [Lecture → ??]  
6    }  
7     $\mathbf{B} = \mathbf{V}$ ;  
8    return [A, B];  
9  }
```

Note that Code 2.7.15 can be used to compute one of the forward eliminations in Sub-problem (2-7.j).

SOLUTION of (2-7.l):

Similarly to Sub-problem (2-7.g), the solution is straightforward: considering the `for` loop and the call to `hmat_forw_elim()` inside, whose cost has been computed in Sub-problem (2-7.i), we obtain that

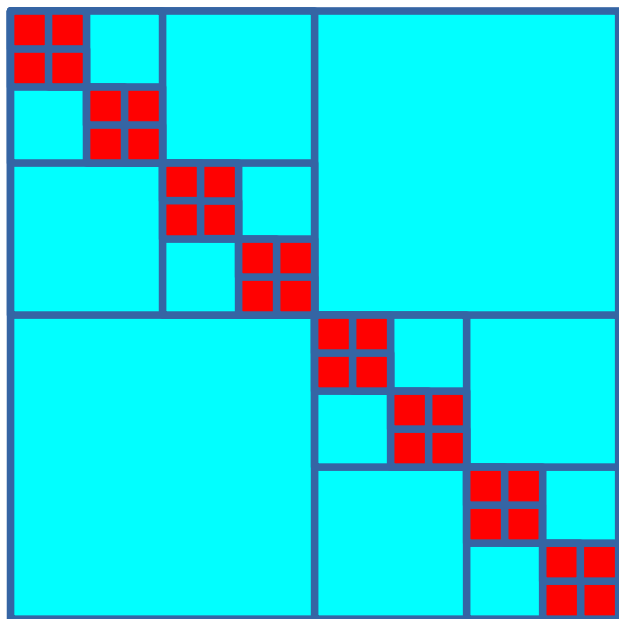
$$W_{mr}(p) = 2^{p-1}q (2 + 4p + p^2 + 2pq + 2p^2q)$$

SOLUTION of (2-7.b):

$\text{adm}(v, w)$ is clearly a mapping $\mathcal{T}_I \times \mathcal{T}_J \rightarrow \{\text{true}, \text{false}\}$, so the first condition is fulfilled.

Let us now consider $\mathcal{I}(v)$ and $\mathcal{I}(w)$ disjoint sets of indices, necessary and sufficient condition for which the pair $v, w \in \mathcal{T}_I$ be admissible. Then any subset $\mathcal{I}'(v) \in \mathcal{I}(v)$ and $\mathcal{I}(w)$ are also disjoint, and the same holds for any subset $\mathcal{I}'(w) \in \mathcal{I}(w)$ and $\mathcal{I}(v)$. This means that the sons of admissible pairs inherit the admissibility, and the second condition is also fulfilled.

SOLUTION of (2-7.c):



The light blue blocks belong to the far-field, the red blocks to the near-field. Note that the off-diagonal red blocks are part of the near-field vector because they contain only one index in each node of the corresponding pair.

See also [Lecture → ??].

SOLUTION of (2-7.d):

The sparsity measure for a pair of binary cluster trees with $n = 2^p$ is always 2, regardless of p . A visual inspection of Fig. 5 gives the answer. Only the leaves can maximize the number of occurrences in pairs of the block partition, and each leaf i appears in at most two pairs: one with itself and another with a neighboring leaf. The index j of the neighboring leaf is $j = i - 1$ or $j = i + 1$ depending whether i is even or odd.

HINT 1 for (2-7.e): To derive a recursion formula for $W_{mv}(p)$, you need to think about how the hierarchical matrix $\mathbf{H} \in \mathcal{S}_p$ is enlarged for every p with respect to $p - 1$. Specifically, consider the following partition:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix} \quad (2.7.3)$$

Correspondingly, the vector $\vec{\mu} \in \mathbb{R}^{2^p}$ that needs to be multiplied can be split as:

$$\vec{\mu} = \begin{bmatrix} \vec{\mu}_1 \\ \vec{\mu}_2 \end{bmatrix}$$

\mathbf{H}_{11} and \mathbf{H}_{22} are hierarchical matrices in $\mathbb{R}^{2^{p-1}, 2^{p-1}}$, for which the number of operations is $W_{mv}(p - 1)$. The off-diagonal matrices \mathbf{H}_{12} and \mathbf{H}_{21} can be treated as admissible matrices for the far-field vector (they share no common indices).

Hence, assume that \mathbf{H}_{12} and \mathbf{H}_{21} are rank- q matrices which can be written as \mathbf{AB}^\top , with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{2^p, q}$. This factorization can stem from a bi- or uni-directional interpolation.

What is the complexity of the matrix-vector multiplications of \mathbf{H}_{12} and \mathbf{H}_{21} with $\vec{\mu}_1, \vec{\mu}_2 \in \mathbb{R}^{2^{p-1}}$, given the rules of Rem. 2.7.4? ┘

SOLUTION of (2-7.e):

❶ An approximation for the cost of the matrix-vector product $W_{mv}(p)$ can be derived from the upper bound of [Lecture → ??] (derived from the memory storage [Lecture → ??]) or from the other bound, in big O notation, of [Lecture → ??]. The upper bound of [Lecture → ??] would makes use of the sparsity measure obtained in the solution of Sub-problem (2-7.d). The assumption is a balanced cluster tree, which our binary cluster tree respects.

❷ However, we want a sharper bound. Let us try to find a more accurate estimate based on the hint.

Given the rank- q off-diagonal matrix $\mathbf{H}_{12} = \mathbf{A}_{12}\mathbf{B}_{12}^\top$ from (2.7.5), the matrix-vector product with $\vec{\mu}_1$ has complexity $q2^{p-1} + 2^{p-1}q = 2^p q$. This must be repeated twice for \mathbf{H}_{21} . We also have to consider the sum of the two resulting sub-vectors to form the full product vector, each a SAXPY operation of cost 2^{p-1} . The total cost is therefore $2^p(2q + 1)$.

Considering the partition of the hint and the presence of two matrices \mathbf{H}_{11} and \mathbf{H}_{22} with cost $W_{mv}(p - 1)$, we finally get the following *recursion formula*:

$$W_{mv}(p) = 2W_{mv}(p - 1) + 2^p(2q + 1) . \quad (2.7.4)$$

We clearly have that $W_{mv}(0) = 1$. The following considerations give a closed formula (“recursion tree technique”)

- At the top level of the recursion the work is $2^p(2q + 1)$
- At the next lower level we need $2 \cdot 2^{p-1}(2q + 1) = 2^p(2q + 1)$ work units.
- One more level below it takes $4 \cdot 2^{p-2}(2q + 1) = 2^p(2q + 1)$ work units
- ...

So at any level (maybe except for the very lowest) we need the same amount of work $2^p(2q + 1)$. So we arrive at the work count

$$W_{mv}(p) = p2^p(2q + 1) + 2^p .$$

HINT 1 for (2-7.f): A rank- q matrix in factorized form can be written as the product \mathbf{AB}^\top , with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{2^p}$,

At the same time, if you denote the columns of \mathbf{A} and \mathbf{B} as \mathbf{a} and \mathbf{b} , we can write $\mathbf{AB}^\top = \sum_{i=1}^q \mathbf{a}_i \mathbf{b}_i^\top$. What

does this imply for the final form of the matrix product \mathbf{HAB}^\top ? ┘

SOLUTION of (2-7.f):

Given the hint, we have that

$$\mathbf{HAB}^\top = \mathbf{H} \sum_{i=1}^q \mathbf{a}_i \mathbf{b}_i^\top = \sum_{i=1}^q (\mathbf{H}\mathbf{a}_i) \mathbf{b}_i^\top$$

The matrix product between a hierarchical matrix and \mathbf{AB}^\top is therefore equivalent to q matrix-vector products, which have already been considered in function `hmv()` of [Lecture → ??]. These matrix-vector products will give the columns of the first factor of the output rank- q matrix, \mathbf{U} .

We can then impose \mathbf{V} , the second factor of the output matrix, $\equiv \mathbf{B}$.

Pseudocode 2.7.5: Matrix multiplication between $\mathbf{H} \in \mathcal{S}_p$ and \mathbf{AB}^\top , $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,q}$

```
1 [U, V ∈ ℝn,q] ← ssmlrm( $\mathcal{H}$ -matrix  $\mathbf{H} \in \mathcal{S}_p$ , Matrix  $\mathbf{A}$ , Matrix  $\mathbf{B}$ ) {
2   matrix  $\mathbf{U}, \mathbf{V} := \mathbf{O}$ ; // output matrices
3   for ( $i = 1 \rightarrow q$ ) {
4     vector  $\vec{\zeta} \in \mathbb{R}^n$ ;
5     hmv( $\mathbf{H}$ ,  $\vec{\zeta}$ ,  $\mathbf{a}_i$ ); // [Lecture → ??]
6      $\mathbf{u}_i = \vec{\zeta}$ ;  $\mathbf{v}_i = \mathbf{b}_i$ ;
7   }
8   return [ $\mathbf{U}, \mathbf{V}$ ];
9 }
```

SOLUTION of (2-7.g):

The solution is straightforward: considering the `for` loop and the operations involved in the pseudocode, we obtain that

$$W_{mr}(p) = q \left(p2^{p+1}q + (p + 1)2^p \right) = 2^{p+1}pq^2 + (p + 1)2^p q .$$

The terms within brackets in the first expression of $W_{mr}(p)$ come from the solution of Sub-problem (2-7.e).

HINT 1 for (2-7.h): Reuse of the function `ssmlrm()` from Sub-problem (2-7.f) is recommended. ┘

HINT 2 for (2-7.h): Study [Lecture → ??] again.

HINT 3 for (2-7.h): Start from the partition (2.7.5) of a semi-separable hierarchical matrix $\mathbf{H}^p \in \mathcal{S}_p$ mentioned in the hint of Sub-problem (2-7.e):

$$\mathbf{H}^p = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{H}_{12} \\ \mathbf{H}_{21} & \mathbf{H}_{22} \end{bmatrix}. \quad (2.7.5)$$

- The *diagonal blocks* $\mathbf{H}_{11}, \mathbf{H}_{22} \in \mathbb{R}^{2^{p-1}, 2^{p-1}}$ are semi-separable hierarchical matrices $\in \mathcal{S}_{p-1}$.
- The *off-diagonal blocks* $\mathbf{H}_{12}, \mathbf{H}_{21} \in \mathbb{R}^{2^{p-1}, 2^{p-1}}$ are *far-field blocks* are rank- q matrices and are given in factorized form, e.g. $\mathbf{H}_{12} = \mathbf{A}\mathbf{B}^\top$, with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{2^{p-1}, q}$.

┘

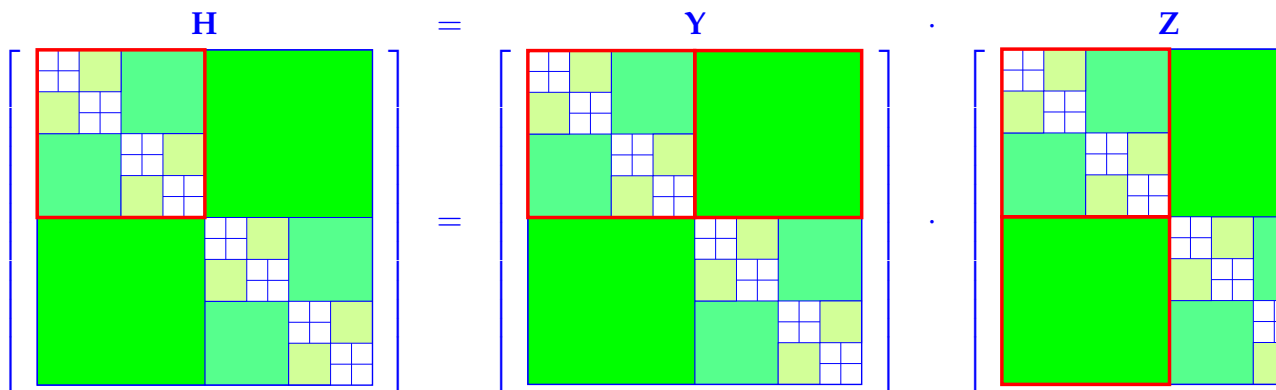
HINT 4 for (2-7.h): Study [Lecture → ??] and use [Lecture → ??].

HINT 5 for (2-7.h): It is important to take into account that all \mathcal{H} -matrices to be multiplied are based on the *same back tree*. This greatly simplifies implementation and rules out many of the cases that had to be discussed in [Lecture → ??].

SOLUTION of (2-7.h):

Based on the block partitioning (2.7.5) we can visualize how the blocks are filled during the matrix multiplication.

- Block \mathbf{H}_{11} :



The block $\mathbf{H}_{11} \in \mathcal{S}_{p-1}$ is updated by the product of two matrices $\in \mathcal{S}_{p-1}$ (\rightarrow *recursion!*) and the product of two rank- q matrices, which is itself a rank- q matrix.

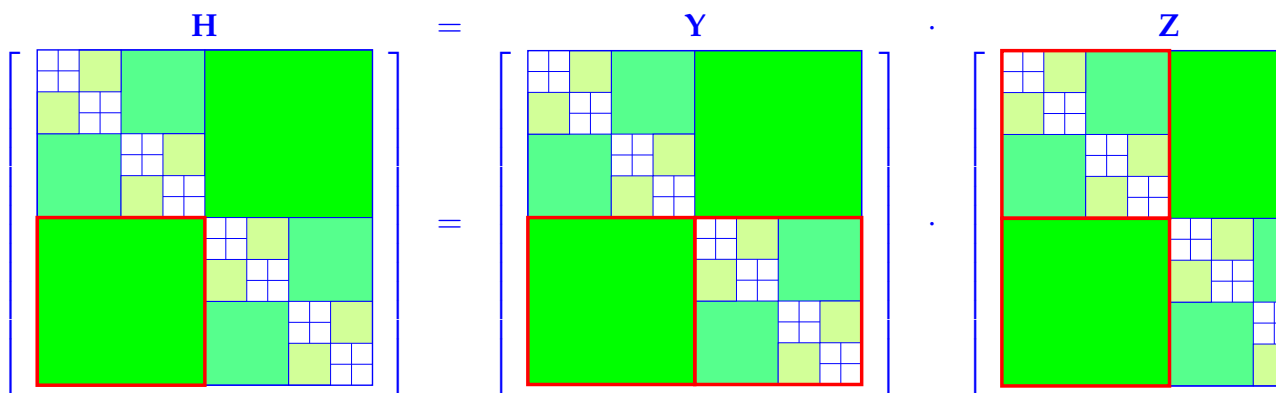
$$\mathbf{H}_{11} = \mathbf{Y}_{11} \cdot \mathbf{Z}_{11} + \mathbf{A}_{12}^Y (\mathbf{B}_{12}^Y)^\top \mathbf{A}_{21}^Z (\mathbf{B}_{21}^Z)^\top, \quad \begin{aligned} \mathbf{Y}_{12} &= \mathbf{A}_{12}^Y (\mathbf{B}_{12}^Y)^\top, \\ \mathbf{Z}_{21} &= \mathbf{A}_{21}^Z (\mathbf{B}_{21}^Z)^\top. \end{aligned} \quad (2.7.6)$$

with $\mathbf{A}_*^*, \mathbf{B}_*^* \in \mathbb{R}^{2^{p-1} \times q}$. The highlighted matrices are \mathcal{H} -matrices $\in \mathcal{S}_p$. To take into account the second summand, which can be recast into rank- q factorized form

$$\mathbf{A}_{12}^Y (\mathbf{B}_{12}^Y)^\top \mathbf{A}_{21}^Z (\mathbf{B}_{21}^Z)^\top = \mathbf{A}_{12}^Y \left(\mathbf{B}_{21}^Z (\mathbf{A}_{21}^Z)^\top \mathbf{B}_{12}^Y \right)^\top, \quad (2.7.7)$$

we can resort to `low_rank_update()` from [Lecture \rightarrow ??].

- Block \mathbf{H}_{21} :



The block \mathbf{H}_{21} is a rank- q block and is updated by adding

1. the product of a hierarchical matrix $\in \mathcal{S}_{p-1}$ and a rank- q matrix, which is itself a rank- q matrix, and
2. the product of a rank- q matrix and a hierarchical matrix $\in \mathcal{S}_{p-1}$, which is itself a rank- q matrix:

$$\mathbf{H}_{21} = \mathbf{A}_{21}^Y \left(\mathbf{Z}_{11}^\top \mathbf{B}_{21}^Y \right)^\top + \left(\mathbf{Y}_{22} \cdot \mathbf{A}_{12}^Z \right) \left(\mathbf{B}_{21}^Z \right)^\top, \quad (2.7.8)$$

$$\mathbf{H}_{12} = \mathbf{A}_{12}^Y \left(\mathbf{Z}_{22}^\top \mathbf{B}_{12}^Y \right)^\top + \left(\mathbf{Y}_{11} \cdot \mathbf{A}_{21}^Z \right) \left(\mathbf{B}_{12}^Z \right)^\top. \quad (2.7.9)$$

Thus we can use `ssmlrm()` from Sub-problem (2-7.f) followed by a call to `low_rank_update()` from [Lecture → ??].

Compared to [Lecture → ??] the discussion is greatly simplified by the following observation. If $\sigma := (v, w)$ is a non-leaf node of the block tree of the \mathcal{H} -matrix $\mathbf{H} \in \mathcal{S}_p$ with $\text{sons}(v) = \{s_1, s_2\}$ and $\text{sons}(w) = \{t_1, t_2\}$ then

$$\begin{aligned} \text{(i)} \quad & \mathbf{H}|_{s_1 \times t_1} \in \mathcal{C}_{s_{p-1}} \quad \text{and} \quad \mathbf{H}|_{s_2 \times t_2} \in \mathcal{S}_{p-1}, \\ \text{(ii)} \quad & \mathbf{H}|_{s_1 \times t_2}, \mathbf{H}|_{s_2 \times t_1} \text{ are rank-}q \text{ matrices.} \end{aligned} \quad (2.7.10)$$

Pseudocode 2.7.11: Matrix multiplication of $\mathbf{Y} \in \mathcal{S}_p$ and $\mathbf{Z} \in \mathcal{S}_p$

```

1 void ssmm(ref  $\mathcal{H}$ -matrix  $\mathbf{H} \in \mathcal{S}_p$ , const  $\mathcal{H}$ -matrix  $\mathbf{Y} \in \mathcal{S}_p$ , const  $\mathcal{H}$ 
2   -matrix  $\mathbf{Z} \in \mathcal{S}_p$ ) {
3    $\sigma := (v, w) := \text{root}(\mathcal{B}_H) = \text{root node of block tree for } \mathbf{H}$ 
4    $\tau := (v, u) := \text{root}(\mathcal{B}_Y) = \text{root node of block tree for } \mathbf{Y}$ 
5    $\kappa := (u, w) := \text{root}(\mathcal{B}_Z) = \text{root node of block tree for } \mathbf{Z}$ 
6   // Taken for granted: none of  $\sigma, \kappa, \tau$  is a leaf
7    $\{s_1, s_2\} := \text{sons}(\sigma)$ ;  $\{t_1, t_2\} := \text{sons}(\sigma)$ ;  $\{r_1, r_2\} := \text{sons}(\sigma)$ ;
8
9   if (sons( $s_1$ ) =  $\emptyset$ ) {
10    // Sub blocks of  $\mathbf{H}$ ,  $\mathbf{Y}$ , and  $\mathbf{Z}$  are all near field
11     $\mathbf{N}_{s_1 \times t_1}^H \leftarrow \mathbf{N}_{s_1 \times t_1}^H + \mathbf{N}_{s_1 \times r_1}^Y \mathbf{N}_{r_1 \times t_1}^Z + \mathbf{N}_{s_1 \times r_2}^Y \mathbf{N}_{r_2 \times t_1}^Z$ ;
12     $\mathbf{N}_{s_1 \times t_2}^H \leftarrow \mathbf{N}_{s_1 \times t_2}^H + \mathbf{N}_{s_1 \times r_1}^Y \mathbf{N}_{r_1 \times t_2}^Z + \mathbf{N}_{s_1 \times r_2}^Y \mathbf{N}_{r_2 \times t_2}^Z$ ;
13     $\mathbf{N}_{s_2 \times t_1}^H \leftarrow \mathbf{N}_{s_2 \times t_1}^H + \mathbf{N}_{s_2 \times r_1}^Y \mathbf{N}_{r_1 \times t_1}^Z + \mathbf{N}_{s_2 \times r_2}^Y \mathbf{N}_{r_2 \times t_1}^Z$ ;
14     $\mathbf{N}_{s_2 \times t_2}^H \leftarrow \mathbf{N}_{s_2 \times t_2}^H + \mathbf{N}_{s_2 \times r_1}^Y \mathbf{N}_{r_1 \times t_2}^Z + \mathbf{N}_{s_2 \times r_2}^Y \mathbf{N}_{r_2 \times t_2}^Z$ ;
15  }
16  else {
17    // Case of generic semi-separable  $\mathcal{H}$ -matrices;; exploit (2.7.12)
18    // First update diagonal sub-blocks
19    ssmm( $\mathbf{H}|_{s_1 \times t_1}$ ,  $\mathbf{Y}|_{s_1 \times r_1}$ ,  $\mathbf{Z}|_{r_1 \times t_1}$ );
20    low_rank_update( $\mathbf{H}|_{s_1 \times t_1}$ ,  $\mathbf{A}_{s_1 \times r_2}^Y$ ,  $\mathbf{B}_{r_2 \times t_1}^Z (\mathbf{A}_{r_2 \times t_1}^Z)^\top \mathbf{B}_{s_1 \times r_2}^Y$ ); // (2.7.8)
21    ssmm( $\mathbf{H}|_{s_2 \times t_2}$ ,  $\mathbf{Y}|_{s_2 \times r_2}$ ,  $\mathbf{Z}|_{r_2 \times t_2}$ );
22    low_rank_update( $\mathbf{H}|_{s_2 \times t_2}$ ,  $\mathbf{A}_{s_2 \times r_1}^Y$ ,  $\mathbf{B}_{r_1 \times t_2}^Z (\mathbf{A}_{r_1 \times t_2}^Z)^\top \mathbf{B}_{s_2 \times r_1}^Y$ );
23    // Update of rank- $q$  off-diagonal sub-blocks
24    Matrix TY = hmat_mult_dense( $\mathbf{Y}_{s_2 \times r_2}$ ,  $\mathbf{A}_{r_1 \times t_2}^Z$ );
25    Matrix TZ = hmat_transpose_mult_dense( $\mathbf{Z}_{r_1 \times t_1}$ ,  $\mathbf{B}_{s_2 \times r_1}^Y$ );
26    [ $\mathbf{A}_{s_2 \times t_1}^H$ ,  $\mathbf{B}_{s_2 \times t_1}^H$ ] = low_rank_sum( $\mathbf{A}_{s_2 \times t_1}^H$ ,  $\mathbf{B}_{s_2 \times t_1}^H$ ,  $\mathbf{A}_{s_2 \times r_1}^Y$ , TZ); // (2.7.10)
27    [ $\mathbf{A}_{s_2 \times t_1}^H$ ,  $\mathbf{B}_{s_2 \times t_1}^H$ ] = low_rank_sum( $\mathbf{A}_{s_2 \times t_1}^H$ ,  $\mathbf{B}_{s_2 \times t_1}^H$ , TY,  $\mathbf{B}_{r_2 \times t_1}^Z$ ); // (2.7.10)
28    Matrix SY = hmat_mult_dense( $\mathbf{Y}_{s_1 \times r_1}$ ,  $\mathbf{A}_{r_2 \times t_1}^Z$ );
29    Matrix SZ = hmat_transpose_mult_dense( $\mathbf{Z}_{r_2 \times t_2}$ ,  $\mathbf{B}_{s_1 \times r_2}^Y$ );
30    [ $\mathbf{A}_{s_1 \times t_2}^H$ ,  $\mathbf{B}_{s_1 \times t_2}^H$ ] = low_rank_sum( $\mathbf{A}_{s_1 \times t_2}^H$ ,  $\mathbf{B}_{s_1 \times t_2}^H$ ,  $\mathbf{A}_{s_1 \times r_2}^Y$ , SZ); // (2.7.11)
31    [ $\mathbf{A}_{s_1 \times t_2}^H$ ,  $\mathbf{B}_{s_1 \times t_2}^H$ ] = low_rank_sum( $\mathbf{A}_{s_1 \times t_2}^H$ ,  $\mathbf{B}_{s_1 \times t_2}^H$ , SY,  $\mathbf{B}_{r_1 \times t_2}^Z$ ); // (2.7.11)
32  }
33 }

```

SOLUTION of (2-7.i):

`hmat_forw_elim()` is made of two recursion, a call to the matrix-vector multiplication `hmv()` [Lecture → ??], whose computational cost has been estimated in Sub-problem (2-7.e), and a sum of two vectors of length 2^{p-1} . The computational cost $W_{fe}(p)$ therefore satisfies the recursive relation

$$W_{fe}(p) = 2W_{fe}(p-1) + 2^{p+1}pq + (p+1)2^p + 2^{p-1} .$$

We clearly have that $W_{fe}(0) = 1$. Hence, by inspecting the recursion or by means of generating functions, we obtain the following closed-form expression:

$$W_{fe}(p) = 2^{p-1} (2 + 4p + p^2 + 2pq + 2p^2q) .$$

SOLUTION of (3-1.a):

False. Counterexample:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 3 \\ 1 & 1 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

SOLUTION of (3-1.b):

We write two general lower triangular $(N + 1) \times (N + 1)$ Toeplitz matrices and directly compute their product:

$$\begin{bmatrix} f_0 & 0 & \dots & 0 \\ f_1 & f_0 & 0 & \\ \vdots & & \ddots & \\ f_N & f_{N-1} & \dots & f_0 \end{bmatrix} \cdot \begin{bmatrix} g_0 & 0 & \dots & 0 \\ g_1 & g_0 & 0 & \\ \vdots & & \ddots & \\ g_N & g_{N-1} & \dots & g_0 \end{bmatrix} = \begin{bmatrix} f_0 g_0 & 0 & \dots & \dots & 0 \\ f_1 g_0 + f_0 g_1 & f_0 g_0 & & & 0 \\ \vdots & \ddots & & & \\ \vdots & & \ddots & & \\ \sum_{i=0}^N f_{N-i} g_i & \sum_{i=0}^{N-1} f_{N-1-i} g_i & \dots & f_1 g_0 + f_0 g_1 & f_0 g_0 \end{bmatrix} \quad (3.1.2)$$

HINT 1 for (3-1.c): The sequence for the product matrix can be obtained by performing discrete convolutions [Lecture → ??] on the input sequences, which is equivalent to multiplying a lower Toeplitz matrix with a vector.

HINT 2 for (3-1.c): Recall from [NumCSE course → ??] how a discrete convolution of two vectors can efficiently be implemented based on **FFT**. You can use the code of `fastconv()` from [NumCSE course → ??].

SOLUTION of (3-1.c):

C++-code 3.1.3: Solution of (3-1.c)

```
2 Eigen::VectorXcd ltpMult(const Eigen::VectorXcd& f, const Eigen::VectorXcd& g) {
3   assert(f.size() == g.size() && "f and g vectors must have the same length!");
4   const std::size_t n = f.size();
5   Eigen::VectorXcd res(n);
6   // The sequence of the product matrix can be obtained through discrete
   // convolution
7   // Discrete convolution can be treated as multiplication of circulant
   // matrix, which
8   // can be represented by Fourier matrix refpar:circul
9
10  // Zero padding
11  Eigen::VectorXcd f_long = Eigen::VectorXcd::Zero(2 * n);
12  Eigen::VectorXcd g_long = Eigen::VectorXcd::Zero(2 * n);
13  f_long.head(n) = f;
14  g_long.head(n) = g;
15  // Periodic discrete convolution using FFT
16  res = pconvfft(f_long, g_long).head(n);
17  return res;
18 }
```

Get it on  [GitLab \(LowTriangToeplitz.cpp\)](#).

HINT 1 for (3-1.d): This code snippet shows the use of the C++ chrono library for measuring runtimes:

C++ code 3.1.4: Measuring runtime of a part of a C++ code.

Get it on [GitLab \(gravitationalforces.cpp\)](#).

```
2 std::pair<double, double> measureRuntimes(unsigned int n, unsigned int n_runs) {
3     assertm((n > 1), "At least two stars required!");
4     // Initialize star positions
5     std::vector<Eigen::Vector2d> pos = GravitationalForces::initStarPositions(n);
6     // All stars have equal (unit) mass
7     std::vector<double> mass(n, 1.0);
8     double ms_exact =
9         std::numeric_limits<double>::max(); // Time measured for exact
10        // evaluation
11    double ms_cluster = std::numeric_limits<
12        double>::max(); // Time taken for clustering-based evaluation
13    // Build quad tree of stars
14    StarQuadTreeClustering qt(pos, mass);
15    // Admissibility parameter
16    const double eta = 1.5;
17
18    // Runtime for exact computation of forces with effort O(n^2)
19    std::vector<Eigen::Vector2d> forces(n);
20    for (int r = 0; r < n_runs; ++r) {
21        auto t1_exact = std::chrono::high_resolution_clock::now();
22        forces = computeForces_direct(qt.starpos_, qt.starmasses_);
23        auto t2_exact = std::chrono::high_resolution_clock::now();
24        /* Getting number of milliseconds as a double. */
25        std::chrono::duration<double, std::milli> ms_double = (t2_exact - t1_exact);
26        ms_exact = std::min(ms_exact, ms_double.count());
27    }
28    std::cout << "n = " << n << " : runtime computeForces_direct= " << ms_exact
29        << "ms\n";
30
31    // Runtime for cluster-based approximate evaluation, cost O(n log n)
32    for (int r = 0; r < n_runs; ++r) {
33        auto t1_cluster = std::chrono::high_resolution_clock::now();
34        for (unsigned int j = 0; j < qt.n; j++) {
35            forces[j] = qt.forceOnStar(j, eta);
36        }
37        auto t2_cluster = std::chrono::high_resolution_clock::now();
38        /* Getting number of milliseconds as a double. */
39        std::chrono::duration<double, std::milli> ms_double =
40            (t2_cluster - t1_cluster);
41        ms_cluster = std::max(ms_cluster, ms_double.count());
42    }
43    std::cout << "n = " << n << " : runtime forceOnStar[eta=" << eta
44        << "] = " << ms_cluster << "ms\n";
45    return {ms_exact, ms_cluster};
46 }
```



For runtime measurements it is essential to compile your code in *Release Mode*: configure your CMake build system accordingly!

In addition, on concurrent operating systems runtimes should be measured several times, then taking the minimal measured time as the result.

SOLUTION of (3-1.d):

C++-code 3.1.5: Implementation of `runtimes_ltpMult()`.

```

2  std::tuple<double, double, double> runtimes_ltpMult(unsigned int N) {
3  // Runtime of matrix-matrix, matrix-vector and vector-vector
4  // multiplication
5  // in seconds
6  double s_dense, s_mv, s_ltp;
7
8  // Measure runtime several times
9  const int num_repetitions = 6;
10
11 // Sequence of Toeplitz matrices
12 Eigen::VectorXcd c(N), r(N), v(N);
13 // Initialization of the sequence of Toeplitz matrices
14 c = Eigen::VectorXcd::Random(N);
15 v = Eigen::VectorXcd::Constant(N, 1.0);
16
17 // Generate dense representation of Toeplitz matrices
18 r.setZero();
19 r(0) = c(0);
20 const Eigen::MatrixXcd T = toeplitz(c, r);
21 r(0) = v(0);
22 const Eigen::MatrixXcd V = toeplitz(v, r);
23
24 // Runtime when using Eigen's built-in multiplication of dense
25 // matrices
26 s_dense = std::numeric_limits<double>::max();
27 Eigen::MatrixXcd T_mult_V;
28 for (int k = 0; k < num_repetitions; k++) {
29 // Use C++ chrono library to measure runtimes
30 auto t1 = std::chrono::high_resolution_clock::now();
31 T_mult_V = T * V;
32 auto t2 = std::chrono::high_resolution_clock::now();
33
34 // Getting number of seconds as a double
35 std::chrono::duration<double> ms_double = (t2 - t1);
36
37 // Taking the minimal measured time as the result
38 s_dense = std::min(s_dense, ms_double.count());
39 }
40
41 // Runtime when multiplying one of the Toeplitz matrix
42 // with the vector defining the second
43 s_mv = std::numeric_limits<double>::max();
44 Eigen::VectorXcd T_mult_v;
45 for (int k = 0; k < num_repetitions; k++) {
46 // Use C++ chrono library to measure runtimes
47 auto t1 = std::chrono::high_resolution_clock::now();
48 T_mult_v = T * v;
49 auto t2 = std::chrono::high_resolution_clock::now();
50
51 // Getting number of seconds as a double
52 std::chrono::duration<double> ms_double = (t2 - t1);
53
54 // Taking the minimal measured time as the result

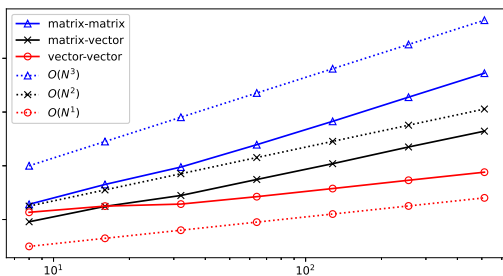
```

```

53     s_mv = std::min(s_mv, ms_double.count());
54 }
55
56 // Runtime when using ltpMult() from Sub-problem (3-1.c)
57 s_ltp = std::numeric_limits<double>::max();
58 Eigen::VectorXcd c_conv_v;
59 for (int k = 0; k < num_repetitions; k++) {
60     // Use C++ chrono library to measure runtimes
61     auto t1 = std::chrono::high_resolution_clock::now();
62     c_conv_v = ltpMult(c, v);
63     auto t2 = std::chrono::high_resolution_clock::now();
64
65     // Getting number of seconds as a double
66     std::chrono::duration<double> ms_double = (t2 - t1);
67
68     // Taking the minimal measured time as the result
69     s_ltp = std::min(s_ltp, ms_double.count());
70 }
71 return {s_dense, s_mv, s_ltp};
72 }

```

Get it on  GitLab (LowTriangToeplitz.cpp).



◁ Measure runtimes for three ways of computing the product of lower-triangular Toeplitz matrices.

CPU : Apple M2 Pro
OS : 13.6 (22G120)
Compiler: AppleClang 14.0.3.14030022
Options : -Wall -Wextra -Wconversion

The asymptotic computational efforts for $N \rightarrow \infty$ are

- N^3 for direct matrix multiplication,
- $O(N^2)$ for matrix \times vector, and
- $O(N \log N)$ for the FFT-based implementation.

This is clearly reflected by the runtimes.

HINT 1 for (3-1.e): As explained in [NumCSE course → ??], by zero-padding, the multiplication of a Toeplitz matrix with a vector can be reduced to the multiplication of a circulant matrix with a vector, which, as periodic discrete convolution, can be computed by means of FFT, see [NumCSE course → ??].

SOLUTION of (3-1.e):

We can implement an auxiliary function `teopMatVecMult()` for the efficient multiplication of a Toeplitz matrix with a vector.

C++-code 3.1.6: Efficient implementation of the Multiplication of a Toeplitz matrix with a vector

```
2 Eigen::VectorXcd teopMatVecMult(const Eigen::VectorXcd& c,
3                               const Eigen::VectorXcd& r,
4                               const Eigen::VectorXcd& x) {
5     assert(c.size() == x.size() && r.size() == x.size() &&
6           "c, r, x have different lengths!");
7
8     const std::size_t n = c.size();
9     Eigen::VectorXcd y(2 * n);
10    // The multiplication of a Toeplitz matrix with a vector can be
11    // reduced
12    // to the multiplication of circulant matrix with a vector
13    // refemp:tpmv,
14    // which can be computed by FFT as in refpar:circul
15    Eigen::VectorXcd cr_tmp(2 * n), x_tmp(2 * n);
16
17    // Assemble the sequence of Toeplitz matrix
18    cr_tmp.head(n) = c;
19    cr_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
20    // Need to reverse the row sequence and add it to the end
21    cr_tmp.tail(n - 1) = r.tail(n - 1).reverse();
22
23    x_tmp.head(n) = x;
24    // Zero padding
25    x_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
26
27    // Periodic discrete convolution
28    y = pconvfft(cr_tmp, x_tmp);
29    return y.head(n);
30 }
```

Get it on  GitLab (LowTriangToeplitz.cpp).

C++-code 3.1.7: Implementation of `ltpSolve`

```
2 Eigen::VectorXcd ltpSolve(const Eigen::VectorXcd& f,
3                           const Eigen::VectorXcd& y) {
4     assert(f.size() == y.size() && "f and y vectors must have the same length!");
5     assert(abs(f(0)) > 1e-10 &&
6           "Lower triangular Toeplitz matrix must be invertible!");
7     assert(log2(f.size()) == floor(log2(f.size())) &&
8           "Size of f must be a power of 2!");
9
10    const std::size_t n = f.size();
11    Eigen::VectorXcd u(n);
12    // When it reduces to scalar, solve directly
13    if (n == 1) {
14        return y.cwiseQuotient(f);
15    }
16 }
```

```

16
17 // Solve by recursion:
18 // Solve for the first half of u
19 const Eigen::VectorXcd u_head = ltpSolve(f.head(n / 2), y.head(n / 2));
20 // Update the right hand side by subtracting the product of the lower
    left part
21 // of Toeplitz matrix and the first half of u
22 const Eigen::VectorXcd t =
23     y.tail(n / 2) -
24     toepMatVecMult(f.tail(n / 2), f.segment(1, n / 2).reverse(), u_head);
25 // Solve for the second half of u
26 // (upper left part of the matrix is identical to the lower right
    part)
27 const Eigen::VectorXcd u_tail = ltpSolve(f.head(n / 2), t);
28 // Assemble results
29 u << u_head, u_tail;
30 return u;
31 }

```

Get it on  [GitLab](#) (LowTriangToeplitz.cpp).

SOLUTION of (3-1.f):

C++-code 3.1.8: Implementation of `runtimes_ltpSolve`.

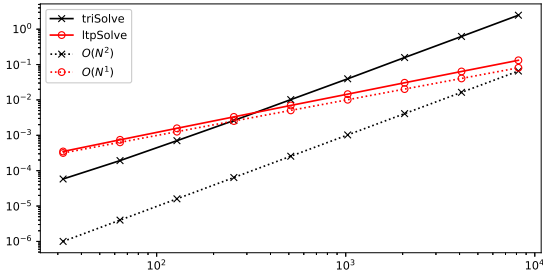
```

2  std::pair<double, double> runtimes_ltpSolve(unsigned int N) {
3  // Runtime of Eigen's triangular solver and ltpSolve() in seconds
4  double s_tria, s_ltp;
5
6  // Measure runtime several times
7  int num_repetitions = 6;
8
9  // Initialization of Toeplitz matrix and vector
10 Eigen::VectorXcd c(N), r(N), v(N);
11 for (int i = 0; i < N; i++) {
12     c(i) = i + 1;
13 }
14 v = Eigen::VectorXcd::Constant(N, 1.0);
15 r.setZero();
16 r(0) = c(0);
17
18 // Generate dense representation of Toeplitz and rhs vector of LSE
19 Eigen::MatrixXcd T = toeplitz(c, r);
20 Eigen::VectorXcd T_mult_v = ltpMult(c, v);
21
22 // Runtime when using Eigen's built-in triangular solver
23 Eigen::VectorXcd u_sol;
24 s_tria = std::numeric_limits<double>::max();
25 for (int k = 0; k < num_repetitions; k++) {
26     // Use C++ chrono library to measure runtimes
27     auto t1 = std::chrono::high_resolution_clock::now();
28     u_sol = T.triangularView<Eigen::Lower>().solve(T_mult_v);
29     auto t2 = std::chrono::high_resolution_clock::now();
30
31     // Getting number of seconds as a double
32     std::chrono::duration<double> ms_double = (t2 - t1);
33
34     // Taking the minimal measured time as the result
35     s_tria = std::min(s_tria, ms_double.count());
36 }
37
38 // Runtime when using ltpSolve() from Sub-problem (3-1.e)
39 Eigen::VectorXcd u_rec;
40 s_ltp = std::numeric_limits<double>::max();
41 for (int k = 0; k < num_repetitions; k++) {
42     // Use C++ chrono library to measure runtimes
43     auto t1 = std::chrono::high_resolution_clock::now();
44     u_rec = ltpSolve(c, T_mult_v);
45     auto t2 = std::chrono::high_resolution_clock::now();
46
47     // Getting number of seconds as a double
48     std::chrono::duration<double> ms_double = (t2 - t1);
49
50     // Taking the minimal measured time as the result
51     s_ltp = std::min(s_ltp, ms_double.count());
52 }
53

```

```
54 return {s_tria, s_ltp};  
55 }
```

Get it on  GitLab (LowTriangToeplitz.cpp).



◁ Measured runtimes for different ways to solve a lower-triangular Toeplitz system.

CPU : Apple M2 Pro

OS : 13.6 (22G120)

Compiler: AppleClang 14.0.3.14030022

Options : -Wall -Wextra -Wconversion

For large n the solution algorithm of `ltpSolve()` is significantly faster than backward elimination, which incurs $O(n^2)$ asymptotic cost for $n \rightarrow \infty$.

SOLUTION of (3-2.a):

We read from (3.2.1)

$$f(t) = \begin{cases} \frac{1}{\sqrt{t}} & , \text{ if } t \geq 0 , \\ 0 & , \text{ if } t < 0 . \end{cases}$$

Note that this is a causal function with a singularity at $t = 0$! Such singular convolution kernels are fairly common in applications.

SOLUTION of (3-2.j):

C++-code 3.2.17: Solution of (3-2.j)

```
2  template <typename FUNC>
3  Eigen::VectorXd cq_ieul_abel(const FUNC& y, size_t N) {
4      Eigen::VectorXd u(N + 1);
5      Eigen::VectorXd w(N + 1);
6      w(0) = 1.;
7      // Calculate weights of convolution quadrature based on
8      // Sub-problem (3-2.h)
9      for (int l = 1; l < N + 1; ++l) {
10         w(l) = w(l - 1) * (l - 0.5) / l; // denominator is factorial
11     }
12     w *= std::sqrt(M_PI / N);
13
14     // Solve the convolution quadrature:
15     // Generate points on the grid
16     Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
17     // Set up the rhs vector of the LSE
18     Eigen::VectorXd y_N(N + 1);
19     for (int i = 0; i < N + 1; ++i) {
20         y_N(i) = y(grid(i));
21     }
22     // Set up the coefficient matrix
23     Eigen::MatrixXd T = toeplitz_triangular(w);
24     // Solve the lse with Eigen's build-in triangular elimination solver
25     u = T.triangularView<Lower>().solve(y_N);
26     return u;
27 }
```

Get it on  [GitLab \(abelintegralequation.h\)](https://gitlab.com/abelintegralequation.h).

SOLUTION of (3-2.k):

C++-code 3.2.18: Solution of (3-2.k)

```
2  template <typename FUNC>
3  Eigen::VectorXd cq_bdf2_abel(const FUNC& y, size_t N) {
4      Eigen::VectorXd u(N + 1);
5      Eigen::VectorXd w1(N + 1);
6      w1(0) = 1.;
7      // Calculate weights of convolution quadrature based on
8      // Sub-problem (3-2.i)
9      // Taylor series expansion of the first factor
10     for (int l = 1; l < N + 1; ++l) {
11         w1(l) = w1(l - 1) * (l - 0.5) / l; // denominator is factorial
12     }
13     // Taylor series expansion of the second factor
14     Eigen::VectorXd w2 = w1;
15     for (int l = 1; l < N + 1; ++l) {
16         w2(l) /= pow(3, l);
17     }
18     // Full expansion by Cauchy product
19     Eigen::VectorXd w = myconv(w1, w2).head(N + 1).real();
20     w *= std::sqrt(M_PI / N) * std::sqrt(2. / 3.);
21     // Solve the convolution quadrature:
22     // Generate points on the grid
23     Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
24     // Set up the rhs vector of the LSE
25     Eigen::VectorXd y_N(N + 1);
26     for (int i = 0; i < N + 1; ++i) {
27         y_N(i) = y(grid(i));
28     }
29     // Set up the coefficient matrix
30     Eigen::MatrixXd T = toeplitz_triangular(w);
31     // Solve the lse with Eigen's build-in triangular elimination solver
32     u = T.triangularView<Lower>().solve(y_N);
33     return u;
34 }
```

Get it on  GitLab (abelintegralequation.h).

SOLUTION of (3-2.I):

C++-code 3.2.19: Solution of (3-2.I)

```

2
3
4 // Exact solution
5 auto u = [](double t) { return 2. / M_PI * sqrt(t); };
6 auto y = [](double t) { return t; };
7
8 double err_max, err_max_alt;
9 cout << "\n\nConvolution Quadrature, Implicit Euler\n\n";
10 for (int N = 16; N <= 2048; N <<= 1) {
11     // Generate points on the grid
12     const Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
13     // Exact solution at grid points
14     const Eigen::VectorXd u_ex = Eigen::VectorXd::NullaryExpr(
15         N + 1, [&](Eigen::Index i) { return u(grid(i)); });
16
17     // Solution using convolution quadrature based on implicit Euler
18     // method
19     const Eigen::VectorXd u_app = AbelIntegralEquation::cq_ieul_abel(y, N);
20     // Maximum norm of discretization error
21     const Eigen::VectorXd diff = u_ex - u_app;
22     err_max = diff.cwiseAbs().maxCoeff();
23
24     std::cout << "N = " << N << std::setw(15) << "Max = " << std::scientific
25         << std::setprecision(3) << err_max;
26     if (N > 16) {
27         std::cout << " EOC = " << std::log2(err_max_alt / err_max);
28     }
29     std::cout << '\n';
30     err_max_alt = err_max;
31 }
32
33 std::cout << "\n\nConvolution Quadrature, BDF-2\n" << '\n';
34 for (int N = 16; N <= 2048; N <<= 1) {
35     // Generate points on the grid
36     Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
37     // Exact solution at grid points
38     Eigen::VectorXd u_ex(N + 1);
39     for (int i = 0; i < N + 1; ++i) {
40         u_ex(i) = 2. / M_PI * sqrt(grid(i));
41     }
42
43     // Solution using convolution quadrature based on BDF-2 method
44     const Eigen::VectorXd u_app = AbelIntegralEquation::cq_bdf2_abel(y, N);
45     // Maximum norm of discretization error
46     const Eigen::VectorXd diff = u_ex - u_app;
47     err_max = diff.cwiseAbs().maxCoeff();
48
49     std::cout << "N = " << N << std::setw(15) << "Max = " << std::scientific
50         << std::setprecision(3) << err_max;
51     if (N > 16) {
52         std::cout << " EOC = " << std::log2(err_max_alt / err_max);
53     }

```

```

54     std::cout << '\n';
55
56     err_max_alt = err_max;
57 }
58 }

```

Get it on  [GitLab \(abelintegralequation_main.cpp\)](#).

N	Implicit Euler	BDF-2	EOC
16	1.811e-02	1.359e-02	
32	1.280e-02	9.611e-03	5.000e-01
64	9.054e-03	6.796e-03	5.000e-01
128	6.402e-03	4.806e-03	5.000e-01
256	4.527e-03	3.398e-03	5.000e-01
512	3.201e-03	2.403e-03	5.000e-01
1024	2.263e-03	1.699e-03	5.000e-01
2048	1.600e-03	1.201e-03	5.000e-01

We observe algebraic convergence with a low, fractional rate (~ 0.5). This behavior of the methods is caused by the square-root singularity of the solution at $t = 0$, because the theory of convolution quadrature assumes higher regularity of the argument function of the convolution operator.

SOLUTION of (3-2.b):

❶ Consider [Lecture → ??] with $q = -\frac{1}{2}$. Then the Laplace transform is

$$\mathcal{L}f(s) = \frac{\Gamma(\frac{1}{2})}{\sqrt{s}} = \sqrt{\frac{\pi}{s}}, \quad s \in \mathbb{C} \setminus]-\infty, 0], \quad (3.2.2)$$

where we used special values of the [Gamma function](#).

❷ You can also find this formula in tables of Laplace transforms, for instance [here](#).

❸ MATHEMATICA offers the built-in function `LaplaceTransform[]`, e.g. `LaplaceTransform[1/Sqrt[t], t, s]` outputs (3.2.2).

❹ You can directly compute the Laplace transform based on the formula from [Lecture → ??]. We first assume $s > 0$ and get by substituting $\eta := st$

$$(\mathcal{L}f)(s) = \int_0^\infty t^{-\frac{1}{2}} e^{-st} dt = \int_0^\infty \sqrt{\frac{s}{\eta}} e^{-\eta} s^{-1} d\eta = \frac{1}{\sqrt{s}} \Gamma(\frac{1}{2}).$$

Analytic continuation makes it possible to view $\mathcal{L}f$ as an analytic function $\mathcal{L}f : \mathbb{C} \setminus]-\infty, 0] \rightarrow \mathbb{C}$.

HINT 1 for (3-2.c): Recall that

$$\int_{-1}^1 \frac{1}{\sqrt{1-\zeta^2}} d\zeta = [\arcsin(t)]_{-1}^1 = \pi.$$

HINT 2 for (3-2.c): Write down the double integral defining A^2 and swap the order of integration. Then use the substitution

$$x' := 2\frac{x-t}{\xi-t} - 1, \quad (3.2.4)$$

denoting x the inner integration variable in A^2 . ┘

SOLUTION of (3-2.c):

To begin with, for smooth and causal $u : \mathbb{R} \rightarrow \mathbb{R}$

$$(A^2u)(t) = \int_0^t \int_0^x \frac{u(\xi)}{\sqrt{t-x}\sqrt{x-\xi}} d\xi dx .$$

All integrals exist as improper integrals. Now, let us change the order of integration (Fubini's theorem), making sure that we integrate on the same triangular domain:

$$(A^2u)(t) = \int_0^t \int_{\xi}^t \frac{1}{\sqrt{t-x}\sqrt{x-\xi}} dx u(\xi) d\xi .$$

Using the substitution

$$x' := 2 \frac{x-t}{\xi-t} - 1 \quad \implies \quad dx' = \frac{2}{\xi-t} dx ,$$

with inverse

$$x = (x' + 1) \frac{1}{2} (\xi - t) + t = (1 - x') \frac{1}{2} (t - \xi) + \xi ,$$

we obtain:

$$\begin{aligned} (A^2u)(t) &= \int_0^t \int_{-1}^{-1} \frac{1}{\sqrt{(x'+1)\frac{1}{2}(t-\xi)}\sqrt{(1-x')\frac{1}{2}(t-\xi)}} \frac{\xi-t}{2} dx' u(\xi) d\xi \\ &= \int_0^t \int_{-1}^{+1} \frac{1}{\sqrt{1-(x')^2}} dx' u(\xi) d\xi = \pi \int_0^t u(\xi) d\xi . \end{aligned}$$

In the last but one step we inserted $\int_{-1}^1 \frac{1}{\sqrt{1-\xi^2}} d\xi = [\arcsin(t)]_{-1}^1 = \pi$.

Remark. The relationship (3.2.3) justifies calling the Abel integral operator a “square root of integration”.

SOLUTION of (3-2.d):

The Abel integral equation (3.2.1) is converted into the discrete linear variational problem: seek $u_h \in V_p$ such that

$$a(u, v) := \int_0^1 (Au)(t)v(t) dt = \ell(v) := \int_0^1 y(t)v(t) dt \quad \forall v \in V_p. \quad (3.2.7)$$

We recall the abstract principles of Galerkin discretization [[NumPDE course](#) → ??]. They tell us that, when using a basis $\mathfrak{B} := (b_h^1, \dots, b_h^N)$, $N := \dim V_p$, of V_p , we end up with the linear system of equations

$$\mathbf{A}\vec{\mu} := \left[(a)(b_h^j, b_h^i) \right]_{i,j=1}^N \vec{\mu} = \vec{\phi} := \left[\ell(b_h^i) \right]_{i=1}^N \quad (3.2.8)$$

for the unknown vector $\vec{\mu} \in \mathbb{R}^N$ of basis expansion coefficients of the Galerkin solution u_h . Using the monomial basis (3.2.6), $b_h^i = \zeta \mapsto \zeta^i$, $i = 1, \dots, N$, the entries of the Galerkin matrix are

$$(\mathbf{A})_{i,j} = \int_0^1 \int_0^t \frac{t^i \zeta^j}{\sqrt{t-\zeta}} d\zeta dt, \quad i, j = 1, \dots, p, \quad (\vec{\phi})_i = \int_0^1 t^i y(t) dt, \quad i = 1, \dots, p. \quad (3.2.9)$$

HINT 1 for (3-2.e): Use the substitution

$$\eta := \sqrt{t - \xi}, \quad 0 \leq \xi \leq t \Leftrightarrow \xi = t - \eta^2, \quad 0 \leq \eta \leq \sqrt{t}, \quad (3.2.10)$$

the evaluation of the relevant integrals is reduced to integrating polynomials. ┘

SOLUTION of (3-2.e):

We have to compute the numbers $(\mathbf{A})_{i,j}$, $i, j = 1, \dots, p$, using the formula (3.2.9):

$$(\mathbf{A})_{i,j} = \int_0^1 \int_0^t \frac{t^i \zeta^j}{\sqrt{t-\zeta}} d\zeta dt.$$

From the substitution $\eta := \sqrt{t-\zeta}$, $0 \leq \zeta \leq t \iff \zeta = t - \eta^2$, $0 \leq \eta \leq \sqrt{t}$, it follows that

$$\frac{d\eta}{d\zeta} = -\frac{1}{2\sqrt{t-\zeta}} \iff -2\eta d\eta = d\zeta.$$

This yields the simpler expression

$$(\mathbf{A})_{i,j} = 2 \int_0^1 \int_0^{\sqrt{t}} t^i (t - \eta^2)^j d\eta dt.$$

After another substitution

$$s := \frac{\eta^2}{t} \iff \eta = \sqrt{ts} \iff d\eta = \frac{1}{2} \sqrt{\frac{t}{s}} ds,$$

we end up with

$$\begin{aligned} (\mathbf{A})_{i,j} &= \int_0^1 \int_0^1 t^i (t - ts)^j \sqrt{\frac{t}{s}} ds dt \\ &= \int_0^1 t^{i+j+\frac{1}{2}} dt \int_0^1 s^{-\frac{1}{2}} (1-s)^j ds \\ &= \frac{1}{i+j+\frac{3}{2}} B(j+1, \frac{1}{2}) \\ &= \frac{1}{i+j+\frac{3}{2}} \frac{\sqrt{\pi} \Gamma(j+1)}{\Gamma(j+\frac{3}{2})}, \end{aligned}$$

where $B(\cdot, \cdot)$ is the Euler beta function.

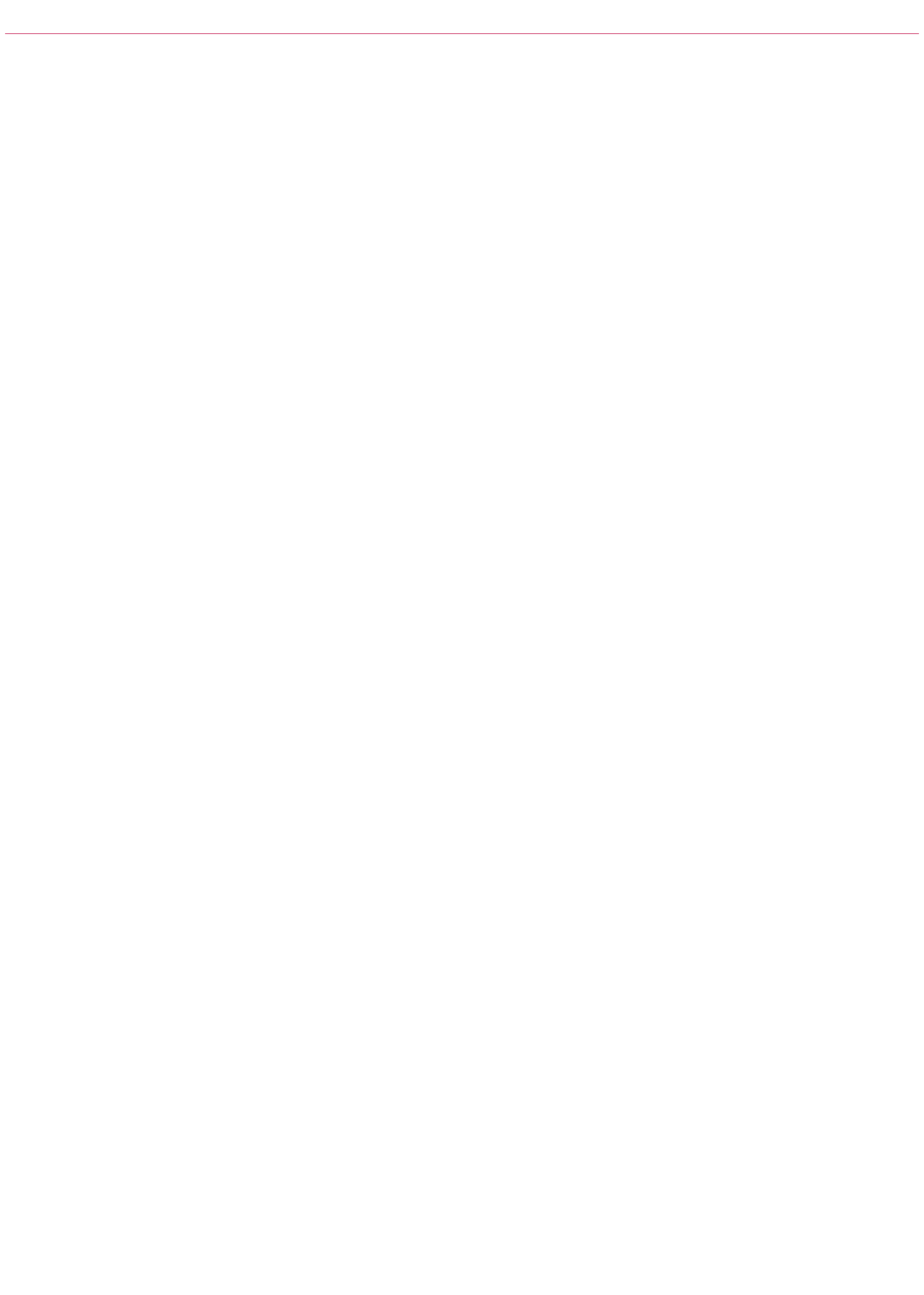
SOLUTION of (3-2.f):

C++-code 3.2.11: Solution of (3-2.f)

```

2  template <typename FUNC>
3  Eigen::VectorXd poly_spec_abel(const FUNC& y, std::size_t p, double tau) {
4      Eigen::MatrixXd A = Eigen::MatrixXd::Zero(p + 1, p + 1);
5      Eigen::VectorXd b = Eigen::VectorXd::Zero(p + 1);
6
7      // generate Gauss-Legendre quadrature points and weights
8      const auto [gauss_pts_p, gauss_wht_p] = gauleg(0., 1., p);
9
10     // set up the Galerkin matrix and rhs vector
11
12     // Precompute factors related to gamma function
13     // Watch out for integer division
14     const Eigen::VectorXd gamma_factor =
15         Eigen::VectorXd::NullaryExpr(p + 1, [](Eigen::Index j) {
16             return std::tgamma(j + 1) / std::tgamma(j + 3. / 2.);
17         });
18     const double sqrt_pi = std::sqrt(M_PI);
19
20     for (int i = 0; i <= p; i++) {
21         // Set up Galerkin matrix based on Sub-problem (3-2.e)
22         for (int j = 0; j <= p; j++) {
23             A(i, j) = sqrt_pi * gamma_factor[j] / (1.5 + i + j);
24         }
25
26         // Set up the rhs vector of the LSE based on Gauss-Legendre
27         // quadrature
28         for (int k = 0; k < p; k++) {
29             const double tk = gauss_pts_p(k);
30             const double wk = gauss_wht_p(k);
31             // Based on (3.2.9)
32             b(i) += wk * std::pow(tk, i) * y(tk);
33         }
34     }
35
36     // linear system solve using QR decomposition
37     const Eigen::ColPivHouseholderQR<Eigen::MatrixXd> solver(A);
38     const Eigen::VectorXd x = solver.solve(b);
39
40     // generate points on the grid
41     const std::size_t N = std::round(1. / tau);
42     const Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
43     Eigen::VectorXd u = Eigen::VectorXd::Zero(N + 1);
44
45     // generate solution at grid points
46     for (int j = 0; j <= p; j++) {
47         u += x(j) * grid.array().pow(j).matrix();
48     }
49
50     return u;
51 }

```



SOLUTION of (3-2.g):

C++-code 3.2.12: Solution of (3-2.g)

```

2  {
3  // Exact solution
4  auto u = [](double t) { return 2. / M_PI * sqrt(t); };
5  auto y = [](double t) { return t; };
6
7  // Generate points on the grid
8  const double tau = 0.01;
9  const std::size_t N = std::round(1. / tau);
10 const Eigen::VectorXd grid = Eigen::VectorXd::LinSpaced(N + 1, 0., 1.);
11 // Exact solution at grid points
12 const Eigen::VectorXd u_ex = Eigen::VectorXd::NullaryExpr(
13     N + 1, [&](Eigen::Index i) { return u(grid(i)); });
14
15 std::cout << "\nSpectral Galerkin\n\n";
16 double err_max, err_max_alt;
17 for (int p = 2; p <= 10; ++p) {
18     // Solution using Galerkin discretization with a polynomial basis
19     const Eigen::VectorXd u_app =
20         AbelIntegralEquation::poly_spec_abel(y, p, tau);
21     // Maximum norm of discretization error
22     const Eigen::VectorXd diff = u_ex - u_app;
23     err_max = diff.cwiseAbs().maxCoeff();
24     const double dp = p;
25
26     std::cout << "p = " << p << std::setw(15) << "Max = " << std::scientific
27         << std::setprecision(3) << err_max << std::setw(15);
28     if (p > 2) {
29         std::cout << " EOC = "
30             << std::log2(err_max_alt / err_max) /
31             std::log2((dp + 1) / dp);
32     }
33     std::cout << '\n';
34
35     err_max_alt = err_max;
36 }
37 }

```

Get it on  [GitLab \(abelintegralequation_main.cpp\)](#).

p	Max error	EOC
2	1.025e-01	
3	7.690e-02	1.00 e+00
4	6.168e-02	9.883 e-01
5	5.155e-02	9.844 e-01
6	4.430e-02	9.831 e-01
7	3.885e-02	9.830 e-01
8	3.460e-02	9.834 e-01
9	3.119e-02	9.842 e-01
10	2.837e-02	9.977 e-01

We observe rather slow convergence that we conjecture to be $\mathcal{O}(p^{-1})$.

Due to the square root singularity of the solution at $t = 0$, even the polynomial best approximation converges only algebraically with some (fractional) rate < 1 .

HINT 1 for (3-2.h): Recall from [Lecture → ??] that the defining **generating-function formula** for the CQ-weights relies on Taylor series expansion

$$F\left(\frac{1-z}{\tau}\right) = \sum_{n=0}^{\infty} w_n^{F,\tau} z^n . \quad (3.2.13)$$

in the neighbourhood of $z = 0$.

┘

HINT 2 for (3-2.h): Consult the formulas presented in [Lecture → ??], Part (I).

SOLUTION of (3-2.h):

The gist of convolution quadrature is that given the convolution defined by $f * u = \int_0^t f(t - \xi)u(\xi) d\xi$ in

Sub-problem (3-2.a), we want to find weights w_ℓ such that [Lecture → ??]

$$\int_0^{n\tau} f(t - \xi)u(\xi) d\xi \approx \sum_{\ell=0}^n w_{n-\ell}^\tau u(\ell\tau), \quad n = 0, \dots, N, \quad \tau = \frac{1}{N}, \quad N \in \mathbb{N},$$

where $\tau > 0$ is the discretization step size.

As discussed in [Lecture → ??], the **CQ-weights** w_ℓ^τ can be found through matching coefficients of Taylor series expansions:

$$\sum_{\ell=0}^{\infty} w_\ell^\tau z^\ell = F\left(\frac{\delta(z)}{\tau}\right), \quad (3.2.14)$$

where $F = \mathcal{L}f$ denotes the Laplace transform [Lecture → ??] of the kernel f . In the case of convolution quadrature based on implicit Euler timestepping we have, $\delta(z) = 1 - z$.

We can compute the CQ-weights $w_\ell^{F,\tau}$ by applying a Taylor expansion at $z = 0$ to the right-hand side of Eq. (3.2.14) using the result of Sub-problem (3-2.b), see also [Lecture → ??],

$$\sum_{\ell=0}^{\infty} w_\ell z^\ell = \sqrt{\frac{\pi\tau}{1-z}} = \sqrt{\pi\tau} \left(1 + \sum_{\ell=1}^{\infty} \left(-\frac{1}{2} + 1\right) \cdot \left(-\frac{1}{2} + 2\right) \cdots \left(-\frac{1}{2} + \ell\right) \frac{z^\ell}{\ell!} \right).$$

Matching equal powers of z yields

$$w_0^{F,\tau} = \sqrt{\pi\tau}, \quad w_\ell^{F,\tau} = \frac{\sqrt{\pi\tau}}{\ell!} \left(-\frac{1}{2} + 1\right) \cdot \left(-\frac{1}{2} + 2\right) \cdots \left(-\frac{1}{2} + \ell\right).$$

HINT 1 for (3-2.i): In the case of the BDF-2 method we find

$$\delta(z) = \frac{1}{2}z^2 - 2z + \frac{3}{2} = \frac{1}{2}(1-z)(3-z),$$

which can be used in the general defining formula for the CQ-weights

$$F\left(\frac{\delta(z)}{\tau}\right) = \sum_{n=0}^{\infty} w_n^{F,\tau} z^n. \quad (3.2.16)$$

┘

HINT 2 for (3-2.i): The function $\delta(z)$ is a product of monomials. Also $F\left(\frac{\delta(z)}{\tau}\right)$ can be written in product form and the Taylor series of the factors can be found as in Sub-problem (3-2.h). Their **Cauchy product** yields the Taylor series of $F\left(\frac{\delta(z)}{\tau}\right)$. ┘

SOLUTION of (3-2.i):

Proceeding similar to the solution of Sub-problem (3-2.h), one difference here being the function $\delta(z) = (1 - z)$

$$\sum_{\ell=0}^{\infty} w_{\ell} z^{\ell} = \sqrt{\frac{\pi\tau}{(1-z) + \frac{1}{2}(1-z)^2}} = \sqrt{\pi\tau} \sqrt{\frac{1}{1-z}} \sqrt{\frac{1}{\frac{3}{2}\left(1 - \frac{1}{3}z\right)}}$$

Now, a Taylor series expansion can be applied to each factor separately. The first factor will lead to the same expansion as in Sub-problem (3-2.h). The second factor leads to

$$\sqrt{\frac{1}{\frac{3}{2}\left(1 - \frac{1}{3}z\right)}} = \sqrt{\frac{2}{3}} \left(1 + \sum_{\ell=1}^{\infty} \frac{1}{3^{\ell}} \left(-\frac{1}{2} + 1\right) \cdot \left(-\frac{1}{2} + 2\right) \cdots \left(-\frac{1}{2} + \ell\right) \frac{z^{\ell}}{\ell!}\right)$$

Finally, a Cauchy product yields the full expansion:

$$\left(\sum_{\ell=0}^{\infty} w_{\ell}^1 z^{\ell}\right) \left(\sum_{\ell=0}^{\infty} w_{\ell}^2 z^{\ell}\right) = \sum_{\ell=0}^{\infty} \left(\sum_{k=0}^{\ell} w_k^1 w_{\ell-k}^2\right) z^{\ell}.$$

Hence, the weights $w_{\ell}^{F,\tau}$ can be computed by a discrete convolution of the coefficients of the two Taylor series expansions.

SOLUTION of (3-3.a):

Following the usual steps [[NumPDE course](#) → ??] we derive the **variational formulation** of (3.3.1). First we test with an arbitrary $v \in H^1(]0, 1[)$, then we integrate, then carry out integration by parts, and, finally, take into account the boundary conditions (3.3.1c) and (3.3.1b):

$$\begin{aligned} & - \int_0^1 \frac{d^2 u}{dx^2} v \, dx + \int_0^1 \sin(\pi x) u v \, dx = 0 \quad \forall v \in H^1(]0, 1[) , \\ & \int_0^1 \frac{du}{dx} \frac{dv}{dx} \, dx - \left[\frac{du}{dx} v \right]_0^1 + \int_0^1 \sin(\pi x) u v \, dx = 0 \quad \forall v \in H^1(]0, 1[) , \\ & \int_0^1 \frac{du}{dx} \frac{dv}{dx} \, dx + (f * u(1, \cdot))(t) v(1) + g(t) v(0) + \int_0^1 \sin(\pi x) u v \, dx = 0 \quad \forall v \in H^1(]0, 1[) , \\ & \int_0^1 \left(\frac{du}{dx} \frac{dv}{dx} + \sin(\pi x) u v \right) \, dx + (f * u(1, \cdot))(t) v(1) = -g(t) v(0) \quad \forall v \in H^1(]0, 1[) . \end{aligned}$$

Not that we use the Sobolev space $H^1(]0, 1[)$ as both trial and test space, because no Dirichlet boundary conditions are imposed. Also remember that the point evaluations $v \mapsto v(0)$ and $v \mapsto v(1)$ are continuous linear functionals on $H^1(]0, 1[)$ [[NumPDE course](#) → ??].

Writing b_h^i , $i = 1, \dots, N := M + 1$, for the standard hat/tent basis functions (lexikographically ordered) that we use as a basis for the finite-element space $\mathcal{S}_1^0(\mathcal{M}) \subset H^1(]0, 1[)$, without using numerical quadrature the entries of **A** and **B** are given by

$$\begin{aligned} (\mathbf{A})_{i,j} &= \int_0^1 \frac{db_h^j}{dx}(x) \frac{db_h^i}{dx}(x) + \sin(\pi x) b_h^j(x) b_h^i(x) \, dx , \quad i, j \in \{1, \dots, N\} . \quad (3.3.4) \\ (\mathbf{B})_{i,j} &= b_h^i(1) b_h^j(1) , \end{aligned}$$

An analytic evaluation is possible, but tedious, whereas, thanks to $b_h^i(hj) = \delta_{ij}$, the use of the local trapezoidal quadrature rule immediately gives the entries of the tri-diagonal matrix **A**:

$$\begin{aligned} (\mathbf{A})_{i,j} &= \begin{cases} \frac{1}{h} & \text{for } i = j = 1 , \\ \frac{2}{h} + h \sin(\pi i h) & \text{for } i = j , \quad i = 2, \dots, N - 1 , \\ \frac{1}{h} & \text{for } i = j = N , \\ -\frac{1}{h} & \text{for } |i - j| = 1 , \\ 0 & \text{else for } i, j \in \{1, \dots, N\} , \quad |i - j| > 1 , \end{cases} \\ (\mathbf{B})_{i,j} &= \begin{cases} 1 & , \text{ if } i = j = N , \\ 0 & \text{ else,} \end{cases} \\ (\vec{\varphi}(t))_i &= \begin{cases} g(t) & \text{for } i = 0 , \\ 0 & \text{for } i \in \{2, \dots, N\} . \end{cases} \end{aligned}$$

Also refer to [[NumPDE course](#) → ??] for details.

SOLUTION of (3-3.b):

Similar to [Lecture → ??], the complex logarithm term in F is analytic in $\mathbf{C} \setminus [-\infty, 0]$. The term $\frac{1}{s^2+1}$ is not defined for $s \in \{-i, +i\}$. Hence, the domain of analyticity of F is $\mathbf{C} \setminus \{[-\infty, 0] \cup \{-i, +i\}\}$.

SOLUTION of (3-3.c):

C++-code 3.3.5: Solution of (3-3.c) [Get it on !\[\]\(42d21e58927ef419cc45be9cb0912795_img.jpg\) GitLab \(absorbingboundarycondition.h\).](#)

```
2 template <typename FFUNC, typename DFUNC>
3 VectorXd cqweights_by_dft(const FFUNC& F, const DFUNC& delta, double tau,
4 size_t M) {
5     Eigen::VectorXcd w = Eigen::VectorXd::Zero(M + 1);
6     // Setting  $r = EPS^{\frac{1}{2M+2}}$ , as discussed in Rem. 3.4.3.20
7     double r = std::pow(10, -16.0 / (2 * M + 2));
8     // Initialize vector for the evaluations in the Laplace domain
9     Eigen::VectorXcd f = Eigen::VectorXd::Zero(M + 1);
10    // Imaginary unit stored in imag
11    std::complex<double> imag = std::complex<double>(0, 1);
12    // Variable for the frequencies along the contour
13    std::complex<double> s_k;
14    for (int k = 0; k < M + 1; k++) {
15        // Integrate on circumference centered in (0,0) with radius r
16        s_k = delta(r * std::exp(2 * M_PI * imag * ((double)k / (double)(M + 1))) /
17            tau;
18        f[k] = F(s_k);
19    }
20    // Integrate with trapezoidal rule at M+1 points
21    Eigen::FFT<double> fft;
22    w = fft.fwd(f) / (M + 1);
23    for (int k = 0; k < M + 1; k++) {
24        // Rescale by the radius of the circle, which arise from the  $z^l$ 
25        // -factor
26        // in the integrand
27        w[k] = w[k] / std::pow(r, k);
28    }
29    return w.real();
30 }
```


HINT 1 for (3-3.d): From Eq. (3.3.2), at time $t = t_n$, we get

$$\mathbf{A}\vec{\mu}_n + \left(\sum_{\ell=0}^n w_{n-\ell}^{F,\tau} \mathbf{B}\vec{\mu}_\ell \right) (t) = \vec{\phi}_n .$$

From this equation we can successively determine $\vec{\mu}_1, \vec{\mu}_2, \dots, \vec{\mu}_n$.

SOLUTION of (3-3.d):

C++-code 3.3.6: Compute matrix A, see (3-3.a)

```

2 SparseMatrix<double> compute_matA( size_t N) {
3     double h = 1. / N;
4     VectorXd grid = VectorXd::LinSpaced(N + 1, 0., 1.);
5
6     SparseMatrix<double> A(N + 1, N + 1);
7     A.reserve(3 * N + 1); // 3(N+1) - 2
8     // Inserting endpoints.
9     A.insert(0, 0) =
10         1. / h + 1. / (h * h * pow(M_PI, 3)) *
11             ((pow(M_PI * h, 2) - 2.) + 2. * cos(M_PI * h)); // A(0,0)
12     A.insert(N, N) = 1. / h + 1. / (h * h * pow(M_PI, 3)) *
13         ((pow(M_PI * h, 2) - 2.) -
14             2. * cos(M_PI * (1 - h))); // A(N,N)
15
16     for (int i = 1; i <= N; ++i) {
17         if (i < N) {
18             // Inserting diagonal entries
19             A.insert(i, i) =
20                 2. / h + 2. / (h * h * pow(M_PI, 3)) *
21                     (2 * M_PI * h * sin(M_PI * grid(i)) +
22                         cos(M_PI * grid(i + 1)) - cos(M_PI * grid(i - 1)));
23         }
24         // Inserting sub- and super-diagonal entries
25         // A(i-1,i) = A(i,i-1)
26         A.insert(i - 1, i) = A.insert(i, i - 1) =
27             -1. / h +
28             1. / (h * h * pow(M_PI, 3)) *
29                 (2. * (cos(M_PI * grid(i - 1)) - cos(M_PI * grid(i))) -
30                     M_PI * h * (sin(M_PI * grid(i - 1)) + sin(M_PI * grid(i))));
31     }
32     return A;
33 }

```

Get it on  [GitLab \(absorbingboundarycondition.h\)](https://gitlab.com/absorbingboundarycondition/h).

C++-code 3.3.7: Solution of (3-3.d)

```

2 template <typename FUNC>
3 VectorXd solve_IBVP(const FUNC& g, size_t M, size_t N, double T) {
4     MatrixXcd u = MatrixXcd::Zero(N + 1, M + 1);
5     auto F = [](complex<double> s) { return log(s) / (s * s + 1.); };
6     // matrix A
7     SparseMatrix<double> A(N + 1, N + 1);
8     A = compute_matA(N);
9     // convolution weights
10    auto delta = [](std::complex<double> z) {
11        return 1.0 / 2.0 * z * z - 2.0 * z + 3.0 / 2.0;
12    };
13    double tau = T / M;
14    Eigen::VectorXd w = cqweights_by_dft(F, delta, tau, M);
15    // Aw <- A + lowToeplitz(w)*B; B(N,N) = 1, else B(i,j) = 0
16    SparseMatrix<complex<double> > Aw = A.cast<complex<double> >();

```

```

17 Aw.coeffRef(N, N) += w(0);
18 SparseLU<SparseMatrix<complex<double>>> solver;
19 solver.compute(Aw);
20 // For visualization
21 ofstream f_ref;
22 if (M == 4096 && N == 4096) f_ref.open(CURRENT_BINARY_DIR "/u_ref.txt");
23 // run solver from t=0 -> t=T
24 for (int i = 1; i <= M; ++i) {
25     // rhs_cq: from the convolution weights  $l = 0, 1, \dots, n-1$ 
26     complex<double> rhs_cq = 0.;
27     for (int l = 0; l < i; ++l) {
28         rhs_cq += w(i - l) * u(N, l);
29     }
30
31     // function  $\phi$ 
32     VectorXcd phi = VectorXcd::Zero(N + 1);
33     phi(0) = -(complex<double>)g(i * tau);
34
35     VectorXcd rhs = phi;
36     rhs(N) -= rhs_cq; // rhs
37     u.col(i) = solver.solve(rhs); // solution at  $t = t_n$ 
38 }
39 // For visualization
40 if (f_ref.is_open()) {
41     f_ref << u.real();
42     f_ref.close();
43 }
44 return u.col(M).real();
45 }

```

Get it on  [GitLab \(absorbingboundarycondition.h\)](#).

HINT 1 for (3-3.e): Using the given Galerkin discretization in spatial one dimension, we can write

$$\int_0^1 \left| \frac{du_{N,M}}{dx} \right|^2 dx = \sum_{i=1}^N \frac{(\mu_i - \mu_{i-1})^2}{h}$$

SOLUTION of (3-4.a):

A formula for w_j^τ is given in [Lecture → ??]; use [Lecture → ??] for $\mu = \frac{1}{2}$:

$$w_0^\tau = \tau^{-1/2} \quad , \quad w_\ell^\tau = \tau^{-1/2} (-1)^\ell \prod_{k=0}^{\ell-1} \frac{1/2 - k}{k + 1} \quad , \quad \ell \in \mathbb{N} . \quad (3.4.8)$$

C++ code 3.4.9: Code for `cqWeights()` →GITLAB

```
2 Eigen::VectorXd cqWeights(unsigned int M, double tau) {
3   Eigen::VectorXd w(M + 1);
4
5   w(0) = std::pow(tau, -0.5);
6   // Calculate weights of convolution quadrature based on Eq. (3.4.8)
7   for (int l = 1; l < M + 1; ++l) {
8     w(l) = w(l - 1) * (-1) * (0.5 - (l - 1)) / l; // denominator is factorial
9   }
10  return w;
11 }
```

SOLUTION of (3-4.b):

We can recast (3.4.7) as

$$\begin{aligned}(w_0^\tau \mathbf{M} + \mathbf{A}) \vec{\mu}_n &= \vec{\varphi}(\tau n) - \sum_{\ell=0}^{n-1} w_{n-\ell}^\tau \mathbf{M} \vec{\mu}_\ell \\ &= \vec{\varphi}(\tau n) - h^2 \sum_{\ell=0}^{n-1} w_{n-\ell}^\tau \vec{\mu}_\ell, \quad n = 0, \dots, M.\end{aligned}\tag{3.4.10}$$

We successively solve for $\vec{\mu}_0, \vec{\mu}_1, \vec{\mu}_2, \dots$ by solving an $N \times N$ linear system of equations with coefficient matrix $w_0^\tau \mathbf{M} + \mathbf{A}$.

C++ code 3.4.11: Creates the sequence of position vectors for the grid nodes. →GITLAB

```
2 namespace FractionalHeatEquation {
3   std::vector<Eigen::Vector2d> generateGrid(unsigned n) {
4     std::vector<Eigen::Vector2d> gridpoints{n * n, Eigen::Vector2d()};
5     double h = 1.0 / (n + 1);
6     int rown = 0;
7     for (int yind = 1; yind < n + 1; yind++) {
8       for (int xind = 1; xind < n + 1; xind++, rown++) {
9         Eigen::Vector2d tmp;
10        tmp << xind * h, yind * h;
11        gridpoints[rown] = tmp;
12      }
13    }
14    return gridpoints;
15  }
```

C++ code 3.4.12: MOT solving of (3.4.7) →GITLAB

```
2 template <typename SOURCEFN,
3           typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
4   Eigen::VectorXd evlMOT(
5     SOURCEFN &&f, unsigned int n, double T, unsigned int M,
6     RECORDER rec = []((const Eigen::VectorXd &mu_n) {})) {
7   const unsigned int N = n * n; // Number of FE d.o.f.s
8   // Vector storing all the states; big memory consumption
9   std::vector<Eigen::VectorXd> mu_vecs{M + 1, Eigen::VectorXd(N)};
10  double tau = T * 1.0 / M; // timestep size
11  double h = 1.0 / (n + 1); // meshwidth
12  // See Sub-problem (3-4.a)
13  Eigen::VectorXd w = cqWeights(M, tau);
14  // Initialise matrix to invert at every timestep, gridpoints and rhs
15  SqrtsMplusA w0MplusA(n, std::complex<double>(std::pow(w[0], 2), 0.0));
16  std::vector<Eigen::Vector2d> gridpoints = generateGrid(n);
17  Eigen::VectorXd rhs(N);
18  for (int time_ind = 0; time_ind < M + 1; time_ind++) {
19    // Evaluate rhs
20    for (int space_ind = 0; space_ind < N; space_ind++) {
21      rhs[space_ind] = f(time_ind * tau, gridpoints[space_ind]);
22    }
23    // Add memory tail from convolution
24    for (int l = 0; l < time_ind; l++) {
```

```
25     rhs += -w[time_ind - 1] * mu_vecs[1];
26 }
27 rhs *= h * h;
28 // Next timestep according to (??)
29 mu_vecs[time_ind] =
30     w0MplusA.solve(rhs).real(); //TODO: Check if this is correct
31 std::cout << w0MplusA.solve(rhs).real() << std::endl;
32 rec(mu_vecs[time_ind]);
33 }
34 return mu_vecs.back();
35 }
```

SOLUTION of (3-4.c):

Assuming an efficient implementation,

- a single LU decomposition of $w_0^T \mathbf{M} + \mathbf{A}$ has to be done in the beginning (the `solve()` method of **SqrtsMplusA** does this.),
 - $2(M + 1)$ triangular eliminations occur during M timesteps (2 per timestep, because we have to do forward and backward elimination),
 - no matrix \times vector product is needed, because \mathbf{M} is just a multiple of the identity matrix \mathbf{I}_N .
-

HINT 1 for (3-4.d): Look up Problem 3-1, [[NumCSE course → ??](#)], and [[NumCSE course → ??](#)].

SOLUTION of (3-4.d):

The block-triangular linear system of equations (3.4.7) reads for $M = 7$:

$$\left[\begin{array}{cccc|cccc} \mathbf{A}' & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 & 0 & 0 & 0 & 0 & 0 \\ h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 & 0 & 0 & 0 & 0 \\ h^2 w_3^\tau \mathbf{I}_N & h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 & 0 & 0 & 0 \\ \hline h^2 w_4^\tau \mathbf{I}_N & h^2 w_3^\tau \mathbf{I}_N & h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 & 0 & 0 \\ h^2 w_5^\tau \mathbf{I}_N & h^2 w_4^\tau \mathbf{I}_N & h^2 w_3^\tau \mathbf{I}_N & h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 & 0 \\ h^2 w_6^\tau \mathbf{I}_N & h^2 w_5^\tau \mathbf{I}_N & h^2 w_4^\tau \mathbf{I}_N & h^2 w_3^\tau \mathbf{I}_N & h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' & 0 \\ h^2 w_7^\tau \mathbf{I}_N & h^2 w_6^\tau \mathbf{I}_N & h^2 w_5^\tau \mathbf{I}_N & h^2 w_4^\tau \mathbf{I}_N & h^2 w_3^\tau \mathbf{I}_N & h^2 w_2^\tau \mathbf{I}_N & h^2 w_1^\tau \mathbf{I}_N & \mathbf{A}' \end{array} \right] \begin{bmatrix} \vec{\mu}_0 \\ \vec{\mu}_1 \\ \vec{\mu}_2 \\ \vec{\mu}_3 \\ \vec{\mu}_4 \\ \vec{\mu}_5 \\ \vec{\mu}_6 \\ \vec{\mu}_7 \end{bmatrix} = \begin{bmatrix} \vec{\phi}_0 \\ \vec{\phi}_1 \\ \vec{\phi}_2 \\ \vec{\phi}_3 \\ \vec{\phi}_4 \\ \vec{\phi}_5 \\ \vec{\phi}_6 \\ \vec{\phi}_7 \end{bmatrix},$$

where we abbreviated $\mathbf{A}' := h^2 w_0^\tau \mathbf{I}_N + \mathbf{A}$ and $\vec{\phi}_j := \vec{\phi}(\tau j)$. The separator lines indicate the recursive structure underlying the algorithm of [Lecture \rightarrow ??]. If we designate by $\mathbf{S}_L \in \mathbb{R}^{2^L N, 2^L N}$ the system matrix of the linear system (3.4.7) for $M = 2^L$, then

$$\mathbf{S}_L = \left[\begin{array}{c|c} \mathbf{S}_{L-1} & \mathbf{O} \\ \mathbf{T}_L \otimes \mathbf{I}_N & \mathbf{S}_{L-1} \end{array} \right], \quad (3.4.13)$$

with \otimes the Kronecker product and a Toeplitz matrix [Lecture \rightarrow ??]

$$\mathbf{T}_L := \left[h^2 w_{(i-j)+2^{L-1}}^\tau \right]_{i,j=0}^{2^{L-1}} \in \mathbb{R}^{2^{L-1}, 2^{L-1}}. \quad (3.4.14)$$

Recall the recursive algorithm:

$$\left[\begin{array}{c|c} \mathbf{S}_{L-1} & \mathbf{O} \\ \mathbf{T}_L \otimes \mathbf{I}_N & \mathbf{S}_{L-1} \end{array} \right] \begin{bmatrix} \vec{\xi}_1 \\ \vec{\xi}_2 \end{bmatrix} = \begin{bmatrix} \vec{\eta}_1 \\ \vec{\eta}_2 \end{bmatrix} \quad \blacktriangleright \quad \begin{aligned} \vec{\xi}_1 &= \mathbf{S}_{L-1}^{-1} \vec{\eta}_1, \\ \vec{\beta} &:= \vec{\eta}_2 - (\mathbf{T}_L \otimes \mathbf{I}_N) \vec{\xi}_1, \\ \vec{\xi}_2 &= \mathbf{S}_{L-1}^{-1} \vec{\beta}. \end{aligned}$$

The key operation is the multiplication of an $2^{L-1}N$ -vector with the block-Toeplitz matrix $\mathbf{T}_L \otimes \mathbf{I}_N$. To implement this efficiently, we first make use of the trick from [NumCSE course \rightarrow ??], [NumCSE course \rightarrow ??], which reduces the task to N multiplications with the Toeplitz matrix \mathbf{T}_L given in (3.4.14). To carry out those efficiently, it is advisable to introduce a dedicated class for multiplying a Toeplitz matrix with many vectors efficiently.

C++ code 3.4.15: Helper class **ToeplitzOp** \rightarrow **GITLAB**

```

2 class ToeplitzOp {
3 public:
4     explicit ToeplitzOp(const Eigen::VectorXd &);
5     [[nodiscard]] Eigen::VectorXd eval(const Eigen::VectorXd &);
6
7 private:
8     Eigen::VectorXcd u_;
9     Eigen::VectorXcd tmp_;
10    Eigen::FFT<double> fft_;
11
12 public:

```

```

13 static unsigned int init_cnt;
14 static unsigned int eval_cnt;
15 };

```

For the efficient implementation of the `eval()` member function we employ FFT. Refer to [NumCSE course → ??] and Problem 3-1 for details. For the sake of efficiency it is important to precompute the DFT of the circulant extension of the Toeplitz matrix in the constructor.

C++ code 3.4.16: Cconstructor for helper class `ToeplitzOp` →GITLAB

```

2 ToeplitzOp::ToeplitzOp(const Eigen::VectorXd &v)
3   : u_(Eigen::VectorXcd::Zero(v.size() + 1)),
4     tmp_(Eigen::VectorXcd::Zero(v.size() + 1)) {
5   const unsigned int n = v.size() + 1;
6   assertm(n > 1, "Vector v missing");
7   assertm((n % 2) == 0,
8           "Only odd-length vectors can define square Toeplitz matrices");
9   // Initialize vector defining circulant extension of Toeplitz matrix
10  tmp_.head(n / 2) =
11      (v.segment((n - 2) / 2, n / 2)).template cast<std::complex<double>>();
12  tmp_[n / 2] = std::complex<double>(0.0, 0.0);
13  tmp_.tail((n - 2) / 2) =
14      (v.head((n - 2) / 2)).template cast<std::complex<double>>();
15  // Store DFT of circulant-defining vector
16  u_ = fft_.fwd(tmp_);
17  init_cnt++;
18 }

```

C++ code 3.4.17: Implementation of `eval()` for helper class `ToeplitzOp` →GITLAB

```

2 Eigen::VectorXd ToeplitzOp::eval(const Eigen::VectorXd &x) {
3   const unsigned int d = x.size();
4   const unsigned int m = u_.size();
5   assertm(d == m / 2, "Vector length mismatch");
6   eval_cnt++;
7   tmp_.head(d) = x.template cast<std::complex<double>>();
8   tmp_.tail(d) = Eigen::VectorXcd::Zero(d);
9   return (fft_.inv(u_.cwiseProduct(fft_.fwd(tmp_)).eval()).real()).head(d);
10 }

```

C++ code 3.4.18: Code for `evlTriangToeplitz()` →GITLAB

```

2 template <typename SOURCEFN,
3          typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
4 Eigen::VectorXd evlTriangToeplitz(
5   SOURCEFN &&f, unsigned int n, double T, unsigned int L,
6   RECORDER rec = [] (const Eigen::Vector2d &mu_n) {} ) {
7   const unsigned int N = n * n;
8   const unsigned int M = std::pow(2, L) - 1;
9   Eigen::MatrixXcd mu_vecs(N, M + 1);
10  const double tau = T * 1.0 / M;
11  const double h = 1.0 / (n + 1);
12  Eigen::VectorXd cq_weights = cqWeights(M, tau);

```

```

13 // Initialise matrix to invert at every timestep, gridpoints and rhs
14 SqrtsMplusA A(n, std::complex<double>(std::pow(cq_weights[0], 2), 0.0));
15 //Generate rhs vector
16 std::vector<Eigen::Vector2d> gridpoints = generateGrid(n);
17 Eigen::MatrixXd rhs(N, (M + 1));
18 for (int time_ind = 0; time_ind < M + 1; time_ind++) {
19     // Evaluate rhs
20     for (int space_ind = 0; space_ind < N; space_ind++) {
21         rhs(space_ind, time_ind) =
22             h * h * f(time_ind * tau, gridpoints[space_ind]);
23     }
24 }
25 std::function<Eigen::MatrixXd(unsigned, Eigen::VectorXd, Eigen::MatrixXd,
26     Eigen::MatrixXd>
27     recursive_solve = [&](unsigned L, Eigen::VectorXd weights,
28     Eigen::MatrixXd mu, Eigen::MatrixXd phi) {
29     assert(mu.size() == phi.size());
30     if (L == 0) {
31         const Eigen::VectorXcd phi_comp =
32             phi.template cast<std::complex<double>>();
33         mu = (A.solve(phi_comp)).real();
34     } else {
35         const unsigned local_M = mu.cols();
36         //Solve upper left part recursively
37         mu.leftCols(local_M / 2) = recursive_solve(
38             L - 1, weights.head(local_M / 2), mu.leftCols(local_M / 2),
39             phi.leftCols(local_M / 2));
40         //Solve lower left part with fft
41         ToeplitzOp T(h * h * weights.tail(local_M - 1));
42         for (unsigned l = 0; l < N; ++l) {
43             phi.row(l).tail(local_M / 2) =
44                 phi.row(l).tail(local_M / 2) -
45                 T.eval(mu.row(l).head(local_M / 2)).transpose();
46         }
47         //Solve lower right part recursively
48         mu.rightCols(local_M / 2) = recursive_solve(
49             L - 1, weights.head(local_M / 2), mu.rightCols(local_M / 2),
50             phi.rightCols(local_M / 2));
51     }
52     return mu;
53 };
54 mu_vecs = recursive_solve(L, cq_weights, mu_vecs, rhs);
55 return mu_vecs.col(M);
56 }

```

SOLUTION of (3-4.e):

C++ code 3.4.22: Code for `ev1ASAOCQ()` →GITLAB

```

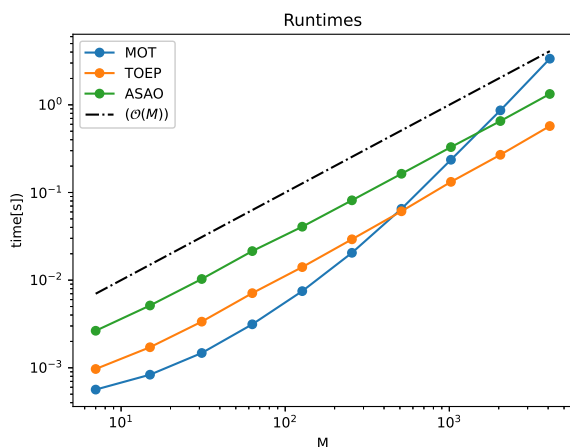
2  template <typename SOURCEFN,
3      typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
4  Eigen::VectorXd ev1ASAOCQ(
5      SOURCEFN &&f, unsigned int n, double T, unsigned int L,
6      RECORDER rec = [] (const Eigen::VectorXd &mu_n) {} ) {
7      const unsigned int N = n * n;
8      const unsigned int M = std::pow(2, L) - 1;
9      double tau = T * 1.0 / M;
10     double h = 1.0 / (n + 1.0);
11     auto delta = [] (std::complex<double> z) {
12         return 1.0 - z;
13         //return 1.0 / 2.0 * z * z - 2.0 * z + 3.0 / 2.0;
14     };
15     // Initialize the numerical solution. This implementation is, for the
16     // sake of clarity, not memory-efficient.
17     Eigen::MatrixXd mu_vecs(N, M + 1);
18     // Initialise array for the whole right hand side (all timepoints)
19     Eigen::MatrixXd phi(N, M + 1);
20     // Initialise array for the right hand side at a single timepoint
21     Eigen::VectorXd phi_slice(N); //TODO: Check this
22     // Set radius of integral contour
23     double r = std::pow(10, -16.0 / (2 * M + 2));
24     // Set gridpoints
25     std::vector<Eigen::Vector2d> gridpoints = generateGrid(n);
26     for (int time_ind = 0; time_ind < M + 1; time_ind++) {
27         // Evaluate rhs
28         for (int space_ind = 0; space_ind < N; space_ind++) {
29             phi_slice[space_ind] = f(time_ind * tau, gridpoints[space_ind]);
30         }
31         phi.col(time_ind) = std::pow(r, time_ind) * h * h * phi_slice;
32     }
33     // Transform the right-hand side from the time domain into the
34     // frequency domain
35     Eigen::MatrixXcd phi_hat(N, M + 1);
36     Eigen::FFT<double> fft;
37     for (int space_ind = 0; space_ind < N; space_ind++) {
38         Eigen::VectorXcd in =
39             phi.row(space_ind).template cast<std::complex<double>>();
40         Eigen::VectorXcd out(M + 1);
41         out = fft.fwd(in);
42         phi_hat.row(space_ind) = out;
43     }
44     // Initializing the frequency domain numerical solution
45     Eigen::MatrixXcd mu_hat(N, M + 1);
46     // Complex variable containing the imaginary unit and the discrete
47     // frequencies
48     std::complex<double> s_l;
49     std::complex<double> imag(0, 1);
50     for (int freq_ind = 0; freq_ind < M + 1; freq_ind++) {
51         s_l = delta(r * std::exp(-2 * M_PI * imag * ((double)freq_ind) /
52             (double)(M + 1))) /
53             tau;
54         // Applying the time-harmonic operator  $G(s_l)^{-1} = (\sqrt{s_l}M + A)^{-1}$ 

```

```
52 SqrtsMplusA sIMplusA(n, s_l);
53 Eigen::VectorXcd phi_hat_slice(N);
54 Eigen::VectorXcd mu_hat_slice(N);
55 phi_hat_slice = phi_hat.col(freq_ind);
56 mu_hat_slice = sIMplusA.solve(phi_hat_slice);
57 mu_hat.col(freq_ind) = mu_hat_slice;
58 }
59 // Transform the numerical solution from the frequency domain to the
   time domain
60 for (int space_ind = 0; space_ind < N; space_ind++) {
61     Eigen::VectorXcd in = mu_hat.row(space_ind);
62     Eigen::VectorXcd out(N);
63     out = fft.inv(in);
64     mu_vecs.row(space_ind) = out.real();
65 }
66 // Rescaling of the numerical solution
67 for (int time_ind = 0; time_ind < M + 1; time_ind++) {
68     mu_vecs.col(time_ind) *= std::pow(r, -time_ind);
69 }
70 return mu_vecs.col(M);
71 }
```

SOLUTION of (3-4.f):

Below we plot the runtimes of the different implementations, for various M and a very coarse grid, defined by $n = 4$. For small M , the standard time stepping implementation `evlMOT()` is competitive. The quadratic growth of the runtime M causes the method to be outperformed for larger M . The main effort of the `evlASAOQ()` is dominated by amount of necessary LU-decompositions, which grows linearly in M . As a consequence, the runtime is larger than the other two methods, but asymptotically almost linear. The most performant method for our specific example is the `evlTriangToeplitz` implementation, which has no quadratic runtime and only requires the computation of a single LU-decomposition.



CPU :
 OS :
 Compiler:
 Options :

C++ code 3.4.23: Code for `main` → [GITLAB](#)

```

2 int main(int /*argc*/, char** /*argv*/) {
3     FractionalHeatEquation::SqrtsMplusA Amat(2, std::complex<double>(1, 1));
4     std::cout << "Running code for ADVNCE HW FractionalHeatEquation"
5               << std::endl;
6     const double T = 1.0;
7     const int L_start = 4;
8     const int L_max = 13;
9     const int n = 4;
10    std::function<double(double, Eigen::Vector2d)> f =
11        [](double t, Eigen::Vector2d x) { return t * t * t; };
12
13    const unsigned num_repetitions = 5;
14    std::ofstream out(CURRENT_SOURCE_DIR "/runtimes.csv");
15
16    std::cout << std::left << std::setw(10) << "M" << std::left << std::setw(15)
17              << "time_MOT" << std::left << std::setw(15) << "time_TOEP"
18              << std::left << std::setw(15) << "time_ASAO" << std::left
19              << std::setw(15) << std::endl;
20    out << "M"
21        << ", "
22        << "time_MOT"
23        << ", "
24        << "time_TOEP"
25        << ", "
26        << "time_ASAO" << std::endl;
27    for (unsigned int L = L_start; L < L_max; ++L) {
28        double time_mot = std::numeric_limits<double>::max();
29        double time_toep = std::numeric_limits<double>::max();

```

```

30 double time_asao = std::numeric_limits<double>::max();
31 for (int k = 0; k < num_repetitions; k++) {
32     // Use C++ chrono library to measure runtimes
33     auto t1 = std::chrono::high_resolution_clock::now();
34     Eigen::VectorXd mu_MOT =
35         FractionalHeatEquation::evIMOT(f, n, T, std::pow(2, L) - 1);
36     auto t2 = std::chrono::high_resolution_clock::now();
37     // Getting number of seconds as a double
38     std::chrono::duration<double> ms_mot = (t2 - t1);
39
40     time_mot = std::min(time_mot, ms_mot.count());
41
42     t1 = std::chrono::high_resolution_clock::now();
43     Eigen::VectorXd mu_Toep =
44         FractionalHeatEquation::evITriangToeplitz(f, n, T, L);
45     t2 = std::chrono::high_resolution_clock::now();
46     std::chrono::duration<double> ms_toep = (t2 - t1);
47
48     // Taking the minimal measured time as the result
49     time_toep = std::min(time_toep, ms_toep.count());
50
51     t1 = std::chrono::high_resolution_clock::now();
52     Eigen::VectorXd mu_AS AO = FractionalHeatEquation::evIASAOQ(f, n, T, L);
53
54     t2 = std::chrono::high_resolution_clock::now();
55     // Getting number of seconds as a double
56     std::chrono::duration<double> ms_asao = (t2 - t1);
57
58     time_asao = std::min(time_asao, ms_asao.count());
59 }
60 std::cout << std::left << std::setw(10) << std::pow(2, L) - 1 << std::left
61     << std::setw(15) << time_mot << std::left << std::setw(15)
62     << time_toep << std::left << std::setw(15) << time_asao
63     << std::left << std::setw(15) << std::endl;
64 out << std::pow(2, L) - 1 << "," << time_mot << "," << time_toep << ","
65     << time_asao << std::endl;
66 }
67 out.close();
68 // Call python script
69 std::system("python3 " CURRENT_SOURCE_DIR "/plot.py " CURRENT_SOURCE_DIR
70     "/runtimes.csv " CURRENT_SOURCE_DIR "/runtimes.png");
71 return 0;
72 }

```


SOLUTION of (4-1.a):

C++ code 4.1.1: Construction of Poisson matrix.

Get it on  [GitLab \(stationarylineariterations.cpp\)](#).

```
2 using triplet = Eigen::Triplet<double>;
3 Eigen::SparseMatrix<double> poissonMatrix(unsigned int n) {
4     const int N = (n - 1) * (n - 1); // Matrix size
5     // define vector of triplets and reserve memory
6     std::vector<triplet> entries; // For temporary COO format
7     int start_id;
8     int end_id;
9     // set the vector of triplets
10
11     for (int block_id = 0; block_id < (n - 1); block_id++) {
12         start_id = block_id * (n - 1);
13         end_id = (block_id + 1) * (n - 1) - 1;
14         // tri-diagonal matrix T
15         for (int i = start_id; i <= end_id; i++) {
16             entries.emplace_back(i, i, 4); // Diagonal entry
17             if (i > start_id) { //  $T(i-1,i) = T(i,i-1)$ 
18                 entries.emplace_back(i, i - 1, -1.0);
19                 entries.emplace_back(i - 1, i, -1.0);
20             }
21         }
22         // identity blocks
23         if (block_id > 0) {
24             for (int i = start_id; i <= end_id; i++) {
25                 entries.emplace_back(i, i - (n - 1), -1.0);
26                 entries.emplace_back(i - (n - 1), i, -1.0);
27             }
28         }
29     }
30     // create the sparse matrix in CRS format
31     Eigen::SparseMatrix<double> A(N, N);
32     A.setFromTriplets(entries.begin(), entries.end());
33     A.makeCompressed();
34     return A;
35 }
```

HINT 1 for (4-1.b): Make use of the known eigenvectors/eigenvalues of the Poisson matrix, see [Lecture
→ Eq. (4.1.3.24)].

SOLUTION of (4-1.b):

The error propagation matrix for the Jacobi method is

$$\mathbf{E}_J := I_N - \omega(\text{diag } \mathbf{A})^{-1} \mathbf{A} . \quad (4.1.3)$$

We have

$$\text{diag}(\mathbf{A}_n) = 4I_N \implies \text{diag}(\mathbf{A})^{-1} = \frac{1}{4}I_N .$$

Let $\vec{v} \in \mathbb{R}^N, \|\vec{v}\| = 1$ be an eigenvector corresponding to the eigenvalue $\lambda(\mathbf{A}_n)$ of the Poisson matrix \mathbf{A}_n . Then

$$\mathbf{E}_J \vec{v} = \left(I - \frac{1}{4} \omega \mathbf{A} \right) \vec{v} = \left(1 - \frac{1}{4} \omega \lambda(\mathbf{A}) \right) \vec{v} ,$$

that is, \vec{v} is also an eigenvector of the error propagation matrix. Since the symmetric matrix \mathbf{A}_n admits a basis of \mathbb{R}^N of orthogonal eigenvectors, those also diagonalize \mathbf{E}_J and the eigenvalues of \mathbf{E}_J can directly be inferred from the eigenvalues of \mathbf{A}_n .

$$\kappa \in \sigma(\mathbf{E}_J) \iff \kappa = \left(1 - \frac{1}{4} \omega \lambda \right) , \quad \lambda \in \sigma(\mathbf{A}_n) , \quad (4.1.4)$$

where $\sigma(\mathbf{M})$ denotes the **spectrum** of the square matrix \mathbf{M} .

Hence, for a fixed relaxation parameter ω , the maximal (in modulus) eigenvalue of \mathbf{E}_J is given by

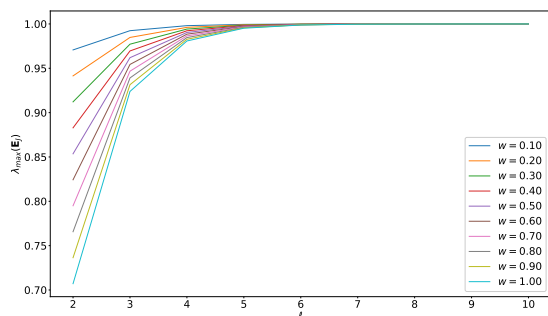
$$\max\{|\kappa|, \kappa \in \sigma(\mathbf{E}_J)\} = \max\left\{1 - \frac{1}{4} \omega \lambda_{\min}(\mathbf{A}_n), \frac{1}{4} \omega \lambda_{\max}(\mathbf{A}_n) - 1\right\} . \quad (4.1.5)$$

From [Lecture \rightarrow Eq. (4.1.3.24)] we learn that

$$\lambda_{\min}(\mathbf{A}_n) = 8 \sin^2\left(\frac{\pi}{2n}\right) , \quad \lambda_{\max}(\mathbf{A}_n) = 8 \sin^2\left((n-1)\frac{\pi}{2n}\right) , \quad (4.1.6)$$

which implies, for $\omega \in [0, 1]$,

$$\max\{|\kappa|, \kappa \in \sigma(\mathbf{E}_J)\} = 1 - 2\omega \sin^2\left(\frac{\pi}{2n}\right) . \quad (4.1.7)$$



We show the variation of the asymptotic rate of convergence with respect to n , for different ω values, in the plot given beside.

For $n \rightarrow \infty$ the rate of linear convergence approaches 1, that is, asymptotically, the method will converge arbitrarily slowly.

SOLUTION of (4-1.c):

We draw on (??) and have to find the minimize of the function

$$\omega \mapsto \max\{|\kappa|, \kappa \in \sigma(\mathbf{E}_J)\} = \max\left\{1 - \frac{1}{4}\omega\lambda_{\min}(\mathbf{A}_n), \frac{1}{4}\omega\lambda_{\max}(\mathbf{A}_n) - 1\right\}.$$

That function is just the maximum of two linear functions, one increasing, the other decreasing. Thus the minimizer $\omega_{\text{opt}} > 0$ of that maximum is that ω , where both linear functions intersect:

$$1 - \frac{1}{4}\omega_{\text{opt}}\lambda_{\min}(\mathbf{A}_n) = \frac{1}{4}\omega_{\text{opt}}\lambda_{\max}(\mathbf{A}_n) - 1,$$

and we find

$$\omega_{\text{opt}} = \frac{2}{\lambda_{\min}(\mathbf{A}_n) + \lambda_{\max}(\mathbf{A}_n)}. \quad (4.1.8)$$

The optimal rate of convergence of the Jacobi method is

$$\|\mathbf{E}_J(\omega_{\text{opt}})\| = 1 - \frac{1}{4} \cdot \frac{\lambda_{\min}(\mathbf{A}_n)}{\lambda_{\min}(\mathbf{A}_n) + \lambda_{\max}(\mathbf{A}_n)} = 1 - \frac{1}{4} \cdot \frac{1}{1 + \kappa(\mathbf{A}_n)}, \quad (4.1.9)$$

where $\kappa(\mathbf{A})$ is the spectral condition number of \mathbf{A}_n . We still conclude the asymptotic behavior

$$\|\mathbf{E}_J(\omega_{\text{opt}})\| = 1 - O(n^{-2}) \quad \text{for } n \rightarrow \infty. \quad (4.1.10)$$

Disappointingly, choosing the optimal relaxation parameter ω does not alleviate the asymptotic deterioration of the rate of convergence.

HINT 1 for (4-1.d): You can use the **triangularView** matrix adaptor member function of **Eigen::Matrix**.

└

SOLUTION of (4-1.d):

C++ code 4.1.11: Construction of Poisson matrix. [Get it on 🐙 GitLab \(stationarylineariterations.](#)

```
2 void gaussSeidel(const Eigen::SparseMatrix<double> &A,
3                 const Eigen::VectorXd &phi, Eigen::VectorXd &mu, int maxltr,
4                 double TOL) {
5     Eigen::VectorXd delta(A.rows());
6     int iter = 0;
7     do {
8         // Rely on Eigen's ability to solve a sparse triangular system
9         // efficiently
10        delta = A.triangularView<Eigen::Lower>().solve(
11            phi - A * mu); // Triangular solve
12        mu += delta;
13        ++iter;
14    } while (delta.norm() > TOL * mu.norm() && iter < maxltr);
15    if (maxltr == iter) {
16        std::cerr << "Did not converge to tol: " << TOL
17            << "in maximal number of iterations: " << iter << std::endl;
18    }
19 }
```

This implementation is a wager on EIGEN's prowess in optimizing the solution of a sparse triangular system. We have assumed that the asymptotic cost of Line 10 is $O(\text{nnz}(\mathbf{A}))$ for $N \rightarrow \infty$, N the matrix dimension.

HINT 1 for (4-1.e): Study [Lecture \rightarrow Rem. 4.1.3.18] to learn how one can measure asymptotic rates of convergence by means of the power iteration method. ┘

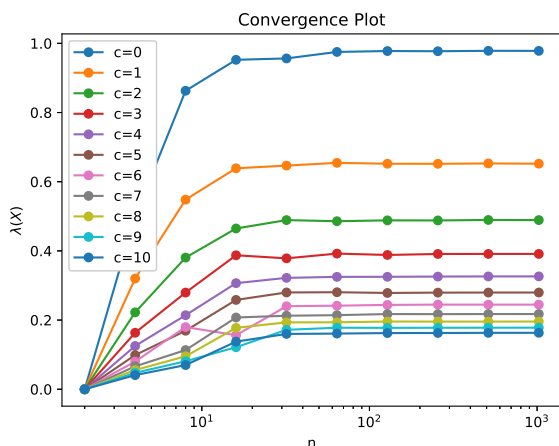
SOLUTION of (4-1.e):

C++ code 4.1.12: Computation of asymptotic rate of convergence of Gauss-Seidel iteration Get it on [GitLab \(stationarylineariterations.cpp\)](#).

```
2 double comp_lmax_gaussSeidel(const Eigen::SparseMatrix<double> &X,  
3                             double TOL = 1.0E-03) {  
4     const int N = X.rows();  
5  
6     Eigen::VectorXd v = Eigen::VectorXd::Random(N);  
7     double lambda_new = 0;  
8     double lambda_old = 1;  
9  
10    // Power iteration  
11    do {  
12        lambda_old = lambda_new;  
13        v /= v.norm();  
14        v -= X.triangularView<Eigen::Lower>().solve(X * v);  
15        lambda_new = v.norm();  
16    } while (std::abs(lambda_new - lambda_old) > TOL * lambda_new);  
17    return lambda_new;  
18 }
```

C++ code 4.1.13: Implementation of gaussSeidelRate() Get it on [GitLab \(stationarylineariterations.cpp\)](#).

```
2 double gaussSeidelRate(unsigned int n, double c, double TOL) {  
3     const unsigned int N = (n - 1) * (n - 1);  
4  
5     Eigen::SparseMatrix<double> X = poissonMatrix(n);  
6     Eigen::SparseMatrix<double> I(N, N);  
7     I.setIdentity();  
8     X += c * I;  
9     const double lambda_max = comp_lmax_gaussSeidel(X, TOL);  
10    return lambda_max;  
11 }
```



We show the variation of the asymptotic rate of convergence with respect to n , for different c values, in the plot given beside.

HINT 1 for (4-1.f): Also write $\|\cdot\|_A$ for the matrix norm induced by the energy norm. For $\mathbf{M} \in \mathbb{R}^{N,N}$ we have

$$\|\mathbf{M}\|_A = \left\| \mathbf{A}^{\frac{1}{2}} \mathbf{M} \mathbf{A}^{-\frac{1}{2}} \right\|_2, \quad (4.1.16)$$

where $\mathbf{A}^{\pm\frac{1}{2}} \in \mathbb{R}^{N,N}$ is the unique s.p.d. matrix whose square is $\mathbf{A}^{\pm 1}$, and $\|\cdot\|_2$ stands for the Euclidean matrix norm. ┘

HINT 2 for (4-1.f): Using (4.1.16) show that

$$q := \left\| \mathbf{I} - \text{triu}(\mathbf{A})^{-1} \mathbf{A} \right\|_A < 1. \quad (4.1.17)$$

┘

HINT 3 for (4-1.f): From [NumCSE course → Lemma 3.4.4.4] we know that

$$\|\mathbf{M}\|_2^2 = \lambda_{\max}(\mathbf{M}^\top \mathbf{M}) \quad \forall \mathbf{M} \in \mathbb{R}^{N,N}. \quad (4.1.18)$$

┘

SOLUTION of (4-1.f):

The identity (4.1.16) is a consequence of

$$\begin{aligned} \|\mathbf{M}\|_A^2 &= \sup_{\vec{\eta} \in \mathbb{R}^{N \setminus \{0\}}} \frac{\|\mathbf{M}\vec{\eta}\|_A^2}{\|\vec{\eta}\|_A^2} = \sup_{\vec{\eta} \in \mathbb{R}^{N \setminus \{0\}}} \frac{\vec{\eta}^\top \mathbf{M}^\top \mathbf{A} \mathbf{M} \vec{\eta}}{\vec{\eta}^\top \mathbf{A} \vec{\eta}} \\ &= \sup_{\vec{\zeta} \in \mathbb{R}^{N \setminus \{0\}}} \frac{\vec{\zeta}^\top \mathbf{A}^{-\frac{1}{2}} \mathbf{M}^\top \mathbf{A} \mathbf{M} \mathbf{A}^{-\frac{1}{2}} \vec{\zeta}}{\vec{\zeta}^\top \vec{\zeta}} = \left\| \mathbf{A}^{\frac{1}{2}} \mathbf{M} \mathbf{A}^{-\frac{1}{2}} \right\|_2^2. \end{aligned}$$

Hence we have

$$\left\| \mathbf{I} - \text{tril}(\mathbf{A})^{-1} \mathbf{A} \right\|_A = \left\| \mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right\|_2.$$

We examine the matrix in the $\|\cdot\|_2$ -norm. We remember that $\mathbf{A} = \mathbf{A}^\top$ and make smart use of associativity:

$$\begin{aligned} & \left(\mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right)^\top \left(\mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right) \\ &= \mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \mathbf{A}^{\frac{1}{2}} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} + \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \mathbf{A} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \\ &= \mathbf{I} - \mathbf{A}^{\frac{1}{2}} \left(\text{tril}(\mathbf{A})^{-\top} + \text{tril}(\mathbf{A})^{-1} - \text{tril}(\mathbf{A})^{-\top} \mathbf{A} \text{tril}(\mathbf{A})^{-1} \right) \mathbf{A}^{\frac{1}{2}} \\ &= \mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \left(\text{tril}(\mathbf{A}) + \text{tril}(\mathbf{A})^\top - \mathbf{A} \right) \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \\ &= \mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \text{diag}(\mathbf{A}) \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}}. \end{aligned}$$

Since the matrix $\mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \text{diag}(\mathbf{A}) \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}}$ is s.p.d. (congruent to the s.p.d. matrix $\text{diag}(\mathbf{A})$!), we infer

$$\begin{aligned} q^2 &:= \lambda_{\max} \left(\left(\mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right)^\top \left(\mathbf{I} - \mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right) \right) \\ &= 1 - \lambda_{\min} \left(\mathbf{A}^{\frac{1}{2}} \text{tril}(\mathbf{A})^{-\top} \text{diag}(\mathbf{A}) \text{tril}(\mathbf{A})^{-1} \mathbf{A}^{\frac{1}{2}} \right) < 1, \end{aligned}$$

which, thanks to (??), is the assertion to be proved.

SOLUTION of (4-2.a):

Refer to [Lecture → Ex. 4.1.1.24] and, in particular, [Lecture → Eq. (4.1.1.28)], from which we can read off the element matrix for $-\Delta$:

$$\mathbf{A}_K^{-\Delta} = \begin{bmatrix} 1 & -1/2 & 0 & -1/2 \\ -1/2 & 1 & -1/2 & 0 \\ 0 & -1/2 & 1 & -1/2 \\ -1/2 & 0 & -1/2 & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 4}.$$

Note that the local shape functions form a cardinal basis of the tensor-product polynomial space $\mathcal{TP}_1(\mathbb{R}^2)$ with respect to the vertices of K (as interpolation nodes). Since those vertices agree with the quadrature nodes the contribution to the element matrix from the zero-order term cu will be the diagonal matrix with constant diagonal $\frac{1}{3}h^2c$. This yields the final element matrix

$$\mathbf{A}_K = \begin{bmatrix} 1 + \frac{1}{4}h^2c & -1/2 & 0 & -1/2 \\ -1/2 & 1 + \frac{1}{4}h^2c & -1/2 & 0 \\ 0 & -1/2 & 1 + \frac{1}{4}h^2c & -1/2 \\ -1/2 & 0 & -1/2 & 1 + \frac{1}{4}h^2c \end{bmatrix} \in \mathbb{R}^{4 \times 4}. \quad (4.2.2)$$

SOLUTION of (4-2.j):

On every level $\ell \in \{2, 3, \dots, L\}$ we need two grid functions (one for the correction, one for the right-hand side residual) $\in \mathcal{G}_\ell$, each requiring $(2^\ell + 1)^2$ **doubles**.

$$\blacktriangleright \text{ total storage} = \sum_{\ell=2}^L 2(2^\ell + 1)^2 = 2 \sum_{\ell=2}^L (4^\ell + 2^{\ell+1} + 1).$$

So the total amount of memory required is just a small multiple of what it takes to store a single grid function on the finest level L .

SOLUTION of (4-2.k):

C++ code 4.2.19: Code for function `estimateMGConvergenceRate()`
Get it on  [GitLab \(mfmg.cpp\)](#).

```
2 double estimateMGConvergenceRate(double c, unsigned int L, double tol) {
3     unsigned M = std::pow(2, L) - 1;
4     double h = 1. / (M + 1);
5     double lambda_new = 1.;
6     double lambda_old = 0.;
7     DirichletBVPMultiGridSolver solver(c, L);
8     GridFunction v = Eigen::MatrixXd::Constant(M + 2, M + 2, 1);
9     v.col(0) = Eigen::VectorXd::Zero(M + 2);
10    v.col(M + 1) = Eigen::VectorXd::Zero(M + 2);
11    v.row(0) = Eigen::VectorXd::Zero(M + 2);
12    v.row(M + 1) = Eigen::VectorXd::Zero(M + 2);
13
14    GridFunction v_old;
15    GridFunction Av;
16    // Power iteration
17    do {
18        lambda_old = lambda_new;
19        v /= v.norm();
20        v_old = v;
21        Av = solver.applyGridOperator(L, v);
22        v = solver.multigridIteration(0 * v, Av, 2);
23        v = v_old - v;
24        lambda_new = v.norm();
25    } while (std::abs(lambda_new - lambda_old) > tol * lambda_new);
26    return lambda_new;
27 }
```

SOLUTION of (4-2.1):

C++ code 4.2.20: Code for function `tabulateMGConvergenceRate()`
Get it on  [GitLab \(mfmg.cpp\)](#).

```
2 void tabulateMGConvergenceRate() {
3     std::vector<double> c_vec({-40, -20, -10, -1, 0, 1, 10, 20, 40});
4     std::vector<unsigned> L_vec({6, 7, 8, 9, 10, 11});
5     std::cout << "c/L";
6     for (auto L : L_vec) {
7         std::cout << std::setw(15) << L;
8     }
9     std::cout << std::endl;
10    for (auto c : c_vec) {
11        std::cout << c;
12        for (auto L : L_vec) {
13            double lambda = estimateMGConvergenceRate(c, L);
14            std::cout << std::setw(15) << lambda;
15        }
16        std::cout << std::endl;
17    }
18 }
```


HINT 1 for (4-2.m): The member function `directSolve()` requires special considerations. When a line of the Galerkin matrix is related to an inactive node, then the diagonal entry should be set to one, and all off-diagonal entries should be set to zero.

SOLUTION of (4-2.m):

C++ code 4.2.21: L-shaped version of member function `applyGridOperator()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmngLshape.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::applyGridOperator(  
3     unsigned int level, const GridFunction &mu) const {  
4     // Assume the gridfunction mu vanishes at the boundary  
5     const unsigned int M = std::pow(2, level) - 1;  
6     const double h = 1.0 / (M + 1);  
7     GridFunction eta = Eigen::MatrixXd::Zero(M + 2, M + 2);  
8     double midpoint = (M + 1) / 2;  
9     for (unsigned int i = 1; i < M + 1; ++i) {  
10        for (unsigned int j = 1; j < M + 1; ++j) {  
11            //Evaluating the Matrix-vector product directly  
12            eta(i, j) = (4.0 + h*h * c_) * mu(i, j) +  
13                (-mu(i - 1, j) - mu(i + 1, j) - mu(i, j - 1) - mu(i, j + 1));  
14            if (i <= midpoint && j >= midpoint) {  
15                eta(i, j) = 0;  
16            }  
17        }  
18    }  
19    return eta;  
20 }
```

C++ code 4.2.22: L-shaped version of member function `directSolve()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmngLshape.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::directSolve(  
3     unsigned int level, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     const double h = 1.0 / (M + 1);  
6     const Eigen::MatrixXd phi_inner = phi.block(1, 1, M, M);  
7     const Eigen::VectorXd phi_v =  
8         Eigen::MatrixXd::Map(phi_inner.data(), M * M, 1);  
9     Eigen::VectorXd sol(M * M);  
10    Eigen::SparseMatrix<double> A(M * M, M * M);  
11    Eigen::SparseMatrix<double> A1(M * M, M * M);  
12    Eigen::SparseMatrix<double> A2(M * M, M * M);  
13    Eigen::SparseMatrix<double> T(M, M);  
14    // Unit matrix eye  
15    Eigen::SparseMatrix<double> Eye(M, M);  
16    A.reserve(Eigen::VectorXi::Constant(A.cols(), 5));  
17    A1.reserve(Eigen::VectorXi::Constant(A1.cols(), 5));  
18    A2.reserve(Eigen::VectorXi::Constant(A2.cols(), 5));  
19    T.reserve(Eigen::VectorXi::Constant(T.cols(), 3));  
20    Eye.reserve(Eigen::VectorXi::Constant(T.cols(), 1));  
21    T.insert(0, 0) = 2 + (h * h) * 0.5 * c_;  
22    T.insert(0, 1) = -1;  
23    T.insert(M - 1, M - 1) = 2. + (h * h) * 0.5 * c_;  
24    T.insert(M - 1, M - 2) = -1.;  
25    Eye.insert(0, 0) = 1.;  
26    Eye.insert(M - 1, M - 1) = 1.;  
27    for (unsigned int i = 1; i < M - 1; ++i) {
```

```

28     T.insert(i, i) = 2. + (h * h) * 0.5 * c_;
29     T.insert(i, i + 1) = -1.;
30     T.insert(i, i - 1) = -1.;
31     Eye.insert(i, i) = 1.;
32 }
33 Eigen::KroneckerProductSparse kron1(Eye, T);
34 Eigen::KroneckerProductSparse kron2(T, Eye);
35 kron1.evalTo(A1);
36 kron2.evalTo(A2);
37 A = A1 + A2;
38 int counter = 0;
39 // middle of the square, from north to south and east to west
40 int midpoint = (M + 1) / 2;
41
42 // We need 4 (spatial) indices: two identify the hat function in space that is in
43 // the domain of A,
44 // two different ones identify the hat function in the range of A.
45 // (the dimension of A is  $M * M \times M * M$ , the following indices go from 0 to  $M - 1$ 
46 // respectively.)
47 int ir = 0;
48 int jr = 0;
49 int ic = 0;
50 int jc = 0;
51 for (int k = 0; k < A.outerSize(); ++k) {
52     for (Eigen::SparseMatrix<double>::InnerIterator it(A, k);
53          it; ++it) {
54         ir = it.row() % M; // Row index of the unit square
55         jr = it.row() / M; // Column index of the unit square
56         ic = it.col() % M; // Row index of the unit square
57         jc = it.col() / M; // Column index of the unit square
58         if ((ir <= midpoint - 1 && jr >= midpoint - 1) ||
59             (ic <= midpoint - 1 && jc >= midpoint - 1)) {
60             // We are in the upper right quadrant
61             if (it.col() == it.row()) {
62                 // Diagonal entry
63                 it.valueRef() = 1.;
64             }
65             if (it.col() != it.row()) {
66                 // Entries away from the diagonal
67                 it.valueRef() = 0.;
68             }
69         }
70     }
71 }
72 Eigen::SparseLU<Eigen::SparseMatrix<double>> sparse_solver;
73 sparse_solver.analyzePattern(A);
74 sparse_solver.factorize(A);
75 assert(sparse_solver.info() == Eigen::Success);
76 sol = sparse_solver.solve(phi_v);
77 GridFunction sol_g = Eigen::MatrixXd::Zero(M + 2, M + 2);
78 sol_g.block(1, 1, M, M) = Eigen::MatrixXd::Map(sol.data(), M, M);
79 return sol_g;
80 }

```

C++ code 4.2.23: L-shaped version of member function `sweepGaussSeidel()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmgLshape.cpp\)](#).

```
2 void DirichletBVPMultiGridSolver::sweepGaussSeidel(  
3     unsigned int level, GridFunction &mu, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     const double h = 1.0 / (M + 1);  
6     const double midpoint = (M + 1) / 2;  
7     for (unsigned int i = 1; i < M + 1; ++i) {  
8         for (unsigned int j = 1; j < M + 1; ++j) {  
9             mu(i, j) = (phi(i, j) +  
10                (mu(i - 1, j) + mu(i + 1, j) + mu(i, j - 1) + mu(i, j + 1))) /  
11                (4.0 + h * h * c_);  
12             if (i <= midpoint && j >= midpoint) {  
13                 mu(i, j) = 0;  
14             }  
15         }  
16     }  
17 }
```

C++ code 4.2.24: L-shaped version of member function `residual()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmgLshape.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::residual(  
3     unsigned int level, const GridFunction &mu, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     GridFunction res(M + 2, M + 2);  
6     res = (phi - applyGridOperator(level, mu));  
7     return res;  
8 }
```

C++ code 4.2.25: L-shaped version of member function `directSolve()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmgLshape.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::prolongate(  
3     unsigned int level, const GridFunction &gamma) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     GridFunction zeta((M + 2), (M + 2));  
6     zeta.setZero();  
7     for (unsigned int i = 1; i < M + 1; ++i) {  
8         //odd  
9         for (unsigned int j = 1; j < M + 1; ++j) {  
10            //odd j  
11            zeta(i, j) =  
12                0.25 *  
13                (gamma((i - 1) / 2, (j - 1) / 2) + gamma((i - 1) / 2, (j + 1) / 2) +  
14                gamma((i + 1) / 2, (j - 1) / 2) + gamma((i + 1) / 2, (j + 1) / 2));  
15            // even j  
16            ++j;  
17            zeta(i, j) =  
18                0.5 * (gamma((i - 1) / 2, j / 2) + gamma((i + 1) / 2, j / 2));  
19        }  
20        ++i;  
21        // even i  
22        for (unsigned int j = 1; j < M + 1; ++j) {
```

```

23     //odd j
24     zeta(i, j) =
25         0.5 * (gamma(i / 2, (j - 1) / 2) + gamma(i / 2, (j + 1) / 2));
26     // even j
27     ++j;
28     zeta(i, j) = gamma(i / 2, j / 2);
29 }
30 }
31 return zeta;
32 }

```

C++ code 4.2.26: L-shaped version of member function `restrict()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmglshape.cpp\)](#).

```

2 GridFunction DirichletBVPMultiGridSolver::restrict(
3     unsigned int level, const GridFunction &rho) const {
4     const unsigned int M = std::pow(2, level - 1) - 1;
5     GridFunction sigma((M + 2), (M + 2));
6     sigma.setZero();
7     for (unsigned int i = 1; i < M + 1; ++i) {
8         for (unsigned int j = 1; j < M + 1; ++j) {
9             sigma(i, j) = rho(2 * i, 2 * j);
10            sigma(i, j) += 0.5 * (rho(2 * i, 2 * j + 1) + rho(2 * i, 2 * j - 1) +
11                               rho(2 * i + 1, 2 * j) + rho(2 * i - 1, 2 * j));
12
13            sigma(i, j) +=
14                0.25 * (rho(2 * i - 1, 2 * j - 1) + rho(2 * i - 1, 2 * j + 1) +
15                      rho(2 * i + 1, 2 * j - 1) + rho(2 * i + 1, 2 * j + 1));
16        }
17    }
18    return sigma;
19 }

```

C++ code 4.2.27: L-shaped version of member function `multigridIteration()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmglshape.cpp\)](#).

```

2 GridFunction DirichletBVPMultiGridSolver::multigridIteration(
3     const GridFunction &mu, const GridFunction &phi, unsigned int L0) const {
4     std::vector<GridFunction> mu_vec;
5     std::vector<GridFunction> phi_vec;
6     std::vector<GridFunction> residual_vec;
7     mu_vec.push_back(mu);
8     const unsigned int M = std::pow(2, L0) - 1;
9     const double h = 1.0 / (M + 1);
10    phi_vec.push_back(phi);
11
12    unsigned counter = 0;
13    // applies pre-smoothers and restrictions until L0 is reached
14    for (unsigned l = L_; l > L0; --l) {
15        sweepGaussSeidel(l, mu_vec.back(), phi_vec.back()); // Pre smoothing
16        residual_vec.push_back(
17            residual(l, mu_vec.back(),
18                  phi_vec.back())); // Compute residual and adds to residual
19        phi_vec.push_back(restrict(
20            l, residual_vec.back())); // Restrict residual and append

```

```
21 GridFunction zero = phi_vec.back();
22 zero.setZero();
23 mu_vec.push_back(zero); // Adds the zero grid function
24 ++counter;
25 }
26 mu_vec.back() = directSolve(L0, phi_vec.back());
27 for (unsigned l = L0; l < L_; ++l) {
28     mu_vec[counter - 1] += prolongate(l + 1, mu_vec[counter]); //prolongate mu
29     sweepGaussSeidel(l + 1, mu_vec[counter - 1],
30                     phi_vec[counter - 1]); //post-smoothing
31     --counter;
32 }
33 return mu_vec[0];
34 }
```

SOLUTION of (4-2.b):

As in [Lecture → Ex. 4.1.1.24] we obtain

$$\mathbf{A} := \begin{pmatrix} \mathbf{B} & -\mathbf{I} & 0 & \cdots & \cdots & 0 \\ -\mathbf{I} & \mathbf{B} & -\mathbf{I} & & & \vdots \\ 0 & -\mathbf{I} & \mathbf{B} & -\mathbf{I} & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & -\mathbf{I} & \mathbf{B} & -\mathbf{I} \\ 0 & \cdots & \cdots & 0 & -\mathbf{I} & \mathbf{B} \end{pmatrix} \in \mathbb{R}^{M^2, M^2}, \quad (4.2.4)$$

$$\mathbf{B} := \begin{pmatrix} 4+h^2c & -1 & 0 & & & 0 \\ -1 & 4+h^2c & -1 & & & \vdots \\ 0 & -1 & 4+h^2c & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & & -1 & 4+h^2c & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 4+h^2c \end{pmatrix} \in \mathbb{R}^{M, M}.$$

Examining the structure of a Kronecker product, we conclude that in (4.2.3)

$$\mathbf{T} := \begin{pmatrix} 2+\frac{1}{2}h^2c & -1 & 0 & & & 0 \\ -1 & 2+\frac{1}{2}h^2c & -1 & & & \vdots \\ 0 & -1 & 2+\frac{1}{2}h^2c & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & & -1 & 2+\frac{1}{2}h^2c & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2+\frac{1}{2}h^2c \end{pmatrix} \in \mathbb{R}^{M, M} \quad (4.2.5)$$

is one possible choice for the tri-diagonal matrix \mathbf{T} .

SOLUTION of (4-2.c):

The only memory that should be allocated is that required for the result matrix. Otherwise do a double loop over the grid and carry out all operations in a stencil-based way.

The matrix \mathbf{A}_ℓ represents a translation-invariant local operator on the tensor-product grid \mathcal{M}_ℓ , whose stencil notation [Lecture \rightarrow § 4.1.1.31] is

$$\mathbf{A}_\ell \longleftrightarrow \begin{bmatrix} & & -1 & & \\ -1 & 4 + ch^2 & -1 & & \\ & & -1 & & \\ & & & & \\ & & & & \end{bmatrix}_h. \quad (4.2.8)$$

This means that the grid function $\underline{\eta}$ for $\vec{\eta} := \mathbf{A}_\ell \vec{\mu}$ is

$$\eta_{i,j} := (4 + h^2 c) \mu_{i,j} - \mu_{i-1,j} - \mu_{i+1,j} - \mu_{i,j-1} - \mu_{i,j+1}, \quad i, j \in \{1, \dots, 2^\ell - 1\}, \quad (4.2.9)$$

where grid function components with “illegal” indices are to be replaced with zero.

Note that the zero boundary of the **GridFunction** data layout avoids the need for checking inadmissible indices in the case of gridpoints adjacent to the boundary.

C++ code 4.2.10: Member function `applyGridOperator()` of **DirichletBVPMultiGridSolver**. Get it on [GitLab \(mfmg.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::applyGridOperator(  
3     unsigned int level, const GridFunction &mu) const {  
4     // Assume the gridfunction mu vanishes at the boundary  
5     const unsigned int M = std::pow(2, level) - 1;  
6     const double h = 1.0 / (M + 1);  
7     GridFunction eta = Eigen::MatrixXd::Zero(M + 2, M + 2);  
8     for (unsigned int i = 1; i < M + 1; ++i) {  
9         for (unsigned int j = 1; j < M + 1; ++j) {  
10            // Evaluating the Matrix-vector product directly  
11            eta(i, j) = (4.0 + h * h * c_) * mu(i, j) +  
12                (-mu(i - 1, j) - mu(i + 1, j) - mu(i, j - 1) - mu(i, j + 1));  
13        }  
14    }  
15    return eta;  
16 }
```



HINT 1 for (4-2.d): Consult [[NumCSE course](#) → Section 2.7.3] to learn how to use EIGEN's sparse solver. Direct initialization of the sparse matrix as in [[NumCSE course](#) → Code 2.7.2.1] is recommended.



SOLUTION of (4-2.d):

A key decision has to be made how to map the **GridFunction** to a vector. For instance, this can be done through lexicographically numbering the interior grid nodes as in [Lecture → Ex. 4.1.1.22].

Based on this numbering, the stencil description of **A** is in one-to-one correspondence to a matrix representation, remember (4.2.4) from Sub-problem (4-2.b). That matrix **A** has to be stored as an **Eigen::SparseMatrix** object. Initialization can be done by means of its `insert()` member function (after appropriate `reserve`

C++ code 4.2.11: Member function `directSolve()` of **DirichletBVPMultiGridSolver**.
Get it on  [GitLab \(mfm.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::directSolve(  
3     unsigned int level, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     const double h = 1.0 / (M + 1);  
6     const Eigen::MatrixXd phi_inner = phi.block(1, 1, M, M);  
7     const Eigen::VectorXd phi_v =  
8         Eigen::MatrixXd::Map(phi_inner.data(), M * M, 1);  
9     Eigen::VectorXd sol(M * M);  
10    Eigen::SparseMatrix<double> A(M * M, M * M);  
11    Eigen::SparseMatrix<double> A1(M * M, M * M);  
12    Eigen::SparseMatrix<double> A2(M * M, M * M);  
13    Eigen::SparseMatrix<double> T(M, M);  
14    A.reserve(Eigen::VectorXi::Constant(A.cols(), 5));  
15    A1.reserve(Eigen::VectorXi::Constant(A1.cols(), 5));  
16    A2.reserve(Eigen::VectorXi::Constant(A2.cols(), 5));  
17    T.reserve(Eigen::VectorXi::Constant(T.cols(), 3));  
18    T.insert(0, 0) = 2 + (h * h) * 0.5 * c_;  
19    T.insert(0, 1) = -1;  
20    T.insert(M - 1, M - 1) = 2. + (h * h) * 0.5 * c_;  
21    T.insert(M - 1, M - 2) = -1.;  
22    for (unsigned int i = 1; i < M - 1; ++i) {  
23        T.insert(i, i) = 2. + (h * h) * 0.5 * c_;  
24        T.insert(i, i + 1) = -1.;  
25        T.insert(i, i - 1) = -1.;  
26    }  
27  
28    Eigen::KroneckerProductSparse kron1(  
29        Eigen::MatrixXd::Identity(T.cols(), T.cols()), T);  
30    Eigen::KroneckerProductSparse kron2(  
31        T, Eigen::MatrixXd::Identity(T.cols(), T.cols()));  
32    kron1.evalTo(A1);  
33    kron2.evalTo(A2);  
34    A = A1 + A2;  
35    Eigen::SparseLU<Eigen::SparseMatrix<double>> sparse_solver;  
36    sparse_solver.analyzePattern(A);  
37    sparse_solver.factorize(A);  
38    assert(sparse_solver.info() == Eigen::Success);  
39    sol = sparse_solver.solve(phi_v);  
40    GridFunction sol_g = Eigen::MatrixXd::Zero(M + 2, M + 2);  
41    sol_g.block(1, 1, M, M) = Eigen::MatrixXd::Map(sol.data(), M, M);  
42    return sol_g;  
43 }
```



SOLUTION of (4-2.e):

A single Gauss-Seidel sweep is implemented in the inner loop of [Lecture → Code 4.1.3.3]. Obviously, no temporary storage is required!


C++ code 4.2.12: Member function `sweepGaussSeidel()` of `DirichletBVPMultiGridSolver`.
Get it on  [GitLab \(mfm.cpp\)](#).

```
2 void DirichletBVPMultiGridSolver::sweepGaussSeidel(  
3     unsigned int level, GridFunction &mu, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     const double h = 1.0 / (M + 1);  
6     for (unsigned int i = 1; i < M + 1; ++i) {  
7         for (unsigned int j = 1; j < M + 1; ++j) {  
8             mu(i, j) = (phi(i, j) +  
9                 (mu(i - 1, j) + mu(i + 1, j) + mu(i, j - 1) + mu(i, j + 1))) /  
10                (4.0 + h * h * c_);  
11         }  
12     }  
13 }
```

SOLUTION of (4-2.f):

The only memory that should be allocated is that required for the result **GridFunction**. Otherwise do a double loop over the grid and do all operations stencil based.

As in Sub-problem (4-2.c) the outer rim of the **GridFunction** data layout avoids the need for checking inadmissible indices in the case of gridpoints adjacent to the boundary.

C++ code 4.2.13: Member function `residual()` of `DirichletBVPMultiGridSolver`.
Get it on  [GitLab \(mfmng.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::residual(  
3     unsigned int level, const GridFunction &mu, const GridFunction &phi) const {  
4     const unsigned int M = std::pow(2, level) - 1;  
5     GridFunction res(M + 2, M + 2);  
6     res = (phi - applyGridOperator(level, mu));  
7     return res;  
8 }
```

HINT 1 for (4-2.g): The process of prolongation, that is the application of the prolongation matrix, is explained in detail in [Lecture → Section 4.2.3]. Thanks to the cardinal basis property of the global shape functions of $\mathcal{S}_1^0(\mathcal{M}_\ell)$ it amounts to computing the values of a finite-element function $\in \mathcal{S}_{1,0}^0(\mathcal{M}_{\ell-1})$ in the nodes of \mathcal{M}_ℓ . ┘

SOLUTION of (4-2.g):

The only memory that should be allocated is that required for the result matrix. The values of the prolonged grid function $\underline{\zeta}$ in the node $h \begin{bmatrix} i \\ j \end{bmatrix}$ of \mathcal{M}_ℓ are set as follows, when $\vec{\zeta} := \mathbf{P}\vec{\mu}$, \mathbf{P} the prolongation matrix:

$$\zeta_{i,j} := \begin{cases} \mu_{i/2,j/2} & \text{if } i,j \text{ are even,} \\ \frac{1}{2}(\mu_{i/2,(j+1)/2} + \mu_{i/2,(j-1)/2}) & \text{if } i \text{ even, } j \text{ odd,} \\ \frac{1}{2}(\mu_{(i-1)/2,j/2} + \mu_{(i+1)/2,j/2}) & \text{if } j \text{ even, } i \text{ odd,} \\ \frac{1}{4}(\mu_{(i-1)/2,(j-1)/2} + \mu_{(i-1)/2,(j+1)/2} + \mu_{(i+1)/2,(j-1)/2} + \mu_{(i+1)/2,(j+1)/2}) & \text{if } i,j \text{ are odd,} \end{cases} \quad (4.2.14)$$

$i, j \in \{1, \dots, 2^\ell - 1\}$. Grid function values for indices outside the valid range are assumed to vanish.

C++ code 4.2.15: Member function `directSolve()` of `DirichletBVPMultiGridSolver`.
Get it on [GitLab](#) (`mfmfg.cpp`).

```

2 GridFunction DirichletBVPMultiGridSolver::prolongate(
3     unsigned int level, const GridFunction &gamma) const {
4     const unsigned int M = std::pow(2, level) - 1;
5     GridFunction zeta((M + 2), (M + 2));
6     zeta.setZero();
7     for (unsigned int i = 1; i < M + 1; ++i) {
8         // odd
9         for (unsigned int j = 1; j < M + 1; ++j) {
10            // odd j
11            zeta(i, j) =
12                0.25 *
13                (gamma((i - 1) / 2, (j - 1) / 2) + gamma((i - 1) / 2, (j + 1) / 2) +
14                gamma((i + 1) / 2, (j - 1) / 2) + gamma((i + 1) / 2, (j + 1) / 2));
15            // even j
16            ++j;
17            zeta(i, j) =
18                0.5 * (gamma((i - 1) / 2, j / 2) + gamma((i + 1) / 2, j / 2));
19        }
20        ++i;
21        // even i
22        for (unsigned int j = 1; j < M + 1; ++j) {
23            // odd j
24            zeta(i, j) =
25                0.5 * (gamma(i / 2, (j - 1) / 2) + gamma(i / 2, (j + 1) / 2));
26            // even j
27            ++j;
28            zeta(i, j) = gamma(i / 2, j / 2);
29        }
30    }
31    return zeta;
32 }

```

HINT 1 for (4-2.h): The process of restriction amounts to the application of the transposed prolongation matrix, see [Lecture → Code 4.2.3.12].

SOLUTION of (4-2.h):

The only memory that should be allocated is that required for the result matrix. The values of the restricted grid function $\underline{\rho}$ in the node $h \begin{bmatrix} i \\ j \end{bmatrix}$ of \mathcal{M}_ℓ are set as follows, when $\vec{\sigma} := \mathbf{P}^\top \vec{\rho}$, \mathbf{P} the prolongation matrix:

$$\sigma_{i,j} := \rho_{2i,2j} + \frac{1}{2}(\rho_{2i,2j-1} + \rho_{2i,2j+1} + \rho_{2i-1,2j} + \rho_{2i+1,2j}) + \frac{1}{4}(\rho_{2i-1,2j-1} + \rho_{2i-1,2j+1} + \rho_{2i+1,2j-1} + \rho_{2i+1,2j+1}), \quad (4.2.16)$$

$i, j \in \{1, \dots, 2^{\ell-1} - 1\}$. This is the “transpose” of (4.2.14) reversing the flow of information. A double-loop implementation is straightforward.

C++ code 4.2.17: Member function `restrict()` of `DirichletBVPMultiGridSolver`.
Get it on [GitLab \(mfmfg.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::restrict(  
3     unsigned int level, const GridFunction &rho) const {  
4     const unsigned int M = std::pow(2, level - 1) - 1;  
5     GridFunction sigma((M + 2), (M + 2));  
6     sigma.setZero();  
7     for (unsigned int i = 1; i < M + 1; ++i) {  
8         for (unsigned int j = 1; j < M + 1; ++j) {  
9             sigma(i, j) = rho(2 * i, 2 * j);  
10            sigma(i, j) += 0.5 * (rho(2 * i, 2 * j + 1) + rho(2 * i, 2 * j - 1) +  
11                rho(2 * i + 1, 2 * j) + rho(2 * i - 1, 2 * j));  
12            sigma(i, j) +=  
13                0.25 * (rho(2 * i - 1, 2 * j - 1) + rho(2 * i - 1, 2 * j + 1) +  
14                    rho(2 * i + 1, 2 * j - 1) + rho(2 * i + 1, 2 * j + 1));  
15        }  
16    }  
17    return sigma;  
18 }
```

HINT 1 for (4-2.i): You need not implement a recursion; two loops over the levels are sufficient. Make sure that you use as little temporary memory as possible.

SOLUTION of (4-2.i):

Except for `directSolve()` your implementation need not rely on any other member function. of course, you may use them, but this will incur some memory penalty.

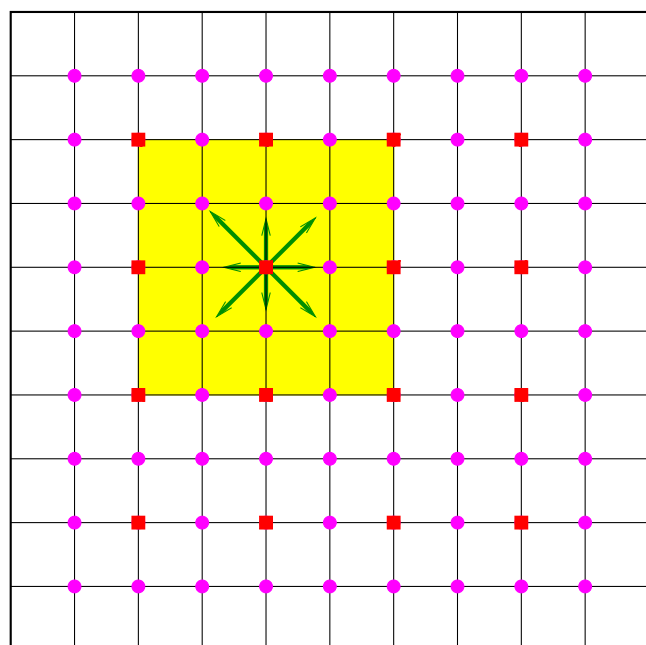
C++ code 4.2.18: Member function `multigridIteration()` of `DirichletBVPMultiGridSolver`. Get it on [GitLab \(mfmng.cpp\)](#).

```
2 GridFunction DirichletBVPMultiGridSolver::multigridIteration(
3     const GridFunction &mu, const GridFunction &phi, unsigned int L0) const {
4     std::vector<GridFunction> mu_vec;
5     std::vector<GridFunction> phi_vec;
6     std::vector<GridFunction> residual_vec;
7     mu_vec.push_back(mu);
8     const unsigned int M = std::pow(2, L0) - 1;
9     const double h = 1.0 / (M + 1);
10    phi_vec.push_back(phi);
11
12    unsigned counter = 0;
13    // applies pre-smoothers and restrictions until L0 is reached
14    for (unsigned l = L_; l > L0; --l) {
15        sweepGaussSeidel(l, mu_vec.back(), phi_vec.back()); // Pre smoothing
16        residual_vec.push_back(
17            residual(l, mu_vec.back(),
18                    phi_vec.back())); // Compute residual and adds to residual
19        phi_vec.push_back(
20            restrict(l, residual_vec.back())); // Restrict residual and add to phi
21        GridFunction zero = phi_vec.back();
22        zero.setZero();
23        mu_vec.push_back(zero); // Adds the zero grid function to mu
24        ++counter;
25    }
26    // Now we do a direct solve at level L0
27    mu_vec.back() = directSolve(L0, phi_vec.back());
28    // Prolongate and do a post-smoothening
29    for (unsigned l = L0; l < L_; ++l) {
30        mu_vec[counter - 1] += prolongate(l + 1, mu_vec[counter]); // prolongate mu
31        sweepGaussSeidel(l + 1, mu_vec[counter - 1],
32                        phi_vec[counter - 1]); // post-smoothening
33        --counter;
34    }
35    return mu_vec[0];
36 }
```

SOLUTION of (4-3.a):

Denote the global shape function of $\mathcal{S}_{1,0}^0(\mathcal{M}_X)$, $X = M, m$, associated with the node at $[ih_X, jh_X]^\top$, $h_M = 1/M$, $h_m := 1/m$, as $b_{i,j}^X$. Evaluating the bilinear tent functions locally on the cells of the mesh and taking into account the cardinal basis property of the tent functions with respect to the nodes of the mesh, we find

$$\begin{aligned} b_{i,j}^m &= \sum_{k=-1}^1 \sum_{\ell=-1}^1 b_{i,j}^m \left(\begin{bmatrix} ih_m + kh_M \\ jh_m + \ell h_M \end{bmatrix} \right) \cdot b_{2i+k,2j+\ell}^M = \\ &= b_{2i,2j}^M + \frac{1}{2}b_{2i+1,2j}^M + \frac{1}{2}b_{2i-1,2j}^M + \frac{1}{2}b_{2i,2j+1}^M + \frac{1}{2}b_{2i,2j-1}^M + \\ &\quad \frac{1}{4}b_{2i+1,2j+1}^M + \frac{1}{4}b_{2i-1,2j+1}^M + \frac{1}{4}b_{2i+1,2j-1}^M + \frac{1}{4}b_{2i-1,2j-1}^M. \end{aligned} \quad (4.3.2)$$



Case ❶:

◁ ● $\hat{=}$ additional fine-grid nodes

■ $\hat{=}$ coarse-grid nodes

■ $\hat{=}$ support of $b_{i,j}^m$

The action of \mathbf{P} indicated by \rightarrow in Fig. 15 can be visualized with a **stencil notation** [Lecture \rightarrow § 4.1.1.31]:

$$\mathbf{P} \sim \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}_h. \quad (4.3.3)$$

The numbers occurring in the stencils correspond to entries in a row of \mathbf{P} .

The size of \mathbf{P} is $(M-1)^2 \times (m-1)^2 = \dim \mathcal{S}_{1,0}^0(\mathcal{M}) \mathcal{M}_M \times \dim \mathcal{S}_{1,0}^0(\mathcal{M}_m)$. Since \mathbf{P} has just nine non-zero entries in each row, it is a **sparse matrix**.

The initialization of the entries of \mathbf{P} is facilitated by introducing an index mapping from node positions to numbers of associated basis functions, e.g. for $X = M$:

$$(ih_M, jh_M) \longleftrightarrow \text{index of } b_{i,j}^X = (i-1)(M-1) + (j-1), \quad 1 \leq i, j < M.$$

Here, C++ indexing has been used.

C++ code 4.3.4: Construction of prolongation matrix
 Get it on [GitLab \(galerkinconstruction.cpp\)](https://gitlab.com/galerkinconstruction).

```

2 Eigen::SparseMatrix<double, Eigen::RowMajor> prolongationMatrix(unsigned int M,
3                                     bool bilinear) {
4     assertm(((M > 3) and (M % 2 == 0)), "prolongationMatrix: M must be even!");
5     const unsigned int N = (M - 1) * (M - 1); // No of inter nodes of fine grid
6     const unsigned int m =
7         M / 2; // Number of cells in each direction of coarse grid
8     const unsigned int n =
  
```

```

9     (m - 1) * (m - 1); // No of interior nodes of fine grid
10    // Sparse matrix in CCS format
11    Eigen::SparseMatrix<double, Eigen::RowMajor> P(N, n);
12    #if SOLUTION
13    // define vector of triplets and reserve memory
14    std::vector<triplet> P_trp; // For temporary COO format
15    // Conversion of node positions to indices: lexikographic ordering
16    auto idxH = [&m](unsigned int I, unsigned int J) -> unsigned int {
17        assertm(((I > 0) and (I < m) and (J > 0) and (J < m)),
18                "idxH: index out of range");
19        return (I - 1) + (J - 1) * (m - 1);
20    };
21    auto idxh = [&M](unsigned int i, unsigned int j) -> unsigned int {
22        assertm(((i > 0) and (i < M) and (j > 0) and (j < M)),
23                "idxh: index out of range");
24        return (i - 1) + (j - 1) * (M - 1);
25    };
26    // Traverse all nodes of the coarse mesh
27    for (unsigned int I = 1; I < m; ++I) {
28        for (unsigned int J = 1; J < m; ++J) {
29            // Set values according to prolongation stencil
30            const unsigned int i = 2 * I;
31            const unsigned int j = 2 * J;
32            const unsigned int idx_H = idxH(I, J);
33            P_trp.emplace_back(idxh(i, j), idx_H, 1.0);
34            P_trp.emplace_back(idxh(i + 1, j), idx_H, 0.5);
35            P_trp.emplace_back(idxh(i - 1, j), idx_H, 0.5);
36            P_trp.emplace_back(idxh(i, j + 1), idx_H, 0.5);
37            P_trp.emplace_back(idxh(i, j - 1), idx_H, 0.5);
38            if (bilinear) {
39                P_trp.emplace_back(idxh(i + 1, j + 1), idx_H, 0.25);
40                P_trp.emplace_back(idxh(i + 1, j - 1), idx_H, 0.25);
41                P_trp.emplace_back(idxh(i - 1, j + 1), idx_H, 0.25);
42                P_trp.emplace_back(idxh(i - 1, j - 1), idx_H, 0.25);
43            } else {
44                P_trp.emplace_back(idxh(i + 1, j + 1), idx_H, 0.5);
45                P_trp.emplace_back(idxh(i - 1, j - 1), idx_H, 0.5);
46            }
47        }
48    }
49    // Initialize CCS matrix from COO format
50    P.setFromTriplets(P_trp.begin(), P_trp.end());
51    #else
52    //
53    // *****
54    // To be supplemented
55    // *****
56    #endif
57    return P;
58 }

```

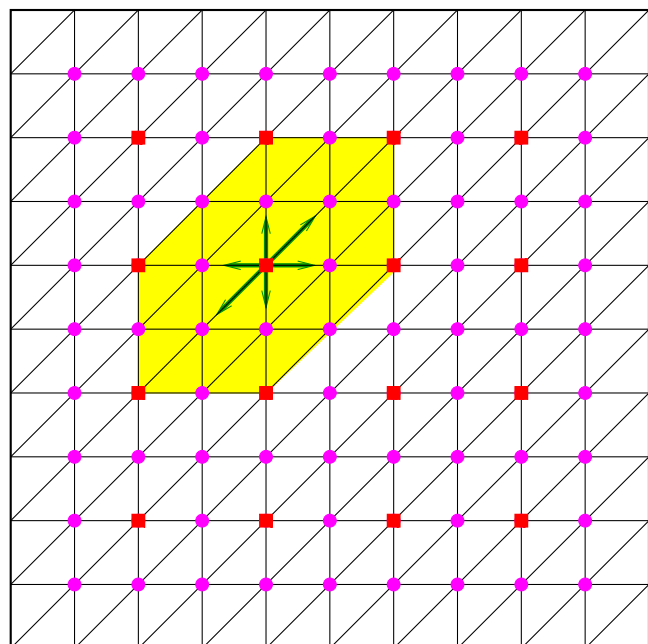
Alternatively, you may use the `reserve()` method of `Eigen::SparseMatrix` to tell the computer how many non-zero entries `P` can have at most. Then use the `insert()` method to set matrix entries. This will also be efficient.

SOLUTION of (4-3.b):

We still use the notation introduced in the solution of Sub-problem (4-3.a).

Now we have

$$\begin{aligned}
 b_{i,j}^m &= \sum_{k=-1}^1 \sum_{\ell=-1}^1 b_{i,j}^m \left(\begin{bmatrix} ih_m + kh_M \\ jh_m + \ell h_M \end{bmatrix} \right) \cdot b_{2i+k,2j+\ell}^M = \\
 &= b_{2i,2j}^M + \frac{1}{2}b_{2i+1,2j}^M + \frac{1}{2}b_{2i-1,2j}^M + \frac{1}{2}b_{2i,2j+1}^M + \frac{1}{2}b_{2i,2j-1}^M + \\
 &\quad \frac{1}{2}b_{2i+1,2j+1}^M + \frac{1}{2}b_{2i-1,2j-1}^M \cdot
 \end{aligned} \tag{4.3.5}$$



Case ②:

◁ ● $\hat{=}$ additional fine-grid nodes

■ $\hat{=}$ coarse-grid nodes

■ $\hat{=}$ support of $b_{i,j}^m$

The action of \mathbf{P} indicated by \rightarrow in Fig. 15 can be visualized with a stencil notation [Lecture \rightarrow § 4.1.1.31]:

$$\mathbf{P} \sim \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}_h \cdot \tag{4.3.6}$$

The numbers occurring in the stencils correspond to entries in a row of \mathbf{P} .

See Code 4.3.4 for the code.

HINT 1 for (4-3.c): Study the code [[NumCSE course](#) → Code 2.7.2.4].

The `outerSize()` of a an object of type

`Eigen::SparseMatrix<double, Eigen::RowMajor>`

is the number of rows. Its **`Innterlterator`** will allow you to run through then non-zero entries in a particular row, see [EIGEN documentation](#). ┘

SOLUTION of (4-3.c):

For the sake of efficiency we have to initialize the result matrix via the triplet (COO) format.

Write N for the dimension of the finite-element space on the fine mesh, n for the dimension of the finite-element space on the coarse mesh. This implies

$$\mathbf{A} \in \mathbb{R}^{N,N}, \quad \mathbf{P} \in \mathbb{R}^{N,n} \quad \implies \quad \mathbf{A}_H \in \mathbb{R}^{n,n}.$$

The key formula from [Lecture \rightarrow § 4.3.1.2] is

$$(\mathbf{A}_H)_{i,j} = \mathbf{a}(b_H^j, b_H^i) = \sum_{\ell=1}^N \sum_{k=1}^N \mathbf{a}((\mathbf{P})_{\ell,j} b_H^\ell, (\mathbf{P})_{k,i} b_H^k) = \sum_{\ell=1}^N \sum_{k=1}^N (\mathbf{P})_{\ell,j} (\mathbf{P})_{k,i} \mathbf{a}(b_H^\ell, b_H^k) = (\mathbf{P}^\top \mathbf{A} \mathbf{P})_{i,j}.$$

Each term

$$\alpha_{i,j}^{\ell,k} := (\mathbf{P})_{\ell,j} (\mathbf{P})_{k,i} (\mathbf{A})_{\ell,k}, \quad 1 \leq i, j \leq n, \quad 1 \leq \ell, k \leq N,$$

will spawn a triplet $(i, j, \alpha_{i,j}^{\ell,k})$. We create those within four nested loops:

- The two outer loops run from 1 to N (indices ℓ, k). Of course, only non-zero entries of \mathbf{A} are to be visited.
- The two inner loops run from 1 to n (indices i, j).

All these loops, except for the outermost one, can be realized by means of **InnerIterator** s.

C++ code 4.3.8: Construction of coarse-grid matrix $\mathbf{A}_H = \mathbf{P}^\top \mathbf{A} \mathbf{P}$
Get it on [GitLab \(galerkinconstruction.cpp\)](https://gitlab.com/galerkinconstruction.cpp).

```
2 Eigen::SparseMatrix<double> buildAH(  
3     const Eigen::SparseMatrix<double> &A,  
4     const Eigen::SparseMatrix<double, Eigen::RowMajor> &P) {  
5     Eigen::SparseMatrix<double> AH(P.cols(), P.cols());  
6     #if SOLUTION  
7         // define vector of triplets and reserve memory  
8         std::vector<Eigen::Triplet<double>> AH_trp{}; // For temporary COO format  
9         // For both A and P the inner dimensions are rows  
10        for (int k = 0; k < A.outerSize(); ++k) // loop over columns of A  
11            for (Eigen::SparseMatrix<double>::InnerIterator l(A, k); l;  
12                 ++l) // loop over rows of A  
13                for (Eigen::SparseMatrix<double, Eigen::RowMajor>::InnerIterator j(  
14                     P, l.row());  
15                     j; ++j) // loop over rows of P  
16                    for (Eigen::SparseMatrix<double, Eigen::RowMajor>::InnerIterator i(P,  
17                         k);  
18                         i; ++i) // loop over rows of P  
19                        AH_trp.emplace_back(i.col(), j.col(),  
20                                              j.value() * i.value() * l.value());  
21        // Initialize CCS matrix from COO format  
22        AH.setFromTriplets(AH_trp.begin(), AH_trp.end());  
23    #else  
24        //  
25        // *****  
26        // To be supplemented  
27    #endif
```

27 #endif

28 return AH;

29 }



SOLUTION of (4-3.d):

Write $\mathbf{A}_h/\mathbf{A}_H$ for the finite-element Galerkin matrices on the fine/coarse mesh and \mathbf{P} for the prolongation matrix.

In the case

- of exact Galerkin matrices,
- and nested FE spaces

the relationship $\mathbf{A}_H = \mathbf{P}^\top \mathbf{A}_h \mathbf{P}$ will always be satisfied. These conditions are met for finite-element discretization ②, `AH_L` and `P_L` in Code 4.3.9.

However, in case ① numerical quadrature was employed and the Galerkin matrices are not exact. This leads to $\mathbf{A}_H \neq \mathbf{P}^\top \mathbf{A}_h \mathbf{P}$.

HINT 1 for (4-3.e): This code snippet shows the use of the C++ `chrono` library for measuring runtimes:

C++ code 4.3.11: Measuring runtime of a part of a C++ code.
Get it on [GitLab \(gravitationalforces.cpp\)](#).

```
2 void runtimeMeasuredemo(void) {
3     auto t1 = std::chrono::high_resolution_clock::now();
4     double s = 0.0;
5     for (long int i = 0; i < 10000000; ++i) {
6         s += 1.0 / std::sqrt((double)i);
7     }
8     auto t2 = std::chrono::high_resolution_clock::now();
9     /* Getting number of milliseconds as a double. */
10    std::chrono::duration<double, std::milli> ms_double = t2 - t1;
11    std::cout << "Runtime = " << ms_double.count() << "ms\n";
12 }
```



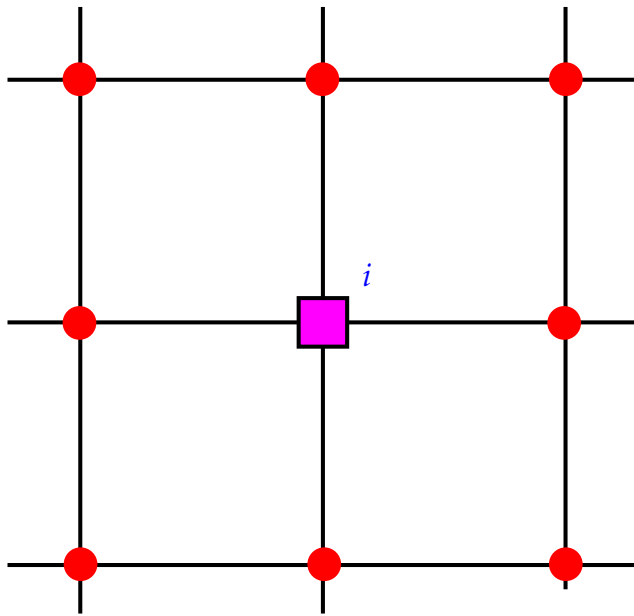
For runtime measurements it is essential to compile your code in *Release Mode*: configure your CMake build system accordingly!

SOLUTION of (4-3.e):

C++ code 4.3.12: Construction of coarse-grid matrix $A_H = P^T A P$
Get it on [GitLab \(galerkinconstruction.h\)](#).

```
2 template <typename SEQ>
3 void tabulateRuntimes(SEQ &&M_vals) {
4     #if SOLUTION
5         // Take the minimum over several runs as runtime
6         int n_runs = 5;
7         std::cout << "MtEigen\t\tMine" << std::endl;
8
9         for (auto M : M_vals) {
10            // Obtain the fine-grid matrix
11            const Eigen::SparseMatrix<double> Ah = poissonMatrix(M);
12            // Generate the prolongation matrix
13            const Eigen::SparseMatrix<double, Eigen::RowMajor> P =
14                prolongationMatrix(M, true);
15            // Measure runtime of Eigen's multiplication of sparse matrices
16            double ms_eigen = std::numeric_limits<double>::max();
17            for (int r = 0; r < n_runs; r++) {
18                auto t1_eigen = std::chrono::high_resolution_clock::now();
19                const Eigen::SparseMatrix<double> AH_eigen = buildAH_eigen(Ah, P);
20                auto t2_eigen = std::chrono::high_resolution_clock::now();
21                std::chrono::duration<double, std::milli> ms_double = t2_eigen - t1_eigen;
22                ms_eigen = std::min(ms_eigen, ms_double.count());
23            }
24
25            // Measure runtime of Sub-problem (4-3.c)
26            double ms_mine = std::numeric_limits<double>::max();
27            for (int r = 0; r < n_runs; r++) {
28                auto t1_mine = std::chrono::high_resolution_clock::now();
29                const Eigen::SparseMatrix<double> AH_mine = buildAH(Ah, P);
30                auto t2_mine = std::chrono::high_resolution_clock::now();
31                std::chrono::duration<double, std::milli> ms_double = t2_mine - t1_mine;
32                ms_mine = std::min(ms_mine, ms_double.count());
33            }
34
35            std::cout << M << "\t" << ms_eigen << "\t\t" << ms_mine << std::endl;
36        }
37    #else
38        //
39        // *****
40        // To be supplemented
41        // *****
42    #endif
43 }
```

SOLUTION of (4-4.a):



$$\mathcal{S}(i) = \mathcal{S}(i)^* = \{\bullet\}$$

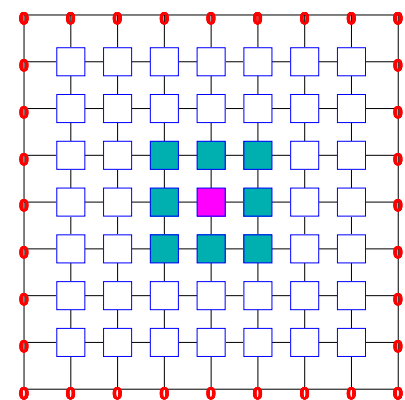
All off-diagonal entries of \mathbf{A} are either 0 or $-\frac{1}{8}$. From [Lecture \rightarrow Eq. (4.3.3.3)] it is immediate that two nodes (i, j) will be strongly coupled, if and only if $(\mathbf{A})_{ij} \neq 0$.

Thus, the set $\mathcal{S}(i)$ of nodes from which i can receive values during prolongation comprises all eight immediate neighboring nodes in the grid.

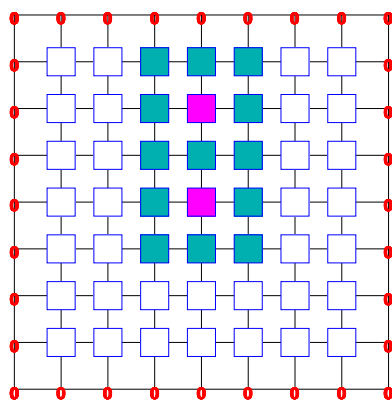
The set $\mathcal{S}(i)^*$ of nodes, to which i can distribute values during prolongation is the same.

SOLUTION of (4-4.b):

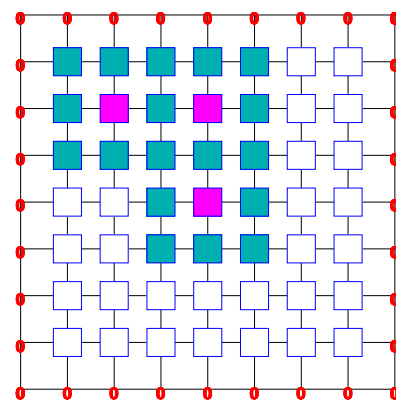
Note that the algorithm of [Lecture → Code 4.3.3.12] is not defined completely in this case, because the selection of the next C-node in Line 8 may not be unique. This leads to several different execution paths, but the final C/F splitting will always be the same.



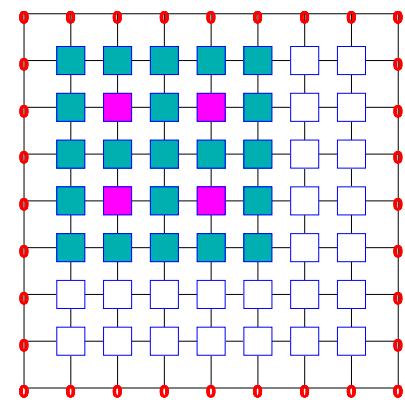
While-loop executed once



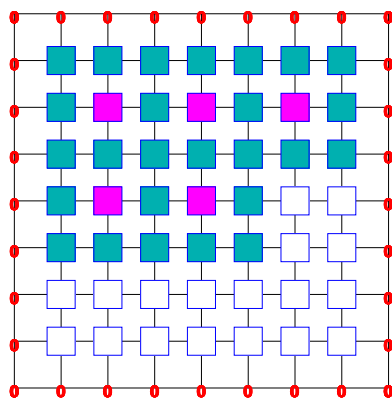
While-loop executed twice



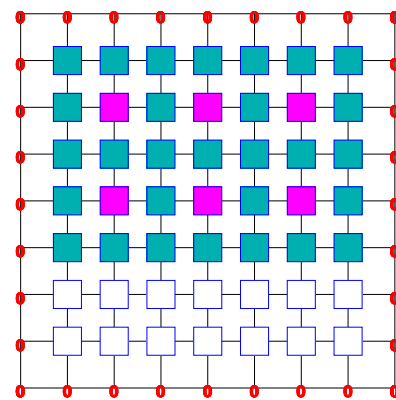
While-loop executed thrice



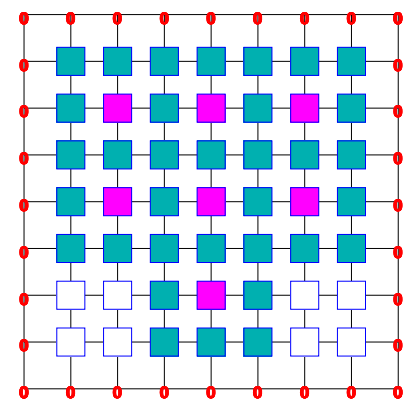
While-loop executed 4×



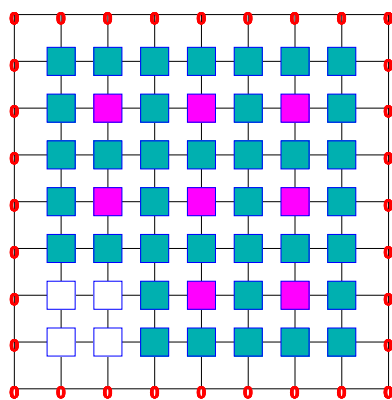
While-loop executed 5×



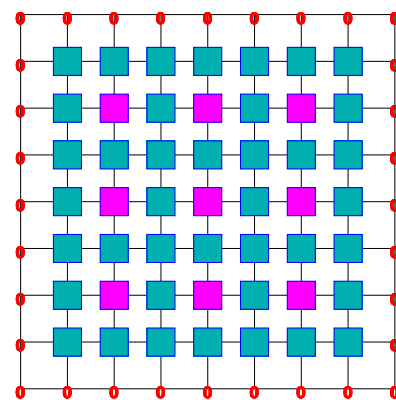
While-loop executed 6×



While-loop executed 7×



While-loop executed 8×



While-loop executed 9×

The C-nodes end up in locations that correspond to the nodes of a “geometric coarse grid”.

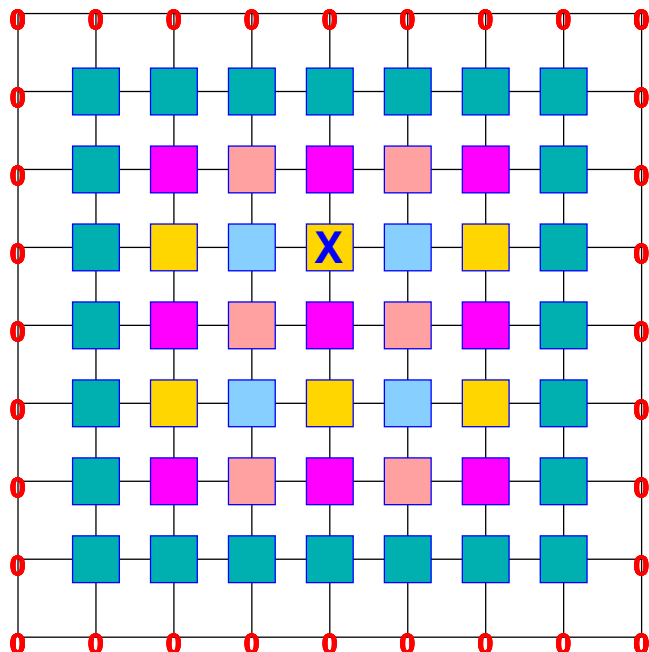
SOLUTION of (4-4.c):

We identify the nodes by their position index:

$$\text{node at } \begin{bmatrix} ih \\ jh \end{bmatrix} \leftrightarrow \text{position index } (i, j), \quad i, j \in \{1, \dots, 7\}.$$

We also use these position indices for coefficient vectors = grid functions, e.g., $\eta_{i,j} := (\vec{\eta})_{i,j}$, $\text{vec}\eta \in \mathbb{R}^{49}$.

① We first determine the direct interpolation from C-node neighbors according to [Lecture \rightarrow Eq. (4.3.4.9)].



Three different classes of F-nodes can be distinguished, marked with three different colors in the figure beside.

■:

$$(\vec{\eta}')_{i,j} := \frac{1}{2} \left((\vec{\eta})_{i+1,j} + (\vec{\eta})_{i-1,j} \right), \quad i \text{ even}, j \text{ odd},$$

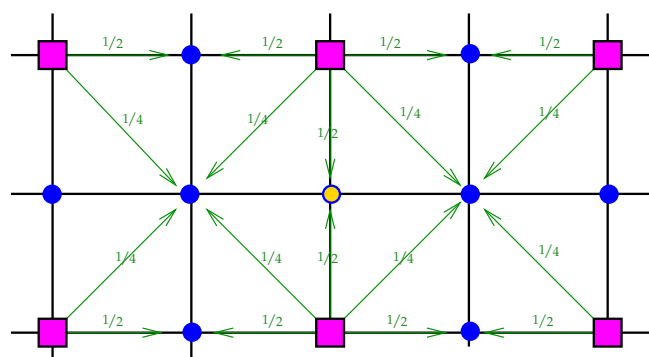
■:

$$(\vec{\eta}')_{i,j} := \frac{1}{2} \left((\vec{\eta})_{i,j+1} + (\vec{\eta})_{i,j-1} \right), \quad i \text{ odd}, j \text{ even},$$

■:

$$(\vec{\eta}')_{i,j} := \frac{1}{4} \left((\vec{\eta})_{i+1,j+1} + (\vec{\eta})_{i+1,j-1} + (\vec{\eta})_{i-1,j+1} + (\vec{\eta})_{i-1,j-1} \right), \quad i, j \text{ even}.$$

The particular node at $\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2}+h \end{bmatrix}$ belongs to the first class ■.



◁ Interpolation into F-nodes from adjacent C-nodes with local preservation of constants taken into account.

- ■ $\hat{=}$ C-node
- ● $\hat{=}$ F-node
- \rightarrow $\hat{=}$ weighted flow of information during interpolation from C-nodes

② The preliminary interpolation formula [Lecture \rightarrow Eq. (4.3.4.7)] for the F-node (4,5) (single index ℓ) at

$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2}+h \end{bmatrix}$ is

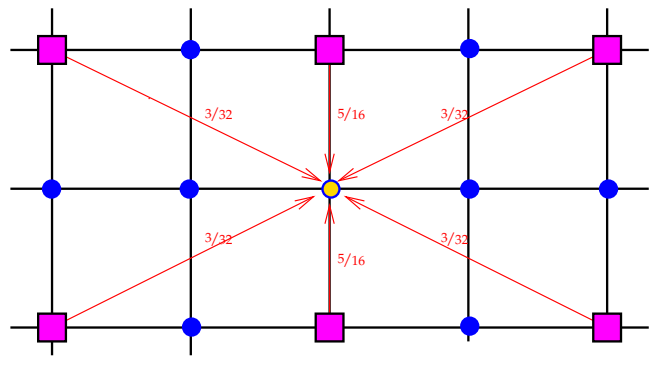
$$\begin{aligned}
 (\mathbf{P}\vec{\eta})_\ell &:= \frac{1}{8} (\eta'_{4+1,5} + \eta'_{4-1,5} + \eta'_{4,5+1} + \eta'_{4,5-1} + \eta'_{4+1,5+1} + \eta'_{4+1,5-1} + \eta'_{4-1,5+1} + \eta'_{4-1,5-1}) \\
 &= \frac{1}{8} \left(\frac{1}{4} (\eta_{4,6} + \eta_{4,4} + \eta_{6,6} + \eta_{6,4}) + \eta_{4,6} + \eta_{4,4} + \right. \\
 &\quad \left. \frac{1}{4} (\eta_{4,6} + \eta_{4,4} + \eta_{2,6} + \eta_{2,4}) + \frac{1}{2} (\eta_{2,4} + \eta_{4,4}) + \frac{1}{2} (\eta_{6,4} + \eta_{4,4}) + \right. \\
 &\quad \left. \frac{1}{2} (\eta_{2,6} + \eta_{4,6}) + \frac{1}{2} (\eta_{6,6} + \eta_{4,6}) \right) \\
 &= \frac{5}{16} \eta_{4,4} + \frac{5}{16} \eta_{4,6} + \frac{3}{32} \eta_{6,6} + \frac{3}{32} \eta_{6,4} + \frac{3}{32} \eta_{2,6} + \frac{3}{32} \eta_{2,4} .
 \end{aligned}$$

This is [Lecture → Eq. (4.3.4.10)]. Since the sum of the prolongation weights is 1 already, no re-weighting is necessary.

The figure illustrates the flow of information to the central F-node during prolongation. ▷

In stencil notation this could be written as

$$\begin{bmatrix} \frac{3}{32} & 0 & \frac{5}{16} & 0 & \frac{3}{32} \\ 0 & 0 & 0 & 0 & 0 \\ \frac{3}{32} & 0 & \frac{5}{16} & 0 & \frac{3}{32} \end{bmatrix}_h .$$



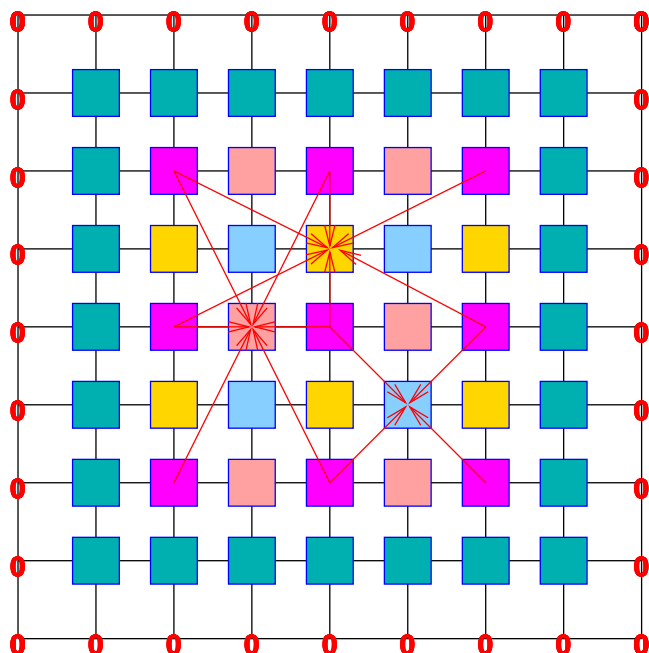
SOLUTION of (4-4.d):

Recall the Galerin construction of the coarse grid matrix \mathbf{A}_H :

$$(\mathbf{A}_H)_{i,j} = a(b_H^j, b_H^i) = \sum_{\ell=1}^N \sum_{k=1}^N a((\mathbf{P})_{\ell,j} b_h^\ell, (\mathbf{P})_{k,i} b_h^k) = \sum_{\ell=1}^N \sum_{k=1}^N (\mathbf{P})_{\ell,j} (\mathbf{P})_{k,i} a(b_h^\ell, b_h^k) = (\mathbf{P}^\top \mathbf{A} \mathbf{P})_{i,j}.$$

Thus, two C-nodes i and j , $i, j \in \mathcal{C}$, $i \neq j$, define an edge in the matrix graph of the coarse grid matrix \mathbf{A}_H , if there is a 3-edge path in the complete graph with node set $\{1, \dots, N\}$,

- whose first edge is an edge in the matrix graph of \mathbf{P} ,
- whose second edge is an edge in the matrix graph of \mathbf{A} , and
- whose last edge is an edge in the in the matrix graph of \mathbf{P} , again.



The figure visualizes edges in the matrix graph of \mathbf{P} by connecting “sending” C-nodes with “receiving” F-nodes by arrows \rightarrow .

The edges are only shown for a single F-node for each class. By symmetry the arrows for the remaining F-nodes can be inferred.

In the matrix graph of \mathbf{A} immediately adjacent nodes are connected.

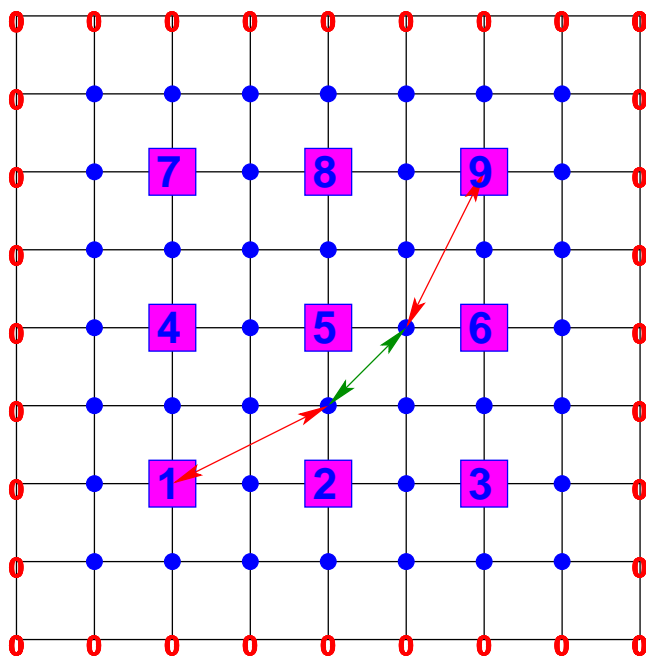
We find that

the matrix \mathbf{A}_H is fully populated!

Even the remotest C-nodes can be connected by a 3-path as described above. \triangleright

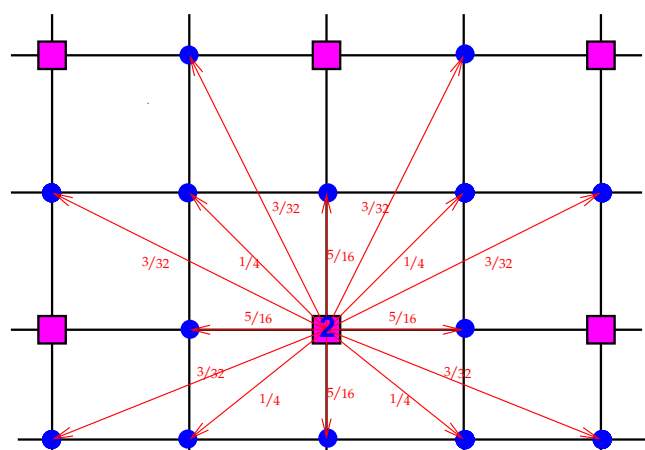
($\leftrightarrow \hat{=}$ edge in matrix graph of \mathbf{P} , $\leftrightarrow \hat{=}$ edge in matrix graph of \mathbf{A})

Here we confront the **AMG fill-in challenge** [Lecture \rightarrow Rem. 4.3.1.15].





SOLUTION of (4-4.e):



◁ Visualization of the prolongation of the value of C-node 2.

Target F-nodes are connected to 2 by \rightarrow , the numbers give the corresponding prolongation weights.

To begin with, **P** couples C-nodes to themselves with weight 1. This yields a contribution of 1 to $(\mathbf{A}_H)_{2,2}$.

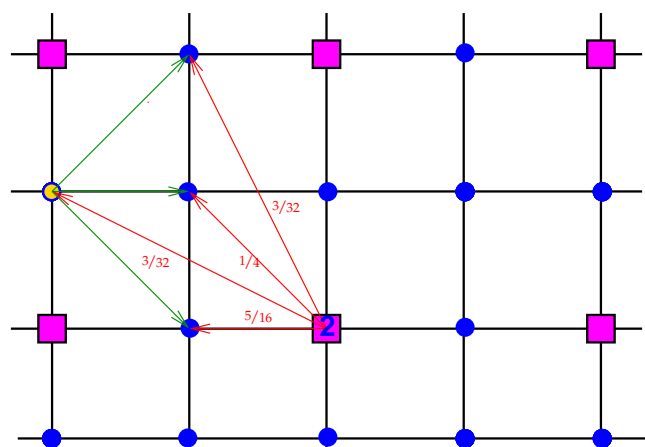
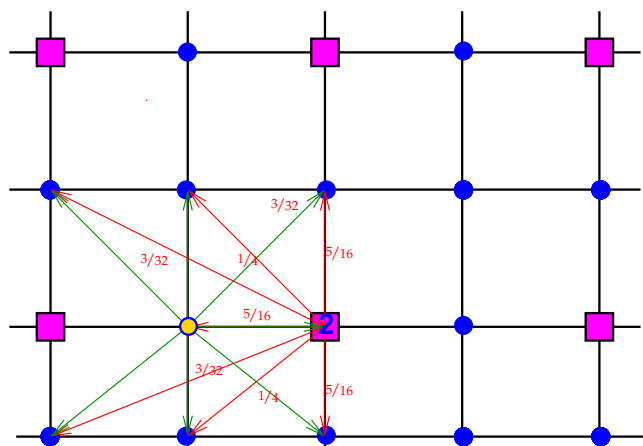
The “**P**-connections” shown in Fig. 42 correspond to 2-paths contributing to $(\mathbf{A})_{2,2}$. In fact, these are 3-paths, for which the edge from the matrix graph of **A** is a loop (corresponding to a non-zero diagonal entry 1 of **A**). The 2-path contributions are

$$6 \cdot \left(\frac{3}{32}\right)^2 + 4 \cdot \left(\frac{1}{4}\right)^2 + 4 \cdot \left(\frac{5}{16}\right)^2 = \frac{710}{1024}.$$

Next, we examine the 3-path contributions for every F-node linked to C-node 2 via an edge of the matrix graph of **P**, see Fig. 42. All these contributions have to be added up. Let us single out the F-node to the left of C-node 2.

In the edge graph of **A** it is connected to 8 other nodes with weight $-\frac{1}{8}$ (arrow \rightarrow). So the contribution to $(\mathbf{A})_{2,2}$ is

$$-\frac{5}{16} \cdot \frac{1}{8} \left(1 + 2 \cdot \frac{3}{32} + 2 \cdot \frac{1}{4} + 2 \cdot \frac{5}{16}\right).$$



Next, the 3-path contribution of the F-node at a distance $\begin{bmatrix} -2h \\ h \end{bmatrix}$ is

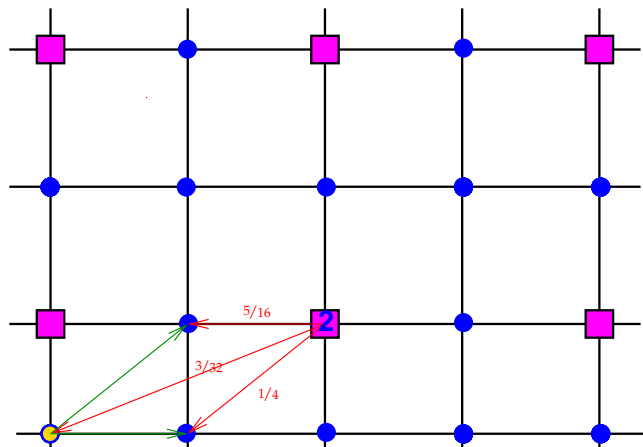
$$-\frac{3}{32} \cdot \frac{1}{8} \left(\frac{1}{4} + \frac{5}{16} + \frac{3}{32}\right).$$

By symmetry the F-node $\begin{bmatrix} 2h \\ h \end{bmatrix}$ away contributes the same value.

The impact of the boundary has to be taken into account when computing the contribution of the node at a distance $\begin{bmatrix} -2h \\ -h \end{bmatrix}$.

$$-\frac{3}{32} \cdot \frac{1}{8} \left(\frac{1}{4} + \frac{5}{16} \right).$$

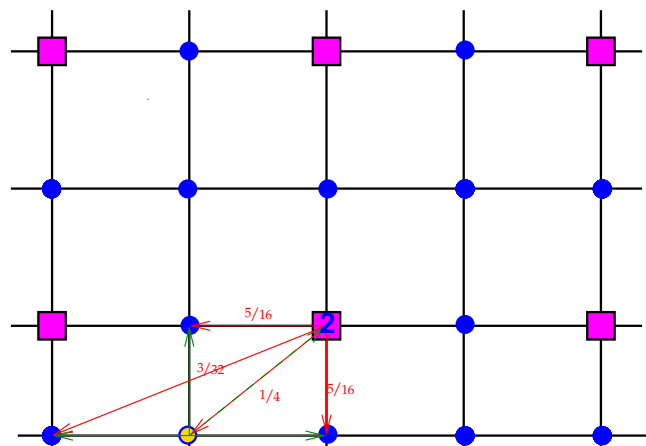
On the other side there is a second node with that contribution.



Since it is located next to the boundary, the F-node located at distance $\begin{bmatrix} -h \\ -h \end{bmatrix}$ contributes

$$-\frac{1}{4} \cdot \frac{1}{8} \left(1 + 2 \cdot \frac{5}{16} + \frac{3}{32} \right).$$

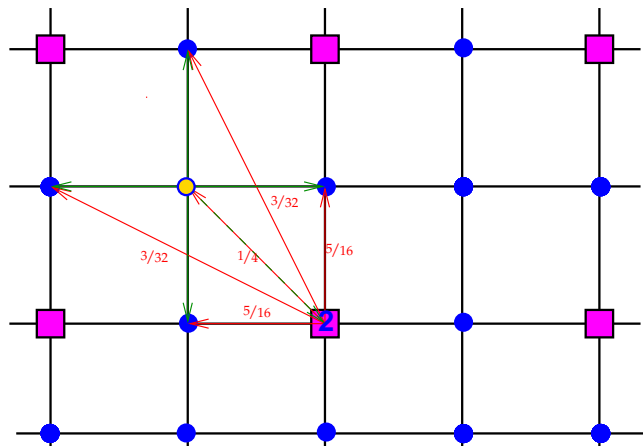
By symmetry there is another F-node with that contribution.



The F-node at a distance $\begin{bmatrix} -h \\ h \end{bmatrix}$ adds

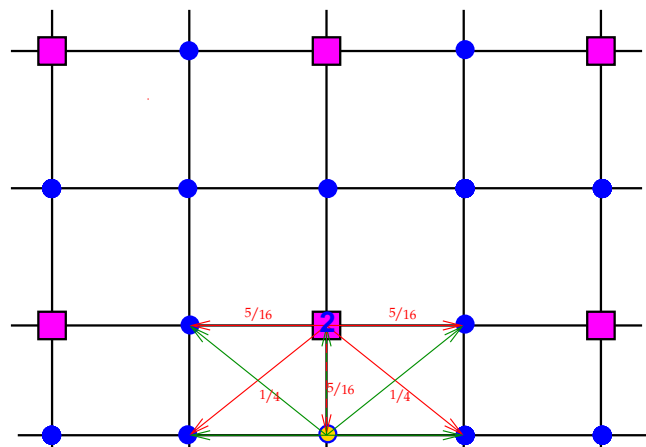
$$-\frac{1}{4} \cdot \frac{1}{8} \left(1 + 2 \cdot \frac{5}{16} + 2 \cdot \frac{3}{32} \right).$$

There is another node of this kind.



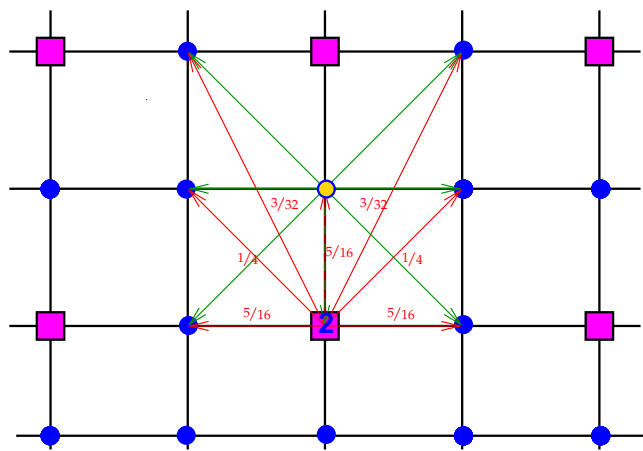
The F-node right below C-node 2 contributes

$$-\frac{5}{16} \cdot \frac{1}{8} \left(1 + 2 \cdot \frac{1}{4} + 2 \cdot \frac{5}{16} \right).$$



The F-node right above C-node 2 contributes

$$-\frac{5}{16} \cdot \frac{1}{8} \left(1 + 2 \cdot \frac{1}{4} + 2 \cdot \frac{5}{16} + 2 \cdot \frac{3}{32} \right).$$



The resulting value is $(\mathbf{A}_H)_{2,2} = 1.090087890625000$.

SOLUTION of (4-5.):

The AMG iteration is a stationary linear iteration so that the algorithm of [Lecture → Code 4.1.3.22] applies. We call the `RugeStuebenAMG::iterate()` member function with r.h.s. vector = `0`.

C++ code 4.5.14: Implementation of `cvgRateAMG()` Get it on [GitLab \(cfsplit.cpp\)](#).

```
2 // TODO : Check if power iteration was to be implemented or something
3 // different
4 double cvgRateAMG(const Eigen::SparseMatrix<double> &A, double tol) {
5     double rate;
6     const unsigned N = A.rows();
7     Eigen::VectorXd rhs = Eigen::VectorXd::Constant(N, 0.0);
8     Eigen::VectorXd v = Eigen::VectorXd::Random(N);
9     Eigen::VectorXd residual;
10    const double tauC = 0.25;
11    const double sigma = 1.5;
12    RugeStuebenAMG M(A, 10, tauC, sigma);
13    const unsigned int max_iter = 1000;
14    unsigned int iter = 0;
15    Eigen::VectorXd v_tmp;
16    double lambda_old;
17    double lambda_new = v.norm();
18    do {
19        lambda_old = lambda_new;
20        v /= v.norm();
21        M.iterate(rhs, v);
22        lambda_new = v.norm();
23        std::cout << lambda_new << std::endl;
24        ++iter;
25    } while (std::abs(lambda_new - lambda_old) > tol * std::abs(lambda_new) &&
26            iter < max_iter);
27
28    if (iter == max_iter) {
29        std::cout << "Maximum Iterations were Reached" << std::endl;
30    }
31    rate = lambda_new;
32    std::cout << "Iteration : " << iter << " rate : " << rate << std::endl;
33
34    return rate;
35 }
```

SOLUTION of (4-5.c):

1. We first build the array S_i for the sets $\mathcal{S}(i)$ by looping through the rows of the matrix:

$$\begin{aligned}\mathcal{S}(i) &:= \{j \in \{1, \dots, N\} : (i, j) \text{ is strongly coupled}\} \\ &:= \left\{j \in \{1, \dots, N\} : -(\mathbf{A})_{ij} \geq \tau_C \max\left\{ |(\mathbf{A})_{ik}| : (\mathbf{A})_{ik} < 0, k \in \{1, \dots, N\} \right\}\right\}.\end{aligned}\tag{4.5.4}$$

We may first compute $\max\left\{ |(\mathbf{A})_{ik}| : (\mathbf{A})_{ik} < 0, k \in \{1, \dots, N\} \right\}$ for every index i and then run through the rows of the matrix again to determine the elements of $\mathcal{S}(i)$. This approach has the advantage that it will work for both row-major and column-major sparse matrices.

2. Then we initialize the array S_{j_star} based on S_i .

$$\mathcal{S}(j)^* := \{i \in \{1, \dots, N\} : j \in \mathcal{S}(i)\},\tag{4.5.5}$$

that is, \mathcal{S}^* is the transposed relation of \mathcal{S} .

C++ code 4.5.6: Implementation of `RugeStuebenAMG::getStrongConn()`
Get it on [GitLab \(cfsplit.cpp\)](#).

```
2 std::pair<std::vector<std::vector<RugeStuebenAMG::nodeidx>>,
3     std::vector<std::vector<RugeStuebenAMG::nodeidx>>>
4 RugeStuebenAMG::getStrongConn(const Eigen::SparseMatrix<double> &A,
5     double tauC) const {
6     const unsigned N = A.cols();
7     int i;
8     int j;
9     std::vector<double> max_values(N, 0);
10    std::vector<std::vector<RugeStuebenAMG::nodeidx>> Si(N);
11    std::vector<std::vector<RugeStuebenAMG::nodeidx>> Sj_star(N);
12    // Loop once over the nnz entries of A to compute the set of maximum
13    // values
14    for (int k = 0; k < A.outerSize(); ++k) {
15        for (Eigen::SparseMatrix<double>::InnerIterator it(A, k); it; ++it) {
16            i = it.row(); // row index
17            j = it.col(); // col index
18
19            if (i == j) { //ignore diagonal entries
20                continue;
21            }
22            if (max_values[i] < std::abs(it.value()) &&
23                it.value() < 0) { // Update the maximal value
24                max_values[i] = it.value();
25            }
26        }
27    }
28    // Loop again over the matrix A to check for strong connectivity
29    for (int k = 0; k < A.outerSize(); ++k) {
30        for (Eigen::SparseMatrix<double>::InnerIterator it(A, k); it; ++it) {
31            i = it.row(); // row index
32            j = it.col(); // col index
33
34            if (i == j) { //ignore diagonal entries
```

```
35     continue;
36 }
37 if (tauC * max_values[i] <=
38     -it.value()) { // add the node to the strongly connected list
39     Si[i].push_back(j);
40 }
41 }
42 }
43 // Loop over Si to find Sj*
44 for (unsigned int i = 0; i < N; ++i) {
45     for (auto j : Si[i]) {
46         Sj_star[j].push_back(i);
47     }
48 }
49 return std::make_pair(Si, Sj_star);
50 }
```


SOLUTION of (4-5.d):

We simply traverse all rows of **A** and check [Lecture → Eq. (4.3.3.9)]. The following implementation works for both row-major and column-major format of the sparse matrix.

C++ code 4.5.7: Implementation of `RugeStuebenAMG::setSmoothable()`
Get it on [👉 GitLab \(cfsplit.cpp\)](#).

```
2 void RugeStuebenAMG::setSmoothable(const Eigen::SparseMatrix<double> &A,  
3                                   double sigma,  
4                                   std::vector<NodeFlag> &nodeflags) const {  
5     const unsigned int N = A.cols();  
6     int i;  
7     int j;  
8     std::vector<double> diagonal_values(N, 0);  
9     std::vector<double> offdiagonal_sum(N, 0);  
10    // Loop once over the nnz entries of A  
11    for (int k = 0; k < A.outerSize(); ++k) {  
12        for (Eigen::SparseMatrix<double>::InnerIterator it(A, k); it; ++it) {  
13            i = it.row(); // row index  
14            j = it.col(); // col index  
  
15  
16            if (i == j) { // diagonal entries  
17                diagonal_values[i] = std::abs(it.value());  
18            } else { // off diagonal values  
19                offdiagonal_sum[i] += std::abs(it.value());  
20            }  
21        }  
22    }  
23    for (unsigned int i = 0; i < N; ++i) {  
24        if (diagonal_values[i] >= sigma * offdiagonal_sum[i]) {  
25            nodeflags[i] = FINE;  
26        }  
27    }  
28 }
```

SOLUTION of (4-5.e):

Of course, an outer loop visits all nodes. The computation of $\lambda(i)$ according to [Lecture \rightarrow Eq. (4.3.3.11)] seems straightforward, since the sets $S(j)^*$ are immediately accessible.

C++ code 4.5.8: Implementation of `RugeStuebenAMG::connectivityMeasure()`

Get it on  [GitLab](#) (`cfsplit.cpp`).

```
2 std::vector<unsigned int> RugeStuebenAMG::connectivityMeasure(  
3     const std::vector<NodeFlag> &nodeflags,  
4     const std::vector<std::vector<nodeidx>> &Sj_star) const {  
5     const unsigned N = Sj_star.size();  
6     std::vector<unsigned int> sum(N, 0);  
7  
8     for (unsigned int j = 0; j < N; ++j) {  
9         for (auto i : Sj_star[j]) {  
10            if (nodeflags[i] == FINE) {  
11                sum[j] += 2;  
12            } else if (nodeflags[i] == UNDECIDED) {  
13                sum[j]++;  
14            }  
15        }  
16    }  
17    return sum;  
18 }
```

SOLUTION of (4-5.f):

The set operations in ?? are easy to realize using the flag array.

C++ code 4.5.9: Implementation of RugeStuebenAMG::CFsplit() Get it on [GitLab \(cfsplit.cpp\)](#).

```
2  std::vector<RugeStuebenAMG::NodeFlag> RugeStuebenAMG::CFsplit(  
3      const Eigen::SparseMatrix<double> &A,  
4      const std::vector<std::vector<nodeidx>> &Sj_star) const {  
5      const unsigned int N = A.cols();  
6      std::vector<NodeFlag> nodeflags(N, UNDECIDED);  
7      setSmoothable(A, sigma_, nodeflags);  
8      std::vector<unsigned int> connectivity =  
9          connectivityMeasure(nodeflags, Sj_star);  
10  
11     // Determines if the algorithm has finished  
12     auto isfinished = [&N](  
13         const std::vector<NodeFlag> &nodeflags,  
14         const std::vector<unsigned int> &connectivity) -> bool {  
15         for (unsigned int i = 0; i < N; ++i) {  
16             if (nodeflags[i] == UNDECIDED) {  
17                 if (connectivity[i] > 0) {  
18                     return false;  
19                 }  
20             }  
21         }  
22         return true;  
23     };  
24     // Determines the most suitable next Coarse node  
25     auto bestC = [&N](const std::vector<NodeFlag> &nodeflags,  
26         const std::vector<unsigned int> &connectivity) -> unsigned {  
27         unsigned bestvalue = 0;  
28         unsigned bestindex = -1;  
29         for (unsigned int i = 0; i < N; ++i) {  
30             if (nodeflags[i] == UNDECIDED) {  
31                 if (connectivity[i] > bestvalue) {  
32                     bestindex = i;  
33                     bestvalue = connectivity[i];  
34                 }  
35             }  
36         }  
37         return bestindex;  
38     };  
39  
40     while (!isfinished(nodeflags, connectivity)) {  
41         const unsigned i = bestC(nodeflags, connectivity);  
42         nodeflags[i] = COARSE;  
43         for (unsigned j : Sj_star[i]) {  
44             nodeflags[j] = FINE;  
45         }  
46         connectivity = connectivityMeasure(nodeflags, Sj_star);  
47     }  
48     // Set all remaining nodes to fine  
49     for (auto &i : nodeflags) {  
50         if (i == UNDECIDED) {
```

```
51     i = FINE;
52   }
53 }
54 return nodeflags;
55 }
```

For the sake of simplicity, this implementation sacrifices efficiency as was already remarked after Code 4.5.3. The lambda functions rely on loops from 1 to N . Since they are called in the body of the outer **while** loop, an asymptotic complexity of $O(N^2)$ will result.

SOLUTION of (4-5.g):

It is advisable to initialize the entries of \mathbf{P} through a temporary triplet format [NumCSE course \rightarrow Section 2.7.2]. In an outer loop run through all the nodes, which corresponds to building \mathbf{P} row-wise. If the current node i is a C-nodes, only a single triplet $(i, j, 1.0)$ is needed, where j is the number of the C-node i "in the coarse grid". If $i \in \mathcal{F}$ (F-node), then the entries of the i -th row of \mathbf{P} can be computed according to [Lecture \rightarrow Eq. (4.3.4.10)].

C++ code 4.5.11: Implementation of `RugeStuebenAMG::computeP()`
Get it on  [GitLab \(cfsplit.cpp\)](#).

```
2 Eigen::SparseMatrix<double, Eigen::RowMajor> RugeStuebenAMG::computeP(  
3     const Eigen::SparseMatrix<double> &A,  
4     const std::vector<std::vector<nodeidx>> &Si,  
5     const std::vector<NodeFlag> &nodeflags) const {  
6     unsigned int N = A.rows();  
7     unsigned int ncoarse = 0;  
8  
9     std::vector<Eigen::Triplet<double>> P_triplet;  
10    std::vector<double> alpha(N, 0.0);  
11    std::vector<double> alpha_hat(N, 0.0);  
12    std::vector<double> offdiagonal_sum(N, 0);  
13    std::vector<nodeidx> fineToCoarse(N, -1);  
14    for (unsigned int i = 0; i < N; ++i) {  
15        if (nodeflags[i] == COARSE) {  
16            fineToCoarse[i] = ncoarse;  
17            ncoarse++;  
18        }  
19    }  
20  
21    // Loop once over the nnz entries of A to compute the row sums  
22    for (int k = 0; k < A.outerSize(); ++k) {  
23        for (Eigen::SparseMatrix<double>::InnerIterator it(A, k); it; ++it) {  
24            int i = it.row(); // row index  
25            int j = it.col(); // col index  
26  
27            if (i == j) { //ignore diagonal entries  
28                continue;  
29            } else { // off diagonal values  
30                offdiagonal_sum[i] += it.value();  
31            }  
32        }  
33    }  
34  
35    // Compute alpha and alpha_hat and count coarse  
36    for (unsigned int i = 0; i < N; ++i) {  
37        std::vector<nodeidx> strongly_connected_outer = Si[i];  
38        for (nodeidx j : strongly_connected_outer) {  
39            alpha[i] += A.coeff(i, j);  
40            if (nodeflags[j] == COARSE) {  
41                alpha_hat[i] += A.coeff(i, j);  
42            }  
43        }  
44        alpha[i] = offdiagonal_sum[i] / alpha[i];  
45        alpha_hat[i] = offdiagonal_sum[i] / alpha_hat[i];  
46    }  
47 }
```

```

46 }
47
48 for (unsigned int i = 0; i < N; ++i) {
49     if (nodeflags[i] == COARSE) {
50         P_triplet.emplace_back(i, fineToCoarse[i], 1.0);
51     } else {
52         std::vector<nodeidx> strongly_connected_outer = Si[i];
53         for (nodeidx j : strongly_connected_outer) {
54             if (nodeflags[j] == COARSE) {
55                 P_triplet.emplace_back(i, fineToCoarse[j],
56                                     -alpha[i] * A.coeff(i, j) / A.coeff(i, i));
57             } else {
58                 std::vector<nodeidx> strongly_connected_inner = Si[j];
59                 for (nodeidx k : strongly_connected_inner) {
60                     if (nodeflags[k] == COARSE) {
61                         P_triplet.emplace_back(i, fineToCoarse[k],
62                                             alpha[i] * alpha_hat[j] * A.coeff(j, k) *
63                                             A.coeff(i, j) /
64                                             (A.coeff(i, i) * A.coeff(j, j)));
65                     }
66                 }
67             }
68         }
69     }
70 }
71
72 Eigen::SparseMatrix<double> P(N, ncoarse);
73 P.setFromTriplets(P_triplet.begin(), P_triplet.end());
74 return P;
75 }

```

SOLUTION of (4-5.h):

The implementation amounts to assembling pieces in the form of the member functions `getStrongConn`, `SFsplit()`, and `computeP()` and going from the fine grid to the coarser grids sequentially.

C++ code 4.5.12: Implementation of [Get it on GitLab \(cfsplit.h\)](#).

```
2  template <typename RECORDER>
3  RugeStuebenAMG::RugeStuebenAMG(const Eigen::SparseMatrix<double> &A,
4                                unsigned int min_mat_size, double tauC,
5                                double sigma, RECORDER &&rec)
6      : tauC_(tauC), sigma_(sigma) {
7      Amats_.push_back(A);
8      L_ = 0;
9      std::vector<std::vector<nodeidx>> Si;
10     std::vector<std::vector<nodeidx>> Sj_star;
11     std::vector<NodeFlag> nodeflags;
12     while (Amats_.back().cols() > min_mat_size) {
13         Si.clear();
14         Sj_star.clear();
15         nodeflags.clear();
16         std::tie(Si, Sj_star) = getStrongConn(Amats_.back(), tauC_);
17         nodeflags = CFsplit(Amats_.back(), Sj_star);
18         Eigen::SparseMatrix<double> P = computeP(Amats_.back(), Si, nodeflags);
19         if (P.cols() == 0) { // Check if there are any suitable coarse nodes
20             break;
21         }
22         Pmats_.push_back(computeP(Amats_.back(), Si, nodeflags));
23         Amats_.push_back(
24             GalerkinConstruction::buildAH_eigen(Amats_.back(), Pmats_.back()));
25         rec(Amats_.back(), Pmats_.back(), nodeflags);
26         L_++;
27     }
28     std::cout << L_ << std::endl;
29     CoarseLU_.analyzePattern(Amats_.back());
30     CoarseLU_.factorize(Amats_.back());
31     if (CoarseLU_.info() != Eigen::Success) {
32         throw std::runtime_error("Did not manage to factorize A at coarsest level");
33     }
34 }
35 }
```

SOLUTION of (4-5.i):

We simply implement the multigrid V-cycle as outlined in [Lecture → Code 4.2.5.7]. The setup phase has provided all required matrices.

Of course, for the sake of efficiency, we have to use EIGEN's facilities for traversing the non-zero elements in a row of a sparse matrix when doing Gauss-Seidel smoothing.

C++ code 4.5.13: Implementation of `RugeStuebenAMG::iterate()`
Get it on [🐱 GitLab \(cfsplit.cpp\)](#).

```
2 void RugeStuebenAMG::iterate(const Eigen::VectorXd &phi ,
3                             Eigen::VectorXd &mu) const {
4     std::function<void(const Eigen::VectorXd &, Eigen::VectorXd &, unsigned)>
5     recursive_iteration =
6     [&](const Eigen::VectorXd &phi , Eigen::VectorXd &mu, unsigned index) {
7         if (index == L_) {
8             mu = CoarseLU_.solve(phi);
9             return;
10        }
11        // Pre-smoothing
12        mu += Amats_[index].triangularView<Eigen::Lower>().solve(
13            phi - Amats_[index] * mu);
14        //Restriction
15        Eigen::VectorXd restricted_phi =
16            Pmats_[index].transpose() * (phi - Amats_[index] * mu);
17        Eigen::VectorXd restricted_mu = 0 * restricted_phi;
18        // Recursive call
19        recursive_iteration(restricted_phi, restricted_mu, index + 1);
20        // Prolongation
21        mu += Pmats_[index] * restricted_mu;
22        // Post-smoothing
23        mu += Amats_[index].triangularView<Eigen::Upper>().solve(
24            phi - Amats_[index] * mu);
25    };
26    recursive_iteration(phi, mu, 0);
27 }
```