Ex2/HermiteBandMatrix.h

```cpp
#include <iostream>
#include <complex>
#include <type_traits>
#include <iomanip>
#include "SCmatrix.h"

namespace SC
{
    template <typename T = double>
    class HermiteBandMatrix : public LinearOperator<T>
    {
    private:
        int n;   // nxn-Matrix
        int b;   // Bandwidth
        T *data; // Bandvalues
        // int alloc_size;

    public:
        using LinearOperator<T>::height;
        using LinearOperator<T>::width;

        /// @brief Constructor
        /// @param n Dimension of n*n-square matrix
        /// @param b Width of band
        HermiteBandMatrix(int n, int b) : n(n), b(b), data(nullptr) //, alloc_size(0)
        {
            int num_elem = n * (n + 1) / 2 - (n - b) * (n - b + 1) / 2; // Maximum allowed storage space
            data = new T[num_elem];
            // alloc_size = num_elem;
            this->height = n;
            this->width = n;
        }

        /// @brief Destructor
        ~HermiteBandMatrix()
        {
            delete[] data;
        }

        /// @brief Copy-Constructor
        /// @param other Object to copy
        HermiteBandMatrix(const HermiteBandMatrix<T> &other) : n(other.n), b(other.b), data(nullptr)
        {
            this->height = other.height;
            this->width = other.width;

            int num_elem = n * (n + 1) / 2 - (n - b) * (n - b + 1) / 2; // Maximum allowed storage space
            data = new T[num_elem];

            for (int i = 0; i < num_elem; ++i)
            {
                data[i] = other.data[i];
            }
        }

        /// @brief Copy-Constructor for disabling operation
        // HermiteBandMatrix(const HermiteBandMatrix<T>&) = delete;

        /// @brief Sets the given value in matrix at (i, j)
        /// @param i Row index
        /// @param j Column index
        /// @param val Value to store at position
        void Set(int i, int j, T val)
        {

            int idx = index(i, j);
            if (idx < 0)
            {
#ifndef NDEBUG
                throw std::out_of_range("Index exceeds matrix dimensions");
#endif
            }

            if (is_swapped(i, j))
            {
                data[idx] = Conjugate(val);
            }
            else
            {
                data[idx] = val;
            }
        }

        /// @brief Checks if given coordinate (i, j) is below the diagonal
        /// @param i Row index
        /// @param j Column index
        /// @return Boolean if index is below diagonal
        bool is_swapped(int i, int j) const
        {
            return j < i;
        }

        /// @brief Calculates the list index of given matrix coordinates (i, j)
        /// @param i Row index
        /// @param j Column index
        /// @return Index in flat data list
        int index(int i, int j) const
        {
            // Check if index is out of bound
            if (i >= n || i < 0 || j >= n || j < 0)
            {
                return -1;
            }

            // Da obere dreiecksstruktur betrachtet
            if (is_swapped(i, j))
                std::swap(i, j);

            int idx = 0;
            // n - b = Anzahl der "ganzen" Zeilen
            if (i <= n - b)
            {
                idx = i * b + (j - i);
            }
            else
            {
                int last_full_idx = (n - b + 1) * b;
                int rel_idx = 0;
                // k = relativer (zeilen) laufindex
                for (int k = n - b + 1; k < i; k++)
                {
                    rel_idx += (n - k);
                }
                rel_idx += (j - i);
                idx = last_full_idx + rel_idx;
            }
            return idx;
        }

        /// @brief Operator for reading data from matrix
        /// @param i Row index
        /// @param j Column index
        /// @return Entry of matrix at given (i, j)
        T operator()(int i, int j) const
        {
            // b inkludiert diagonale -> (b - 1)
            if (j < i - b + 1 || j > i + b - 1)
            {
                return T(0);
            }

            int idx = index(i, j);
            // Check if index is out of bounds
            if (idx < 0)
            {
#ifndef NDEBUG
                throw std::out_of_range("Index exceeds matrix dimensions");
#endif
                return T(0);
            }
            if (is_swapped(i, j))
            {
                return Conjugate(data[idx]);
            }

            return data[idx];
        }

        /// @brief Calculates the matrix product
        /// @param a Applied vector
        /// @param r Resulting vector
        /// @param factor Linear scaling factor; Default = 1
        virtual void Apply(const Vector<T> &a, Vector<T> &r, T factor = 1.) const override
        {
            for (int row = 0; row < this->height; row++)
            {
                int b_right = this->width - row;
                int b_left = row;
                if (row > b - 1)
                {
                    b_left = b - 1;
                }
                if (row < this->width - b)
                {
                    b_right = b;
                }

                r(row) = 0;
                for (int col = row - b_left; col < row + b_right; col++)
                {
                    r(row) += operator()(row, col) * a(col);
                }
                r(row) *= factor;
            }
        }

        /// @brief Calculates the the matrix product of A.T*x
        /// @param a Applied vector
        /// @param r Resulting vector
        /// @param factor Linear scaling factor; Default = 1
        virtual void ApplyT(const Vector<T> &a, Vector<T> &r, T factor = 1.) const override
        {
            for (int row = 0; row < this->height; row++)
            {
                int b_right = this->width - row;
                int b_left = row;
                if (row > b - 1)
                {
                    b_left = b - 1;
                }
                if (row < this->width - b)
                {
                    b_right = b;
                }

                r(row) = 0;
                for (int col = row - b_left; col < row + b_right; col++)
                {
                    r(row) += operator()(col, row) * a(col);
                }
                r(row) *= factor;
            }
        }

        /// @brief Calculates the matrix hermitian product with a
        /// @param a Applied vector
        /// @param result Resulting vector of A.H*x
        /// @param factor Linear scaling factor; Default = 1
        virtual void ApplyH(const Vector<T> &a, Vector<T> &result, T factor = 1.) const override
        {
            // Hermitesche Matrix ist gleich ihrer adjungierten (transponiert-konjugierten) Matrix
            Apply(a, result, factor);
        }

        /// @brief Prints the calling hermite band matrix
        /// @param os Output stream
        virtual void Print(std::ostream &os) const
        {
            os << "[HermiteBandMatrix, size " << this->height << " x " << this->width << ", bandwidth " << b << "]\n";
            for (int row = 0; row < this->height; row++)
            {
                os << "|";
                for (int col = 0; col < this->width; col++)
                {
                    os << std::setw(8) << (*this)(row, col) << std::setw(8);
                }
                os << std::setw(4) << "|\n";
            }
        }
    };
}

```