Program Title: Nevanlinna
Programming Language: C++
Dependency: Eigen3 and GMP libraries

- Prepare a file for input parameters. Our example needs the file name of the Matsubara Green's function data (ifile), the number of Matsubara points (imag_num) and where to output spectral function (ofile).
- Prepare the Matsubara Green's function data file (in the format of 'frequency real_part imag_part\n' with increasing positive Matsubara frequencies)
- Change the real grid descretization as needed, including the minimum and maximum frequency, number of discretized points and eta *i.e.* η (evaluation axis is $\omega + i\eta$) in Listing 5 line number 74.
- Change output from $A(\omega)$ to $G^R = -\mathcal{N}\mathcal{G}(\omega + i\eta)$ or else as needed in Listing 5 line number 95.
- Change calculation precision in Listing 3 line number 10 as needed. A typical sufficient precision for Schur algorithm is 128.
- Can output the ultimate $\{a(z), b(z), c(z), d(z)\}$ in Listing 4 line number 92 for convenience of calculating the functional norm during optimization, without the need to rerun this program.
- This program can also be used to evaluate $A(\omega)$ with an optimized θ_{M+1}. Change the constant 0 θ_{M+1} in Listing 4 line number 90 to the formula for your $\theta_{M+1}(z)$.
- Compile the program with gnu c++ compiler and run the executable ./nevanlinna with input redirection.

Listing 1: compile and run the program

```
g++ -o nevanlinna nevanlinna.cpp -I path/to/eigen3 -lgmp -lgmpxx
./nevanlinna < input.txt
```

Listing 2: input.txt

```
ifile imag_num ofile
```

Listing 3: nevanlinna.cpp

```
1  #include "schur.h"
2
3
4  int main (int argc, char * argv[]) {
5      std::string ifile, ofile;
6      int imag_num;
7      //prompt user for input parameters
8      std::cin >> ifile >> imag_num >> ofile;
9      //set calculation precision
10     mpf_set_default_prec(128);
11     //begin evaluation
12     Schur<mpf_class> NG(ifile, imag_num, ofile);
13     NG.evaluation();
14     return 0;
15 }
```

Listing 4: schur.h

```
1  #include "nevanlinna.h"
2
3
4  template <class T>
5  class Schur : precision_<T> {
6  private:
7      using typename precision_<T>::nev_complex;
8      using typename precision_<T>::nev_complex_vector;
9      using typename precision_<T>::nev_complex_matrix;
10     using typename precision_<T>::nev_complex_matrix_vector;
11 public:
12     //check Nevanlinna/contractive interpolant existence condition
13     Schur (std::string ifile, int imag_num, std::string ofile);
14     //evaluation with 0 parametric function
15     void evaluation ();
16 private:
17     int M; //number of Matsubara points
18     imag_domain_data <T> imag; //theta values at Matsubara points (G -> NG -> theta)
19     real_domain_data <T> real; //real frequency NG storage, at omega + i*eta
20     nev_complex_vector phis; //phi_1 to phi_M
21     nev_complex_matrix_vector abcds; //intermediate {a, b, c, d}s used to calculate phis
22     //memoize intermediate abcds and calculate phis by iteration
23     void core ();
24 };
```

```


template <class T>
Schur<T>::Schur (std::string ifile, int imag_num, std::string ofile) : imag(ifile, imag_num), real(ofile)  {
    M = imag_num;
    //fill the Pick matrix
    nev_complex_matrix Pick (M, M);
    nev_complex I {0., 1.};
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            nev_complex freq_i = (imag.freq()[i] - I) / (imag.freq()[i] + I);
            nev_complex freq_j = (imag.freq()[j] - I) / (imag.freq()[j] + I);
            nev_complex one {1., 0.};
            nev_complex nom = one - imag.val()[i] * std::conj(imag.val()[j]);
            nev_complex den = one - freq_i * std::conj(freq_j);
            Pick(i, j) = nom / den;
        }
    }
    //check the positive semi-definiteness of the Pick matrix using Cholesky decomposition
    Eigen::LLT<nev_complex_matrix> lltOfPick(Pick + nev_complex_matrix::Identity(M, M) * 1e-250);
    if(lltOfPick.info() == Eigen::NumericalIssue)
        std::cerr << "Pick matrix is non positive semi-definite matrix in Schur method." << std::endl;
    else std::cerr << "Pick matrix is positive semi-definite." << std::endl;
}


template <class T>
void Schur<T>::core() {
    phis.resize(M);
    abcds.resize(M);
    phis[0] = imag.val()[0];
    for (int k = 0; k < M; k++) abcds[k] = nev_complex_matrix::Identity(2, 2);
    for (int j = 0; j < M - 1; j++) {
        for (int k = j; k < M; k++) {
            nev_complex_matrix prod(2, 2);
            prod(0, 0) = (imag.freq()[k] - imag.freq()[j]) / (imag.freq()[k] - std::conj(imag.freq()[j]));
            prod(0, 1) = phis[j];
            prod(1, 0) = std::conj(phis[j])*
                        ((imag.freq()[k] - imag.freq()[j]) / (imag.freq()[k] - std::conj(imag.freq()[j])));
            prod(1, 1) = nev_complex{1., 0.};
            abcds[k] *= prod;
        }
        phis[j + 1] = (- abcds[j + 1](1, 1) * imag.val()[j + 1] + abcds[j + 1](0, 1)) /
                        (abcds[j + 1](1, 0) * imag.val()[j + 1] - abcds[j + 1](0, 0));
    }
}


template <class T>
void Schur<T>::evaluation () {
    core();
    nev_complex I {0., 1.};
    nev_complex One {1., 0.};
    for (int i = 0; i < real.N_real(); i++) {
        nev_complex_matrix result = nev_complex_matrix::Identity(2, 2);
        nev_complex z = real.freq()[i];
        for (int j = 0; j < M; j++) {
            nev_complex_matrix prod(2, 2);
            prod(0, 0) = (z - imag.freq()[j]) / (z - std::conj(imag.freq()[j]));
            prod(0, 1) = phis[j];
            prod(1, 0) = std::conj(phis[j])*
                        ((z - imag.freq()[j]) / (z - std::conj(imag.freq()[j])));
            prod(1, 1) = nev_complex{1., 0.};
            result *= prod;
        }
        nev_complex param {0., 0.}; //theta_{M+1}, choose to be constant function 0 here
        nev_complex theta = (result(0, 0) * param + result(0, 1)) / (result(1, 0) * param + result(1, 1));
        //can output "real.freq(), a.real(), a.imag(), ..., d.imag()\n" into a file for optimization convenience
        real.val()[i] = I * (One + theta) / (One - theta); //inverse Mobius transform from theta to NG
    }
    real.write();
}
```

Listing 5: nevanlinna.h

```
#include <iostream>
#include <iomanip>
#include <complex>
#include <vector>
#include <fstream>
#include <gmpxx.h>
#include <cmath>
#include <algorithm>
```

```
#include <Eigen/Dense>


//precision class is used to define typenames
//template T can be any precision type, e.g. double or mpf_class
template <class T>
class precision_ {
protected:
    using nev_real = T;
    using nev_complex = std::complex<T>;
    using nev_complex_vector = std::vector<nev_complex>;
    using nev_complex_matrix = Eigen::Matrix <nev_complex, Eigen::Dynamic, Eigen::Dynamic>;
    using nev_complex_matrix_vector = std::vector<nev_complex_matrix>;
};


//Matsubara data storage (theta values)
template <class T>
class imag_domain_data : precision_<T> {
private:
    using typename precision_<T>::nev_real;
    using typename precision_<T>::nev_complex;
    using typename precision_<T>::nev_complex_vector;
public:
    //calculate theta (G -> NG -> theta) and store Matsubara frequencies and theta
    imag_domain_data (std::string ifile, int imag_num) : N_imag_(imag_num) {
        std::ifstream ifs(ifile);
        val_.resize(N_imag_);
        freq_.resize(N_imag_);
        nev_real freq, re, im;
        nev_complex I {0., 1.};
        for (int i = 0; i < N_imag_; i++) {
            ifs >> freq >> re >> im;
            nev_complex val = nev_complex{-re, -im}; //minus signs to transform G to NG
            freq_[i] = nev_complex{0., freq};
            val_[i] = (val - I) / (val + I); //Mobius transform from NG to theta
        }
        //reverse input frequency order (decreasing then) and the corresponding thetas,
        //which tests to be the most robust interpolation order with Schur algorithm
        std::reverse(freq_.begin(),freq_.end());
        std::reverse(val_.begin(), val_.end());
    }
    //number of Matsubara points
    int N_imag() const { return N_imag_; }
    //contractive interpolant theta values at Matsubara points
    const nev_complex_vector &val() const { return val_; }
    //Matsubra frequencies
    const nev_complex_vector &freq() const { return freq_; }
private:
    int N_imag_;
    nev_complex_vector val_;
    nev_complex_vector freq_;
};


//real frequency NG storage, at omega+i*eta
template <class T>
class real_domain_data : precision_<T> {
private:
    using typename precision_<T>::nev_real;
    using typename precision_<T>::nev_complex;
    using typename precision_<T>::nev_complex_vector;
public:
    //calculate and store real frequencies (at omega+i*eta), uniform grid
    //***change N_real_, omega_min, omega_max and eta as needed***
    real_domain_data (std::string ofile) : ofs(ofile), N_real_(6000), omega_min(-10), omega_max(10), eta(0.001) {
        val_.resize(N_real_);
        freq_.resize(N_real_);
        nev_real inter = (omega_max - omega_min) / (N_real_ - 1);
        nev_real temp = omega_min;
        freq_[0] = nev_complex{omega_min, eta};
        for (int i = 1; i < N_real_; i++) {
            temp += inter;
            freq_[i] = nev_complex{temp, eta};
        }
    }
    //number of real frequencies
    int N_real() const { return N_real_; }
    //NG values at real frequencies
    nev_complex_vector &val() { return val_; }
    //real frequencies
    const nev_complex_vector &freq() const { return freq_; }
    //write real frequencies and spectral function A(omega) values to the output file
```

```
    void write () {
        for(int i = 0;i < N_real_; i++){
            ofs << std::fixed << std::setprecision(15);
            ofs << freq_[i].real() << " " << 1 / M_PI * val_[i].imag() <<std::endl;
        }
    }
private:
    std::ofstream ofs;
    int N_real_;
    nev_real omega_min;
    nev_real omega_max;
    nev_real eta;
    nev_complex_vector val_;
    nev_complex_vector freq_;
};
```