

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 discretization 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

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

Listing 2: input.txt

```
1 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 };
```

```

25
26
27 template <class T>
28 Schur<T>::Schur(std::string ifile, int imag_num, std::string ofile) : imag(ifile, imag_num), real(ofile) {
29     M = imag_num;
30     //fill the Pick matrix
31     nev_complex_matrix Pick (M, M);
32     nev_complex I {0., 1.};
33     for (int i = 0; i < M; i++) {
34         for (int j = 0; j < M; j++) {
35             nev_complex freq_i = (imag.freq()[i] - I) / (imag.freq()[i] + I);
36             nev_complex freq_j = (imag.freq()[j] - I) / (imag.freq()[j] + I);
37             nev_complex one {1., 0.};
38             nev_complex nom = one - imag.val()[i] * std::conj(imag.val()[j]);
39             nev_complex den = one - freq_i * std::conj(freq_j);
40             Pick(i, j) = nom / den;
41         }
42     }
43     //check the positive semi-definiteness of the Pick matrix using Cholesky decomposition
44     Eigen::LLT<nev_complex_matrix> lltOfPick(Pick + nev_complex_matrix::Identity(M, M) * 1e-250);
45     if(lltOfPick.info() == Eigen::NumericalIssue)
46         std::cerr << "Pick matrix is non positive semi-definite matrix in Schur method." << std::endl;
47     else std::cerr << "Pick matrix is positive semi-definite." << std::endl;
48 }
49
50
51 template <class T>
52 void Schur<T>::core() {
53     phis.resize(M);
54     abcds.resize(M);
55     phis[0] = imag.val()[0];
56     for (int k = 0; k < M; k++) abcds[k] = nev_complex_matrix::Identity(2, 2);
57     for (int j = 0; j < M - 1; j++) {
58         for (int k = j; k < M; k++) {
59             nev_complex_matrix prod(2, 2);
60             prod(0, 0) = (imag.freq()[k] - imag.freq()[j]) / (imag.freq()[k] - std::conj(imag.freq()[j]));
61             prod(0, 1) = phis[j];
62             prod(1, 0) = std::conj(phis[j]) *
63                 ((imag.freq()[k] - imag.freq()[j]) / (imag.freq()[k] - std::conj(imag.freq()[j])));
64             prod(1, 1) = nev_complex{1., 0.};
65             abcds[k] *= prod;
66         }
67         phis[j + 1] = (- abcds[j + 1](1, 1) * imag.val()[j + 1] + abcds[j + 1](0, 1)) /
68             (abcds[j + 1](1, 0) * imag.val()[j + 1] - abcds[j + 1](0, 0));
69     }
70 }
71
72
73 template <class T>
74 void Schur<T>::evaluation () {
75     core();
76     nev_complex I {0., 1.};
77     nev_complex One {1., 0.};
78     for (int i = 0; i < real.N_real(); i++) {
79         nev_complex_matrix result = nev_complex_matrix::Identity(2, 2);
80         nev_complex z = real.freq()[i];
81         for (int j = 0; j < M; j++) {
82             nev_complex_matrix prod(2, 2);
83             prod(0, 0) = (z - imag.freq()[j]) / (z - std::conj(imag.freq()[j]));
84             prod(0, 1) = phis[j];
85             prod(1, 0) = std::conj(phis[j]) *
86                 ((z - imag.freq()[j]) / (z - std::conj(imag.freq()[j])));
87             prod(1, 1) = nev_complex{1., 0.};
88             result *= prod;
89         }
90         nev_complex param {0., 0.}; //theta_{M+1}, choose to be constant function 0 here
91         nev_complex theta = (result(0, 0) * param + result(0, 1)) / (result(1, 0) * param + result(1, 1));
92         //can output "real.freq(), a.real(), a.imag(), ..., d.imag()\n" into a file for optimization convenience
93         real.val()[i] = I * (One + theta) / (One - theta); //inverse Mobius transform from theta to NG
94     }
95     real.write();
96 }

```

Listing 5: nevanlinna.h

```

1 #include <iostream>
2 #include <iomanip>
3 #include <complex>
4 #include <vector>
5 #include <fstream>
6 #include <gmpxx.h>
7 #include <cmath>
8 #include <algorithm>

```

```

9 #include <Eigen/Dense>
10
11
12 //precision class is used to define typenames
13 //template T can be any precision type, e.g. double or mpf_class
14 template <class T>
15 class precision_ {
16 protected:
17     using nev_real = T;
18     using nev_complex = std::complex<T>;
19     using nev_complex_vector = std::vector<nev_complex>;
20     using nev_complex_matrix = Eigen::Matrix<nev_complex, Eigen::Dynamic, Eigen::Dynamic>;
21     using nev_complex_matrix_vector = std::vector<nev_complex_matrix>;
22 };
23
24
25 //Matsubara data storage (theta values)
26 template <class T>
27 class imag_domain_data : precision_<T> {
28 private:
29     using typename precision_<T>::nev_real;
30     using typename precision_<T>::nev_complex;
31     using typename precision_<T>::nev_complex_vector;
32 public:
33     //calculate theta (G -> NG -> theta) and store Matsubara frequencies and theta
34     imag_domain_data (std::string ifile, int imag_num) : N_imag_(imag_num) {
35         std::ifstream ifs(ifile);
36         val_.resize(N_imag_);
37         freq_.resize(N_imag_);
38         nev_real freq, re, im;
39         nev_complex I {0., 1.};
40         for (int i = 0; i < N_imag_; i++) {
41             ifs >> freq >> re >> im;
42             nev_complex val = nev_complex{-re, -im}; //minus signs to transform G to NG
43             freq_[i] = nev_complex{0., freq};
44             val_[i] = (val - I) / (val + I); //Mobius transform from NG to theta
45         }
46         //reverse input frequency order (decreasing then) and the corresponding thetas,
47         //which tests to be the most robust interpolation order with Schur algorithm
48         std::reverse(freq_.begin(), freq_.end());
49         std::reverse(val_.begin(), val_.end());
50     }
51     //number of Matsubara points
52     int N_imag() const { return N_imag_; }
53     //contractive interpolant theta values at Matsubara points
54     const nev_complex_vector &val() const { return val_; }
55     //Matsubara frequencies
56     const nev_complex_vector &freq() const { return freq_; }
57 private:
58     int N_imag_;
59     nev_complex_vector val_;
60     nev_complex_vector freq_;
61 };
62
63
64 //real frequency NG storage, at omega+i*eta
65 template <class T>
66 class real_domain_data : precision_<T> {
67 private:
68     using typename precision_<T>::nev_real;
69     using typename precision_<T>::nev_complex;
70     using typename precision_<T>::nev_complex_vector;
71 public:
72     //calculate and store real frequencies (at omega+i*eta), uniform grid
73     //***change N_real_, omega_min, omega_max and eta as needed***
74     real_domain_data (std::string ofile) : ofs(ofile), N_real_(6000), omega_min(-10), omega_max(10), eta(0.001) {
75         val_.resize(N_real_);
76         freq_.resize(N_real_);
77         nev_real inter = (omega_max - omega_min) / (N_real_ - 1);
78         nev_real temp = omega_min;
79         freq_[0] = nev_complex{omega_min, eta};
80         for (int i = 1; i < N_real_; i++) {
81             temp += inter;
82             freq_[i] = nev_complex{temp, eta};
83         }
84     }
85     //number of real frequencies
86     int N_real() const { return N_real_; }
87     //NG values at real frequencies
88     nev_complex_vector &val() { return val_; }
89     //real frequencies
90     const nev_complex_vector &freq() const { return freq_; }
91     //write real frequencies and spectral function A(omega) values to the output file

```

```
92 void write () {  
93     for(int i = 0; i < N_real_; i++){  
94         ofs << std::fixed << std::setprecision(15);  
95         ofs << freq_[i].real() << " " << 1 / M_PI * val_[i].imag() <<std::endl;  
96     }  
97 }  
98 private:  
99     std::ofstream ofs;  
100     int N_real_;  
101     nev_real omega_min;  
102     nev_real omega_max;  
103     nev_real eta;  
104     nev_complex_vector val_;  
105     nev_complex_vector freq_;  
106 };
```