LinRealRecursiveSeq Reference

1 Introduction

`LinRealRecursiveSeq` is a simple C++ class allowing for relatively fast calculation of a recursive sequence elements with constant coefficients. Sequence elements are given as a double-precision floating-point numbers and their accuracy is limited to machine precision. It uses a method based on eigendecomposition of a matrix representation of recurrence relation, which allows for evaluation $\mathcal{O}(1)$ complexity with respect to the sequence element index.

2 Mathematical background

All recursive sequences S with constant coefficients a_i can be defined as:

$$S(k) = \begin{cases} S_k & ; \ k < N \\ \sum_{i=0}^{N-1} a_i \cdot S(k-i-1) & ; \ k \geq N \end{cases} \tag{1}$$

with $S_0, \ldots, S_{N-1}$ being predefined first N elements of a given sequence. This recurrence relation can be expressed in a matrix form:

$$\begin{pmatrix} S(k) \\ S(k+1) \\ \vdots \\ S(k+(N-3)) \\ S(k+(N-2)) \\ S(k+(N-1)) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ a_0 & a_1 & a_2 & \cdots & a_{N-2} & a_{N-1} \end{pmatrix} \cdot \begin{pmatrix} S(k-1) \\ S(k) \\ \vdots \\ S(k+(N-4)) \\ S(k+(N-3)) \\ S(k+(N-2)) \end{pmatrix} \tag{2}$$

$$= \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ a_0 & a_1 & a_2 & \cdots & a_{N-2} & a_{N-1} \end{pmatrix}^k}_{M^k} \cdot \begin{pmatrix} S(0) \\ S(1) \\ \vdots \\ S(N-3) \\ S(N-2) \\ S(N-1) \end{pmatrix} \tag{3}$$

M^k can be simply calculated with eigendecomposition $M = C \cdot \mathrm{diag}(m_1, \ldots, m_N) \cdot C^{-1}$ with $\mathrm{diag}(\ldots)$ being a diagonal matrix composed of given arguments, $m_1, \ldots, m_N$ are (in general complex) being eigenvalues of M anc C being a matrix composed of M's eigenvectors. In this form:

$$M^k = \left(C \cdot \mathrm{diag}(m_1, \ldots, m_N) \cdot C^{-1}\right)^k \tag{4}$$

$$= \left(C \cdot \mathrm{diag}(m_1, \ldots, m_N) \cdot C^{-1}\right) \times \ldots \times \left(C \cdot \mathrm{diag}(m_1, \ldots, m_N) \cdot C^{-1}\right) \tag{5}$$

$$= C \cdot \left(\mathrm{diag}(m_1, \ldots, m_N)\right)^k \cdot C^{-1} = C \cdot \mathrm{diag}(m_1^k, \ldots, m_N^k) \cdot C^{-1} \tag{6}$$

which means that calculating the kth sequence element requires the knowledge of first N sequence elements and M's eigenvalues (to the power k) and eigenvectors. This method, although limited by the machine precision (the eigenvalues are represented by floating-point variables), allows for much faster computation than straightforward recursive or sequential algorithm, with constant memory usage (assuming the eigensystem is precomputed). `LinRealRecursiveSeq` class is based on that premise.

3 Prerequisites and compilation

`LinRealRecursiveSeq` class requires:

- GCC 4.7 (or later) - provides support for constructor delegation (part of C++11 standard)
- GSL 1.9 (or later) - provides nonsymmetric matrix diagonalization method, GSL CBLAS Library and GSL BLAS Interface
- (optional) other BLAS library (OpenBLAS, ATLAS, ...) - alternative for GSL CBLAS Library

During all compilations with `LinRealRecursiveSeq` class `-std=c++11` flag must be used (optionally: `-DHAVE_INLINE` for optimizing GSL methods usage). During every linking GSL and GSL CBLAS libraries have to be linked (`-lgsl -lgslcblas`). Optionally, instead of GSL CBLAS user can link other BLAS library.

4 Class reference

```
class LinRealRecursiveSeq {
public:
      LinRealRecursiveSeq();
      LinRealRecursiveSeq (const LinRealRecursiveSeq& realRecurSeq);
      LinRealRecursiveSeq& operator= (const LinRealRecursiveSeq& realRecurSeq);
      LinRealRecursiveSeq (const std::vector<double>& firstElements,
                           const std::vector<double>& recurrenceRelation);
      LinRealRecursiveSeq (const double *firstElements,
                           const double *recurrenceRelation,
                           size_t N);
     ~LinRealRecursiveSeq ();
      double Element(unsigned int k);

private:
      gsl_vector_complex *_first_elems_;
      gsl_vector_complex *_eigenvals_;
      gsl_matrix_complex *_transM_;
      gsl_matrix_complex *_invTransM_;
      gsl_matrix_complex *_diag_;
      gsl_vector_complex *_elems1_;
      gsl_vector_complex *_elems2_;
      size_t _N_;
};
```

4.1 Constructors and destructor

`LinRealRecursiveSeq::LinRealRecursiveSeq ();`

Default constructor.
Creates a constant sequence of zeroes.
Exceptions: None.

`LinRealRecursiveSeq::LinRealRecursiveSeq (const LinRealRecursiveSeq& realRecurSeq);`

Copy constructor.
Creates a copy of given `LinRealRecursiveSeq` object (`realRecurSeq`).
Exceptions: None.

```
LinRealRecursiveSeq::LinRealRecursiveSeq (const std::vector<double>& firstElements,
                                          const std::vector<double>& recurrenceRelation);
```

Constructor.
Takes first N sequence elements $S_0, \ldots, S_{N-1}$ (`firstElements`) and recurrence relations coefficients $a_0, \ldots, a_{N-1}$ (`recurrenceRelation`), both stored in ascending order with respect to their indices.
If $N = 1$, only `_first_elems_` and `_eigenvals_` are initialized as 1×1 complex vectors, other GSL matrix/vectors objects becomes NULL pointers. Otherwise, constructor performs eigendecomposition and allocates all GSL objects.
Exceptions: If one of `firstElements` or `recurrenceRelation` is empty or `firstElements` and `recurrenceRelation` aren't the same size, throws `std::length_error`.

```
LinRealRecursiveSeq::LinRealRecursiveSeq (const double *firstElements,
                                          const double *recurrenceRelation,
                                          size_t N);
```

Constructor.
Takes first N sequence elements $S_0, \ldots, S_{N-1}$ (`firstElements`) and recurrence relations coefficients $a_0, \ldots, a_{N-1}$ (`recurrenceRelation`), both stored in ascending order with respect to their indices. Size of both arrays is passes in `N`. If one of them has size larger than `N`, only first N elements will be used. User must ensure that both of `firstElements` and `recurrenceRelation` are the same size.
If $N = 1$, only `_first_elems_` and `_eigenvals_` are initialized as 1×1 complex vectors, other GSL matrix/vectors objects becomes NULL pointers. Otherwise, constructor performs eigendecomposition and allocates all GSL objects.
Exceptions: If `N` is 0, throws `std::length_error`.

`LinRealRecursiveSeq::~LinRealRecursiveSeq ();`

Destructor.
Exceptions: None.

4.2 Operators

```
LinRealRecursiveSeq&
LinRealRecursiveSeq::operator= (const LinRealRecursiveSeq& realRecurSeq);
```

Copy operator.
Reallocates and copies (if necessary) all private members of given `LinRealRecursiveSeq` object (`realRecurSeq`).
Exceptions: None.

4.3 Other public members

```
double
LinRealRecursiveSeq::Element (unsigned int k);
```

Returns kth sequence element.
All matrix and vector operations are perfomed by using GSL BLAS Interface and (allocated during construction) private members. Returns real part of calculated vector's zeroth element.
Exceptions: None.

4.4 Private members

```
gsl_matrix_complex*
LinRealRecursiveSeq::_diag_;
```

If $N = 1$, points to NULL. Otherwise, points to a diagonal buffor matrix used in `Element()` function for holding the eigenvalues during calculations.

```
gsl_vector_complex*
LinRealRecursiveSeq::_eigenvals_;
```

Points to the eigenvalues of recurrence sequence matrix form (complex in general).

```
gsl_vector_complex*
LinRealRecursiveSeq::_elems1_;
```

If $N = 1$, points to NULL. Otherwise, points to a complex buffor vector used in `Element()` function for holding the intermediate values of calculated vector.

```
gsl_vector_complex*
LinRealRecursiveSeq::_elems2_;
```

If $N = 1$, points to NULL. Otherwise, points to a complex buffor vector used in `Element()` function for holding the intermediate values of calculated vector.

```
gsl_vector_complex*
LinRealRecursiveSeq::_first_elems_;
```

Points to first N elements of sequence.

```
gsl_matrix_complex*
LinRealRecursiveSeq::_invTransM_;
```

If $N = 1$, points to NULL. Otherwise, points to the inverse of a matrix composed of eigenvectors of recurrence sequence matrix form.

```
size_t
LinRealRecursiveSeq::_N_;
```

Stores N.

```
gsl_matrix_complex*
LinRealRecursiveSeq::_transM_;
```

If $N = 1$, points to NULL. Otherwise, points to the matrix composed of eigenvectors of recurrence sequence matrix form.

5 Examples

Fibonacci ($F(k)$ and Perrin ($P(k)$) numbers can serve as examples of `LinRealRecursiveSeq` usage. They're defined as:

$$F(k) = \begin{cases} 0 & ; k = 0 \\ 1 & ; k = 1 \\ F(k-2) + F(k-1) & ; k > 1 \end{cases} \qquad P(k) = \begin{cases} 3 & ; k = 0 \\ 0 & ; k = 1 \\ 2 & ; k = 2 \\ F(k-3) + F(k-2) & ; k > 2 \end{cases} \tag{7}$$

Below code implements a simple program calculating first 20 Fibonacci and Perrin numbers and prints them to the standard output:

```
#include <LinRealRecursiveSeq.h> // LinRealRecursiveSeq class
#include <vector>                // std::vector
#include <iostream>              // std::cout

int main(){
    /* Input vectors */
    std::vector<double> fibonacciFirstElems, fibonacciRecurRel;
    std::vector<double> perrinFirstElems,    perrinRecurRel;

    /* Saving first two Fibonacci numbers and recurrence relation
     * coefficients */
    fibonacciFirstElems.push_back(0.0);
    fibonacciFirstElems.push_back(1.0);

    fibonacciRecurRel.push_back(1.0);
    fibonacciRecurRel.push_back(1.0);

    /* Saving first three Perrin numbers and recurrence relation
     * coefficients */
    perrinFirstElems.push_back(3.0);
    perrinFirstElems.push_back(0.0);
    perrinFirstElems.push_back(2.0);

    perrinRecurRel.push_back(1.0);
    perrinRecurRel.push_back(1.0);
    perrinRecurRel.push_back(0.0);

    /* Creating Fibonacci and Perrin sequences */
    LinRealRecursiveSeq fibonacciSeq(fibonacciFirstElems,
                                     fibonacciRecurRel);

    LinRealRecursiveSeq perrinSeq;
    perrinSeq = LinRealRecursiveSeq(perrinFirstElems,
                                    perrinRecurRel);

    /* Print first 20 Fibonacci and Perrin numbers */
    for (int k = 0; k < 20; ++k){
        std::cout << k                      << " "
                  << fibonacciSeq.Element(k) << " "
                  << perrinSeq.Element(k)    << std::endl;
    }

    return 0;
}
```

Similiar programs were used to compare calculated values with the exact values. Both absolute and relative errors were obtained. Comparisons were performed in two ways: with elements as floating-point variables and rounded to the nearest `long long int` value (`roundl()` function from the C Standard Library was used). Results are shown in Figure 1.

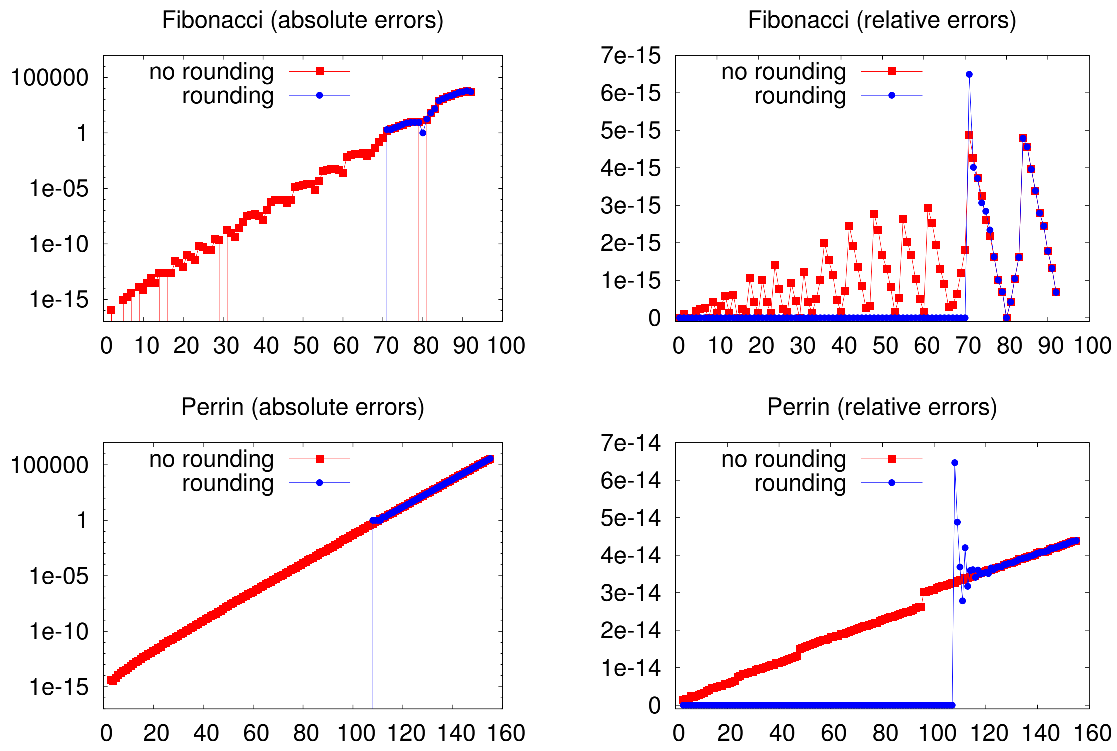

Figure 1: Absolute and relative errors between exact and calculated Fibonacci and Perrin sequences elements (both with and without rounding).