The Bloss package

IKEGAMI, Tsutomu
⟨t-ikegami@aist.go.jp⟩
AIST

Version 1.00
Feb. 2012

Abstract

This package provides an MPI parallel framework for an easy implementation of the block Sakurai-Sugiura eigensolver. The block Sakurai-Sugiura method calculates interior eigenvalues of generalized eigenvalue problems, $(\lambda B - A)q = 0$, for those λ in the specific region G. Typically, G is taken as the inside of a circular path on the complex plane, which may lie in the middle of the eigen-spectrum. In the Sakurai-Sugiura method, you have to solve a set of linear equations, which are independent of each other, and can be solved concurrently. Bloss is designed to help the concurrent execution of the linear solvers, along with pre-/post-processes required for the block Sakurai-Sugiura method.

Contents

1 Introduction

The block Sakurai-Sugiura (SS) method is a parallel solver for interior eigenproblems. It solves generalized eigenproblems, $(\lambda B - A)q = 0$, where the matrix $\lambda B - A$ is regular. Its main target is large-scale sparse matrix systems, where conventional methods work less efficiently on massively parallel computers. As an interior eigensolver, not all of the eigenvalues are calculated, but those that reside in the interior region G are extracted. In the block SS method, the region G is defined mathematically as the inside of a closed Jordan curve Γ, along which a contour integral is evaluated. Typically, G is set up such that $10 \sim 20$ eigenpairs are located inside.

Conventionally, iterative methods are employed to solve large-scale eigenproblems, where the interior eigen-subspace is refined sequentially in a build-up manner. In contrast, the block SS method generates the eigen-subspace in one step. To generate the subspace, a set of linear equations has to be solved along the contour path, which forms the most time-consuming step in the block SS method. Fortunately, those equations are independent of each other, and can be solved simultaneously. The Bloss package offers a framework to manage concurrent solution of these linear equations, as well as the pre-/post-processes necessary for the block SS method. Each linear equation may be solved further in parallel, where a large amount of computational resources can be organized in a hierarchical manner. In the next section, a minimal usage of the package is illustrated, followed by the theories behind it. References to the Bloss API are described next, and some advanced topics are discussed last.

2 Tutorial

The Bloss package offers a framework to write block SS applications based on the MPI programming model. In this section, an outline of a simple Bloss application is given. An overview of the calculation is summarized in Fig. 1.

Definition of the problem The eigenproblem to be solved is $(\lambda B - A)q = 0$, where $A, B \in \mathbb{C}^{n\times n}$ are complex matrices. What is to be calculated are those $\lambda \in G$, where G is inside of a circle Γ placed on the complex plane. The center and the radius of Γ are γ and ρ, respectively. It is assumed that you have *a priori* knowledge of the approximate number of eigenvalues in G.

Parameters for the block SS method The contour integral along Γ is approximated by the M-point trapezoidal rule. A moderate M (~ 32) may suffice, because the accuracy of

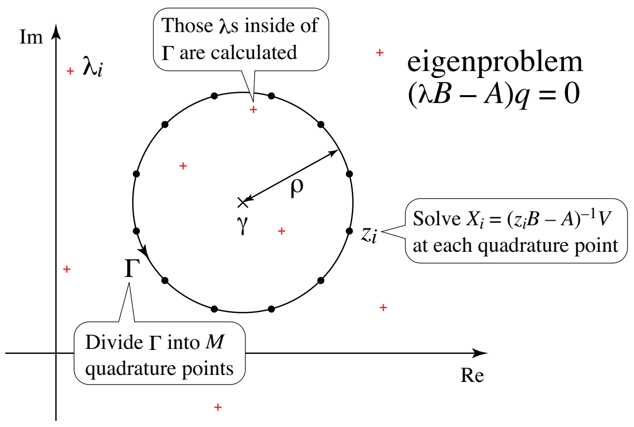
Im
λ_i
Those λs inside of Γ are calculated
eigenproblem $(\lambda B - A)q = 0$
ρ
γ
z_i
Solve $X_i = (z_i B - A)^{-1} V$ at each quadrature point
Γ
Divide Γ into M quadrature points
Re

Figure 1: Setup of the tutorial calculation.

the integral is not directly related to the accuracy of the results. A set of basis vectors spanning the interior eigen-subspace is generated from L arbitrary initial vectors. During the block SS procedure, the initial vectors are projected onto the subspace, and further multiplexed by K, where K is the maximum order of modulation (see Sec. 3.1). The dimension of the subspace is thus LK, at most, so that L and K have to be chosen large enough to span the interior eigen-subspace completely.

Routines you must prepare The Bloss package requires you to provide two kinds of operations:

1. Solve linear equations $Y = (\omega B - A)^{-1}V$, and
2. Operate matrix $Y = AV$ and $Y = BV$,

where $V \in \mathbb{C}^{n \times l}$ is a multi-column matrix with $l \simeq L$, and $\omega \in \mathbb{C}$ is a constant. Besides parameters ω, V, and l, a small MPI communicator is also provided by the package, which may be used to process the operations in parallel. The matrices A and B are accessed only via these requests; the package does not handle the matrices directly.

Design of the parallel execution Suppose that we have N processors in the base MPI communicator. At the inversion phase (1), a total of M linear equations have to be solved. In this tutorial, we employ a single-threaded linear solver, so that M equations are assigned evenly among N processors. As a result, the maximum number of processors is $N \leq M$. A similar setup is used at the mat-vec phase (2). That is, a single-threaded matrix-vector multiplier is employed, and the total number of vectors ($\simeq LK$) is divided evenly among N processors.

Sample code An example of the Bloss code needed to solve the tutorial problem defined above is listed in Fig. 2. Different from one-step eigensolvers, such as those in LAPACK, the problem is solved through interaction with users. Before starting the calculation, the MPI execution environment has to be established by `MPI_Init()`. A base communicator `Comm` (typically `MPI_COMM_WORLD`) that has N processors should be prepared and passed to the package. The complete code, including pre- and post-processes, is available at `examples/C/tutorial.c`.

INPUT

`int n`	dimension of the system
`complex_16 A[n*n], B[n*n]`	input matrices
`complex_16 gamma`	center of a contour path Γ
`double rho`	radius of a contour path Γ
`int M`	number of quadrature points (~ 32)
`int L`	number of initial vectors (~ 8)
`int K`	maximum order of modulation (~ 8)
`MPI_Comm Comm`	base MPI communicator

All the INPUT data should be `MPI_Bcast`-ed over all the ranks of the communicator `Comm`. The type `complex_16` is defined in `bloss.h`, and is equivalent to `double complex`.

OUTPUT

`int *neig`	number of eigenpairs obtained ($\leq$ `L*K`)
`complex_16 (*lambda)[neig]`	eigenvalues
`complex_16 (*q)[n*neig]`	eigenvectors

```
#include "bloss.h"

void tutorial( int n, complex_16 *A, complex_16 *B,
               complex_16 gamma, double rho, int M, int L, int K,
               int *neig, complex_16 **lambda, complex_16 **q,
               MPI_Comm Comm )
{
  Bloss *om;
  int type = 0;
  double ellipse = 1.0, tolerance = 1.0e-12;

  om = bloss_setup_ellipse(type, Comm, n, L, K, M, gamma, rho,
                           ellipse, tolerance);

  while(1) {
    int task, l;
    complex_16 *V, *Y, omega;
    MPI_Comm worker;

    bloss_do(om, &task, &l, &V, &Y, &omega, &worker);
    switch(task) {
    case BLOSS_TASK_INVERT:
      // Solve (omega B - A).Y = V for Y.  V, Y ∈ ℂ^{n×l}.
      linear_solve( n, A, B, omega, V, l, Y );
      break;

    case BLOSS_TASK_MATMUL_A:
      // Calculate Y = A.V.  V,Y ∈ ℂ^{n×l}.
      mat_vec( n, A, V, l, Y );
      break;

    case BLOSS_TASK_MATMUL_B:
      // Calculate Y = B.V.  V,Y ∈ ℂ^{n×l}.
      mat_vec( n, B, V, l, Y );
      break;

    case BLOSS_TASK_DONE:
      if ( 0 == bloss_get_rank(om) ) {
        *neig   = bloss_get_neig(om);
        *lambda = bloss_get_workspace(om, BLOSS_WS_EIGVALS);
        *q      = bloss_get_workspace(om, BLOSS_WS_EIGVECS);

        bloss_detach_ptr(om, *lambda);
        bloss_detach_ptr(om, *q);
      }
      bloss_free(om);
      return;
    }
  }
}
```

Figure 2: Code used to solve the tutorial example.

OUTPUT data are available only on rank 0 of `Comm`. Note that, in addition to the eigenvalues inside of Γ, several eigenvalues located on the periphery of Γ appear in `lambda[]`. Those peripheral eigenpairs are not so accurate as the interior ones. It also must be noted that some ghost eigenpairs, which are totally inaccurate, may appear inside of Γ. These contaminant eigenpairs can be discriminated either by residual calculations or by reliability indices, both of which will be described later.

During the Bloss procedure, the user interaction context is kept in `Bloss *om`. The context is established by `bloss_setup_ellipse()`, where the contour path, the quadrature, and the initial vectors are prepared. The parallel environments are also set up here, by dividing the base communicator `Comm`. A subsequent call to `bloss_do()` starts the interactive session. Instructions are given in `int task`, according to which users should take an appropriate action and call `bloss_do()` again. Basically, users should calculate either `Y = (omega*B - A)`$^{-1}$`.V`, `Y = A.V`, or `Y = B.V`. Besides these requests, `bloss_do()` is returned at several *hook points*, which reports the progress of the calculation and gives users a chance to access intermediate results. All these hooks are ignored in the present example.

The Bloss procedure finishes when `task == BLOSS_TASK_DONE`. At this point, the eigenpairs are stored in `*om`, and can be accessed by `bloss_get_workspace()`. These storage areas allocated in `*om` are discarded when `bloss_free()` is called to destroy the Bloss context. To protect them from the destruction, `bloss_detach_ptr()` can be called to allow the storage for eigenvalues and eigenvectors to survive beyond `bloss_free()`.

A small communicator, `worker`, which is generated by dividing `Comm`, is also provided by `bloss_do()`. At the inversion phase, only a single processor is assigned to `worker` if the number of processors N is equal to or less than M. When more processors are available, multiple processors are assigned to `worker`, which can be utilized for parallel linear solvers, if available. Note that, because a total of M tasks are assigned statically to `worker`, N should be taken as either a divisor or a multiple of M for better load balance. Just as in the inversion phase, it is also possible to assign multiple processors to `worker` at the mat-vec phase.

More robust examples with error checks can be found under the directory `examples`.

3 Theories behind Bloss

In this section, a theoretical background of the block Sakurai-Sugiura method is outlined. A basic knowledge of this background will be of help to determine parameters to set up the Bloss context.

3.1 Block Sakurai-Sugiura method

Let $A, B \in \mathbb{C}^{n\times n}$, Γ be a positively oriented closed Jordan curve, and G be the inside of Γ. The aim of the block SS method is to determine eigenvalues $\lambda \in G$ for the generalized eigenproblem $(\lambda B - A)q = 0$. In the block SS method, a moment operator M_k,

$$M_k = \frac{1}{2\pi i}\oint_\Gamma z^k (zB - A)^{-1} dz, \tag{1}$$

plays an important role. The moment operator works as a projection operator (or, from the filter-diagonalization point of view, as a filter operator).

To illustrate the properties of M_k, let's take an Hermitian system, $A = H$ and $B = \mathbb{I}$, as an example. Suppose that an arbitrary vector v is expanded in terms of eigenvectors as

$$v = \sum_i c_i q_i, \tag{2}$$

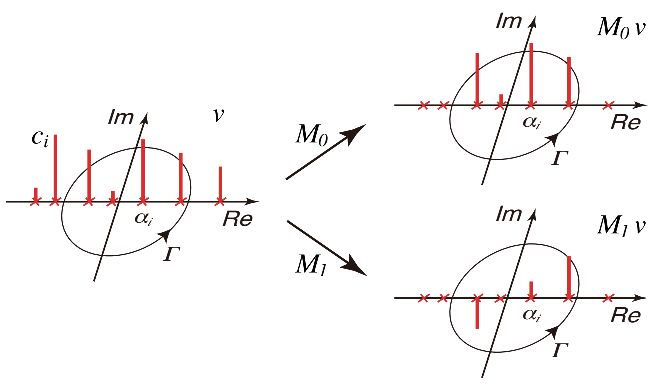

Figure 3: Modulation of the eigen-spectrum by the moment operator. The eigen-spectrum of the initial vector v is shown in a histogram, along with those of $M_0 v$ and $M_1 v$.

where $(\lambda_i \mathbb{I} - H)q_i = 0$. When operated by M_k, the eigenspectrum of v is modulated as

$$M_k v = \sum_i f_k(\lambda_i) c_i q_i, \tag{3}$$

where

$$f_k(x) = \frac{1}{2\pi i} \oint_\Gamma \frac{z^k}{z - x} dz = \begin{cases} x^k, & x \in G, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

By the operation of M_k, the spectrum of v is not only projected on G, but also modulated according to λ^k. The modulation effect is schematically shown in Fig. 3. Thanks to the modulation, several independent vectors can be generated from a single vector v. While the filter function Eq. (4) is derived intuitively for an Hermitian system, the same filter concept can be applicable for generalized eigenproblems. A thorough derivation is given in [1].

Based on the projector-modulator nature of M_k, we can construct the interior eigen-subspace from a set of arbitrary initial vectors. Let $V \in \mathbb{C}^{n \times L}$ be a set of L initial vectors, and n_Γ be the number of eigenvalues in G. The interior eigen-subspace can be spanned by a set of basis vectors $S_R \in \mathbb{C}^{n \times LK}$,

$$S_R = \{M_0 V, M_1 V, \ldots, M_{K-1} V\}, \tag{5}$$

where K and L are taken large enough that $LK \geq n_\Gamma$. Thanks to the spectral modulation, the L-dimensional subspace V is multiplexed by a factor of K, so that the eigen-subspace can be spanned completely even if $L < n_\Gamma$. To obtain eigenpairs, we prepare arbitrary left basis vectors $S_L \in \mathbb{C}^{n \times LK}$, and apply the Petrov-Galerkin procedure. Let $\mathbb{A} = S_L^H A S_R$ and $\mathbb{B} = S_L^H B S_R$, where the superscript H indicates the complex conjugate transpose. The original interior eigenproblem is reduced to a smaller eigenproblem,

$$(\lambda \mathbb{B} - \mathbb{A})\mathsf{q} = 0, \tag{6}$$

where the eigenvalue λ is identical to the original, and the corresponding eigenvector is given by $q = S_R \mathsf{q}$.

Based on the above theory, the Bloss package is equipped with two approaches to construct $\mathbb{A}$ and $\mathbb{B}$: a Rayleigh-Ritz (RR) method and a moment-based method. Both are described in the following two sections.

3.2 Rayleigh-Ritz method

Because $LK \geq n_\Gamma$, the basis set S_R is over-complete. To obtain a minimal basis set, a singular value decomposition is performed on S_R,

$$S_R = U \mathsf{s} \mathbb{W}^H, \tag{7}$$

where $U \in \mathbb{C}^{n\times LK}$ and $\mathbb{W} \in \mathbb{C}^{LK\times LK}$ are sets of ortho-normal vectors, and $\mathsf{s} \in \mathbb{R}^{LK\times LK}$ is a diagonal matrix of singular values. Mathematically, only n_Γ singular values are non-zero. A new basis set $\bar{U} \in \mathbb{C}^{n\times n_\Gamma}$ is then constructed by collecting non-zero singular components from U. According to the Rayleigh-Ritz procedure, the projections are generated as $\mathbb{A} = \bar{U}^H A\bar{U}$ and $\mathbb{B} = \bar{U}^H B\bar{U}$. The eigenvalues are obtained by solving $(\lambda\mathbb{B} - \mathbb{A})\mathsf{q} = 0$, and the corresponding eigenvectors are given by $q = \bar{U}\mathsf{q}$.

3.3 Moment-based method

The moment operator satisfies the following addition theorem [2],

$$\begin{aligned} M_i A M_j &= M_{i+j+1}, &(8)\\ M_i B M_j &= M_{i+j}. &(9)\end{aligned}$$

Let the left basis vectors be

$$S_L = \{M_0^H\tilde{V}, M_1^H\tilde{V}, \ldots, M_{K-1}^H\tilde{V}\}, \tag{10}$$

where $\tilde{V} \in \mathbb{C}^{n\times L}$ is an arbitrary set of vectors (typically $\tilde{V} = V$). Then, the projections of A and B become

$$\mathbb{A} = \begin{pmatrix} \mu_1 & \mu_2 & \cdots & \mu_K \\ \mu_2 & \mu_3 & \cdots & \mu_{K+1} \\ \vdots & \vdots & & \vdots \\ \mu_K & \mu_{K+1} & \cdots & \mu_{2K-1} \end{pmatrix}, \quad \mathbb{B} = \begin{pmatrix} \mu_0 & \mu_1 & \cdots & \mu_{K-1} \\ \mu_1 & \mu_2 & \cdots & \mu_K \\ \vdots & \vdots & & \vdots \\ \mu_{K-1} & \mu_K & \cdots & \mu_{2K-2} \end{pmatrix}, \tag{11}$$

respectively, where $\mu_k = \tilde{V}^H M_k V \in \mathbb{C}^{L\times L}$. That is, both $\mathbb{A}$ and $\mathbb{B}$ can be constructed from the series $\{\mu_k, k = 0, \ldots, 2K-1\}$.

The projected matrices are implicitly based on the over-complete basis set. To remove the null space, $\mathbb{B}$ is singular-value decomposed as

$$\mathbb{B} = \mathbb{U}\mathsf{s}\mathbb{W}^H, \tag{12}$$

where $\mathbb{U}, \mathbb{W} \in \mathbb{C}^{LK\times n_\Gamma}$ are sets of ortho-normal vectors, and $\mathsf{s} \in \mathbb{R}^{n_\Gamma\times n_\Gamma}$ is a diagonal matrix of non-zero singular values. An effective projection of A is then obtained as

$$\tilde{\mathbb{A}} = \mathsf{s}^{-1/2}\mathbb{U}^H\mathbb{A}\mathbb{W}\mathsf{s}^{-1/2} \quad \in \mathbb{C}^{n_\Gamma\times n_\Gamma}. \tag{13}$$

The original interior eigenproblem is now converted to $(\lambda\mathbb{I} - \tilde{\mathbb{A}})\mathsf{q} = 0$, where the eigenvalue λ is identical to the original, and the corresponding eigenvector is given by $q = S_R\mathbb{W}\mathsf{s}^{-1/2}\mathsf{q}$.

3.4 Numerical implementation

For numerical reasons, the modulation term z^k in Eq. (1) is replaced by the shifted-and-scaled one, $((z-\gamma)/\rho)^k$. The shift-and-scale parameters are chosen such that the modulation term stays around unity on G. That is, a circle centered at γ with radius ρ roughly overlaps with the contour path Γ. It must be noted that, in the moment-based method, the obtained eigenvalues are also shifted-and-scaled. This is because the addition theorem Eqs. (8) and (9) now work on the eigenproblem of $(\frac{\lambda-\gamma}{\rho}B - A)q = 0$. In the Rayleigh-Ritz method, eigenvalues are not affected by the configuration of the modulation term.

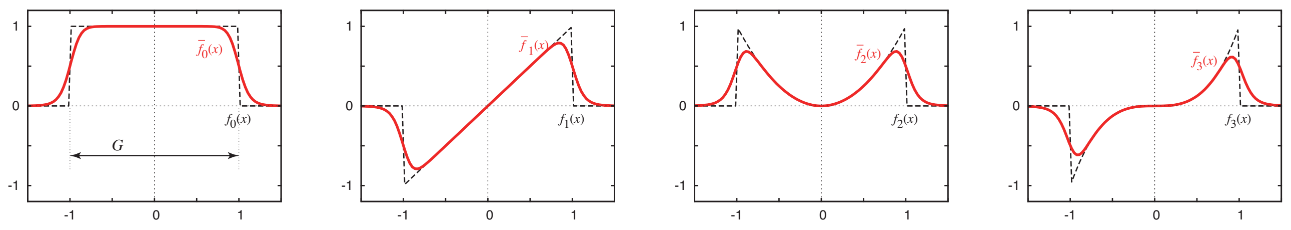

Figure 4: Practical filter function $\bar{f}_k(x)$ of the numerical block SS method is plotted along the real axis. The corresponding ideal shape $f_k(x)$ is also plotted in a broken line. The number of quadrature points is $M = 16$, and $k = 0 \sim 3$ are shown from left to right.

The contour integral in Eq. (1) is then evaluated numerically by using an appropriate quadrature. The approximated moment operator becomes

$$\bar{M}_k = \sum_{j=1}^{M} w_j \left(\frac{z_j - \gamma}{\rho}\right)^k (z_j B - A)^{-1}, \tag{14}$$

where z_j and w_j are the quadrature points and weights, respectively. Because of the quadrature approximation, $\bar{M}_k$ loses its sharpness as a filter operator. The corresponding filter function becomes

$$\bar{f}_k(x) = \sum_{j=1}^{M} w_j \left(\frac{z_j - \gamma}{\rho}\right)^k \frac{1}{z_j - x}, \tag{15}$$

where the clear boundary of Eq. (4) is obscured.

As a demonstration, let Γ be a circle centered at γ with radius ρ. Employing the M-point trapezoidal rule, we have $z_j = \gamma + \rho \exp(\frac{2\pi i}{M}(j - \frac{1}{2}))$ and $w_j = (z_j - \gamma)/M$. This set of $\{z_j, w_j\}$ gives $\bar{f}_k(x) = \tilde{x}^k/(1 + \tilde{x}^M)$, where $\tilde{x} = (x - \gamma)/\rho$. Taking $\gamma = 0$ and $\rho = 1$, the shape of the filter functions are calculated along the real axis and are plotted in Fig. 4, for $k = 0 \sim 3$ and $M = 16$. The sharp edges of the ideal filter function $f_k(x)$ are blunted in the practical filter $\bar{f}_k(x)$, where tails are protruding from G. These tails cause the contamination of peripheral eigenpairs in the numerical block SS method.

Due to the contamination, the dimension of the constructed eigen-subspace $\bar{n}_\Gamma$ is usually larger than n_Γ. The extra components originate from the peripheral eigenpairs. These components are more or less diminished after the operation of $\bar{M}_k$, so that they appear as singular components with small singular values in Eq. (7) or (12). To determine the effective dimension $\bar{n}_\Gamma$, a threshold ϵ are used: those singular components with singular values less than ϵ are omitted to construct the subspace. Roughly speaking, the errors of the resulting eigenvectors are about the order of ϵ. Using the singular values, it is also possible to assess the soundness of the block SS calculation: if all singular values are small, there might be no eigenvalues inside of Γ. Similarly, if no singular value is small, the constructed subspace is not large enough and we should increase either K or L.

Because the peripheral eigenvectors are expressed by inferior singular components, they are calculated less accurately than the interior eigenvectors. Sometimes, they are so erroneous that the corresponding eigenvalues appear inside of Γ as ghosts. The eigenvectors of the ghosts are mainly composed of faint singular components near the threshold. To discriminate the ghosts, the relative residuals,

$$\frac{\|(\lambda B - A)q\|_2}{|\lambda| \|q\|_2}, \tag{16}$$

work best if eigenvectors q are available. It is possible, however, to give a rough estimate on the reliability of the results, even if q is not calculated. Based on the eigenvector q of the projected system, the reliability index r is defined as

$$r = \left(\frac{\sum_{j=1}^{\bar{n}_\Gamma} s_j^{-1} |\mathsf{q}_j|^2}{\sum_{j=1}^{\bar{n}_\Gamma} |\mathsf{q}_j|^2} \right)^{-1}, \tag{17}$$

where s_j is the j-th singular value and q_j is the j-th element of q. The reliability index becomes smaller if the contribution of inferior singular components becomes larger. Note that the mathematical meaning of r is different between the Rayleigh-Ritz method and the moment-based method, though r works well as a ghostbuster in both cases.

The most time consuming part of the block SS calculation is the solution of the set of linear equations, $Y_j = (z_j B - A)^{-1} V$. Fortunately, these equations are independent of each other, so that they can be solved concurrently. This makes the block SS method attractive for modern distributed parallel architectures. It is also possible to omit the calculation for one half of the equations, if A and B are real and Γ is symmetric about the real axis. By employing a real V and a symmetric quadrature, a solution for z_j^* is given by Y_j^*, so that only the equations for $\mathrm{Im}(z_j) > 0$ need to be solved.

4 Reference Manual

The Bloss package is an eigen solver for interior eigenproblems, $(\lambda B - A)q = 0$, where A and B are n-dimensional matrices and (λ, q) are eigenpairs. The use of the Bloss package is not as simple as eigensolvers in LAPACK. You have to, at least, write some code to handle requests from the package. Besides, the following *boundary conditions* have to be determined beforehand.

Type of calculation

Whether to use the moment-based method or the Rayleigh-Ritz method.

Initial vectors

The initial subspace, which will be filtered by the moment operators. The subspace is defined by the number of basis vectors L and a set of vectors $V \in \mathbb{C}^{n \times L}$.

Contour path Γ and numerical quadrature

Those eigenvalues enclosed in Γ are to be calculated. Γ need not be circular, as far as a numerical quadrature along the path is given. The quadrature (and the path Γ itself) is defined by the number of quadrature points M, and a set of quadrature points and weights.

Modulation term

The maximum order of modulation K. You can also specify the modulation function itself. As mentioned in Sec. 3.4, the modulation term z^k in Eq. (1) is replaced by $((z - \gamma)/\rho)^k$ for numerical reasons. Indeed, the choice of the modulation term is more flexible. For example, a set of the Chebyshev polynomials $T_k((z - \gamma)/\rho)$ is a good candidate, if all the eigenvalues reside near the real axis. You can prepare an arbitrary set of modulation functions $m(z, k)$ for $k = 0 \ldots K-1$ (or $0 \ldots 2K-1$ in the moment-based method). Note, however, that an arbitrary choice of modulation functions is not allowed in the moment-based method, because the addition theorem (Eqs. (8) and (9)) only works for the power modulation function.

SVD tolerance ϵ

The cutoff threshold of the singular value. To determine the effective dimension of the filtered subspace, a singular value decomposition is performed. Those components with singular values less than ϵ are omitted to construct the eigen-subspace. Roughly speaking, the error in the resulting eigenvector is limited to the order of ϵ, if other parameters are chosen appropriately.

Parallel environment
: The base MPI communicator used by the Bloss package. In most cases, `MPI_COMM_WORLD` may suffice. You should also specify how the base communicator is divided into workers at the inversion and mat-vec phases of the calculation. In the inversion phase, the package requests the worker to solve the linear equation $Y = (\omega B - A)^{-1}V$. In the mat-vec phase, evaluations of $Y = AV$ and $Y = BV$ are requested.

Too much? Don't panic. The Bloss package offers helper functions to set up those boundary conditions. In most cases, you should choose the path Γ as well as the parameters L, K, M, and ϵ to suit to your system, and the rest of the context is set up automatically. Indeed, the example in Fig. 2 simply calls `bloss_setup_ellipse()` to perform the whole setup. In the next section, a set of functions to set up context is described, from basic setup to customization. It is followed by the sections for running context and harvesting results.

4.1 Setup context

4.1.1 Basics

■ `bloss_setup_ellipse()`

```
Bloss *bloss_setup_ellipse ( int type, MPI_Comm comm, int n, int L, int K, int M,
    complex_16 gamma, double rho, double ellipse, double tol );
```

`int type` IN

Flag to select the type of the calculation. It is specified by *or*'ing the following values:

`BLOSS_MOMENT_METHOD`
: Select the moment-based method. The Rayleigh-Ritz method is used by default.

`BLOSS_NO_EIGVEC`
: Don't calculate eigenvectors. With `BLOSS_MOMENT_METHOD`, the filtered subspace is not constructed explicitly, so that less work space is allocated internally.

`BLOSS_REAL_SYM`
: Assume a real and symmetric system for the reduced eigenproblem, $(\lambda\mathbb{B} - \mathbb{A})\mathsf{q} = 0$. For that, the matrices A and B are real-symmetric, B is positive definite, and the `BLOSS_SYM_PATH` flag should be turned on. Don't use `bloss_set_left_projector()` unless you know what you are doing. Eigenpairs are returned as real values.

`BLOSS_SYM_PATH`
: Assume the contour path (quadrature) to be symmetric about the real axis. The matrices A, B, and initial vectors should be real. In this case, the complex conjugate symmetry is utilized and only the upper half of the contour path is integrated. If you use `bloss_set_path()` to set up quadrature, you should specify only the upper half set of quadrature points. Eigenvalues λ_i are returned as complex values: if $\mathrm{Im}(\lambda_i) > 0$, λ_{i+1} is its complex conjugate. Eigenvectors are returned as real vectors, in the same manner as `dggev()` of LAPACK: if $\mathrm{Im}(\lambda_i) > 0$ and $\lambda_{i+1} = \lambda_i^*$, eigenvectors corresponding to λ_i and λ_{i+1} are $q_i + iq_{i+1}$ and $q_i - iq_{i+1}$, respectively.

`BLOSS_CALC_RESIDUALS`
: Calculate absolute residuals $\|(\lambda B - A)q\|_2$ of the resulting eigenpairs. The Bloss

package requests you to calculate $A.V$ and $B.V$ for a given set of vectors V. Note that the eigenvectors are always normalized to $\|q\|_2 = 1$.

For example, if you are to solve a complex system by using the moment-based method, and want to calculate residuals, you should set `type = BLOSS_MOMENT_METHOD | BLOSS_CALC_RESIDUALS`.

`MPI_Comm comm` IN

The base MPI communicator used for the Bloss procedure. In most cases, `MPI_COMM_WORLD` will be passed as `comm`. The base communicator is automatically divided into workers. The numbers of workers are $\min(\texttt{M}, N)$ and N for the inversion and mat-vec phases, respectively, where N is the number of ranks in `comm`. If `comm` is big enough, multiple ranks may be assigned to a worker, with which each task can be processed in parallel.

`int n` IN

Dimension of the eigensystem to be solved.

`int L` IN

Number of initial vectors. Initial vectors of `n`×`L` dimension are generated internally, and filled by random numbers. In the case of `BLOSS_SYM_PATH`, real initial vectors are generated; otherwise, they are complex.

`int K` IN

Maximum order of modulation. A power function with shift-and-scale is employed for the modulation term: $m(z, k) = ((z - \texttt{gamma})/\texttt{rho})^k$ for $k = 0 \ldots \texttt{K-1}$ (or $0 \ldots \texttt{2K-1}$ in the moment-based method).

`int M` IN

Number of quadrature points. The `M`-point trapezoidal rule is employed for the contour integral.

`complex_16 gamma, double rho, double ellipse` IN

Define a contour path. A circular path centered at `gamma` and radius `rho` is set up, which is then scaled by a factor of `ellipse` along the imaginary axis. See `bloss_prepare_path_ellipse()` for the details.

`double tol` IN

Cutoff threshold ϵ of the singular value.

`Bloss*` Return

Pointer to the newly created Bloss context. NULL is returned on error.

The function `bloss_setup_ellipse()` creates and furnishes the prototypal Bloss context, and returns a pointer to the context. The generated context may be customized further before running the interactive session. The same context should be set up on all the ranks of the base MPI communicator `comm`.

The context generated by `bloss_setup_ellipse()` will work for most of the interior eigenproblems. It is also carefully designed to work well for Hermitian systems, where all eigenvalues reside on the real axis. If you take `gamma` on the real axis and `M` as an even number, the quadrature points are arranged symmetrically about, and avoiding, the real axis. And if you take `ellipse`< 1, the elliptic contour path runs closer to the real axis, which usually gives better results for Hermitian systems.

The context is actually set up by calling several setup functions described in the following sections.

4.1.2 Create context

■ `bloss_setup()`

```
Bloss *bloss_setup ( int type, MPI_Comm comm );
```

`int type` IN
: Flag to select the calculation mode. See `bloss_setup_ellipse()` for the details.

`MPI_Comm comm` IN
: The Base MPI communicator used for the present Bloss calculation. Most of the results are accumulated on rank 0 of `comm`.

`Bloss*` Return
: Pointer to the Bloss context. NULL is returned on error.

Create and initialize the Bloss context. The consistency of the `type` flag is checked.

4.1.3 Initial vectors

■ `bloss_set_initial_vector()`

```
int bloss_set_initial_vector ( Bloss *om, int n, int L, complex_16 *V );
```

`Bloss *om` IN
: Bloss context.

`int n` IN
: Dimension of the eigensystem to be solved. This is the leading dimension of the array `V`.

`int L` IN
: Number of initial vectors.

`complex_16 *V` IN
: Complex array of `n`×`L` dimension. Initial vectors are stored in the column-major order (i.e. FORTRAN style).

`int` Return
: 0 on success (always).

Set `V` to be used as the initial subspace to be filtered. `V` should be kept intact while `om` is alive, and may be deallocated afterwards. Even in the case of `BLOSS_SYM_PATH`, where real initial vectors are required, `V` should be supplied as a complex array with a zero imaginary part. The same `V` should be set uniformly on all ranks of the base communicator.

■ `bloss_prepare_initial_vector()`

```
int bloss_prepare_initial_vector ( Bloss *om, int n, int L );
```

`Bloss *om` IN
: Bloss context.

`int n` IN
: Dimension of the eigensystem to be solved. This is used as the length of the initial vectors.

`int L` IN

Number of initial vectors to be generated.

`int` Return

0 on success, -1 on error.

Random initial vectors are generated and stored internally in the Bloss context `om`. If `BLOSS_SYM_PATH` flag is set, the generated vectors have a zero imaginary part. Otherwise, complex vectors are generated. An identical random sequence is used to fill the vectors, unless you change a seed explicitly by `bloss_set_random_seed_for_initial_vector()`. The set of initial vectors are *statistically* orthonormalized: i.e., $v_i^H \cdot v_j \to \delta_{i,j}$ at $n \to \infty$ and $n \gg L$.

■ `bloss_get_initial_vector()`

```
complex_16 *bloss_get_initial_vector ( Bloss *om );
```

`Bloss *om` IN

Bloss context.

`complex_16*` Return

Pointer to the initial vectors. The vectors are stored in the column-major order (i.e. FORTRAN style).

Returns a pointer to the initial vectors. With this function, you can access to the initial vectors generated internally by `bloss_prepare_initial_vector()`. In that case, the returned pointer is *owned* by the Bloss context `om`, so that it is discarded on destruction of `om`. If you want to keep the pointer beyond the destruction, use `bloss_detach_ptr()`.

■ `bloss_set_left_projector()`

```
int bloss_set_left_projector ( Bloss *om, complex_16 *U );
```

`Bloss *om` IN

Bloss context.

`complex_16 *U` IN

Pointer to a set of vectors that span the left projection subspace. The vectors are stored in the column-major order (i.e. FORTRAN style).

`int` Return

0 on success (always).

In the moment-based method, `U` is an `n`$\times$`L` complex array, which is used as $\tilde{V}$ in Eq. (10) to construct μ_k. In conjunction with `bloss_set_initial_vector()`, you can set $V = B\tilde{V}$, for example. The same `U` should be set on rank 0 of every `BLOSS_MPI_INVERT` worker. By default, the initial vector `V` is used as $\tilde{V}$. In the case of `BLOSS_SYM_PATH`, all the imaginary part of `U` should be zero.

In the Rayleigh-Ritz method, `U` is an `n`$\times \bar{n}_\Gamma$ complex array, where $\bar{n}_\Gamma$ is an effective dimension of the eigen-subspace determined by the singular value decomposition (Sec. 3.4). The routine should be called at the hook point `BLOSS_HOOK_RRSVD`, where $\bar{n}_\Gamma$ becomes available via `bloss_get_neig()`. If you set the left projector explicitly, `U`^{H} is used instead of $\bar{U}^H$ for the projection described in Sec. 3.2; that is, $\mathbb{A} =$ `U`$^H A\bar{U}$ and $\mathbb{B} =$ `U`$^H B\bar{U}$. The same `U` should be set on rank 0 of every `BLOSS_MPI_MATMUL` worker. By default, `U`$= \bar{U}$. In the case of `BLOSS_SYM_PATH`, the left projector should be a real array, and you have to pass a `double*` pointer as `U`.

■ `bloss_fill_random_vectors_D()`

```
int bloss_fill_random_vectors_D ( int n, int L, complex_16 *V );
```

`int n` IN
: Dimension of a vector.

`int L` IN
: Number of vectors.

`complex_16 *V` OUT
: On return, `V` is filled by random vectors with a zero imaginary part. The vectors are stored in the column-major order (i.e. FORTRAN style).

`int` Return
: 0 on success, −1 on error.

Random vectors are generated and stored in `V`. `V` should be allocated beforehand. The imaginary part of `V` is zero. The set of vectors are statistically orthonormalized. An identical random sequence is used to fill the vectors, unless you change a seed explicitly by `bloss_set_random_seed_for_initial_vector()`.

■ `bloss_fill_random_vectors_Z()`

```
int bloss_fill_random_vectors_Z ( int n, int L, complex_16 *V );
```

`int n` IN
: Dimension of a vector.

`int L` IN
: Number of vectors.

`complex_16 *V` OUT
: On return, `V` is filled by complex random vectors. The vectors are stored in the column-major order (i.e. FORTRAN style).

`int` Return
: 0 on success, −1 on error.

Complex random vectors are generated and stored in `V`. `V` should be allocated beforehand. The set of vectors are statistically orthonormalized. An identical random sequence is used to fill the vectors, unless you change a seed explicitly by `bloss_set_random_seed_for_initial_vector()`.

■ `bloss_set_random_seed_for_initial_vector()`

```
void bloss_set_random_seed_for_initial_vector ( const int seed );
```

`const int seed` IN
: Seed for the random number generator.

Set a seed for the random number generator, which is used to generate random vectors. The Fast Mersenne Twister random number generator dSFMT 2.1 [3] is employed. Note that the random sequence is reset on every entry to `bloss_fill_random_vectors_*()`, so that the same random vectors are generated unless the seed is changed explicitly.

4.1.4 Contour path / quadrature

■ `bloss_set_path()`

```
int bloss_set_path ( Bloss *om, int M, complex_16 *omega, complex_16 *weight,
    complex_16 *tau );
```

`Bloss *om` IN
: Bloss context.

`int M` IN
: Number of quadrature points.

`complex_16 *omega` IN
: Pointer to an array of the quadrature points.

`complex_16 *weight` IN
: Pointer to an array of the quadrature weights.

`complex_16 *tau` IN
: Pointer to an array of the shifted-and-scaled quadrature points.

`int` Return
: 0 on success (always).

Define a contour path Γ and a quadrature along the path. In addition to the quadrature points `omega[]`, a shifted-and-scaled set `tau[]` is also necessary, which is used to evaluate the modulation term. The shift-and-scale factor should be chosen such that the modulation term $m(\tau, k)$, which can be defined by `bloss_set_moment()`, stays around unity inside of Γ. Note that the `omega[]` → `tau[]` conversion may not necessarily be linear; the choice of `tau[]` is more flexible. The contour integral is evaluated as

$$\bar{M}_k = \sum_{i=0}^{\mathtt{M}-1} \mathtt{weight[i]} * m(\mathtt{tau[i]}, k) * (\mathtt{omega[i]} * B - A)^{-1}. \quad (18)$$

The maximum order of k is $K-1$ for the Rayleigh-Ritz method, and $2K-1$ for the moment-based method. In the case of `BLOSS_SYM_PATH`, the complex conjugate symmetry is utilized and the integral is evaluated as

$$\bar{M}_k = \sum_{i=0}^{\mathtt{M}-1} 2\,\mathrm{Re}(\mathtt{weight[i]} * m(\mathtt{tau[i]}, k) * (\mathtt{omega[i]} * B - A)^{-1}). \quad (19)$$

Only those `omega[]`s on the upper half of the complex plane should be given.

■ `bloss_set_convert_function()`

```
int bloss_set_convert_function ( Bloss *om, Bloss_convert_func *convert, void
    *RefCon );
```

`Bloss *om` IN
: Bloss context.

`Bloss_convert_func *convert` IN
: Pointer to a back-conversion function, `convert( int k, void *lambda, void *RefCon )`.

`void *RefCon` IN

Pointer to a reference constant where data required for the back-conversion may be stored.

`int` Return

0 on success (always).

Set a back-conversion function for the moment-based method. As shown in Eq. (18), the modulation term is not evaluated directly by `omega[]` but by `tau[]`. Typically, `tau[]` is a shifted-and-scaled `omega[]`, `tau[i]` $=$ (`omega[i]` $-\gamma)/\rho$ for example. In the moment-based method, the resulting eigenvalues are also shifted-and-scaled. This function registers the back-conversion function `convert()`, which is called during the moment-based method.

The function `convert()` receives an array of shifted-and-scaled eigenvalues `lambda[0..k-1]`, and should replace the content by the back-converted version. That is, if you pass `tau[]` as `lambda[]`, the content should be replaced by `omega[]`. The data necessary for the conversion can be stored in an anonymous pointer `RefCon`, which is also passed to `convert`. In the case of `BLOSS_REAL_SYM`, `lambda` is passed as a pointer to the `double` array. Otherwise, `lambda` is a `complex_16*` pointer.

■ bloss_prepare_path_ellipse()

```
int bloss_prepare_path_ellipse ( Bloss *om, int M, complex_16 gamma, double rho,
    double ellipse );
```

`Bloss *om` IN

Bloss context.

`int M` IN

Number of quadrature points.

`complex_16 gamma` IN

Center of the ellipse.

`double rho` IN

Radius of the ellipse.

`double ellipse` IN

Aspect ratio of the ellipse: the radius along the complex axis is scaled by `ellipse`.

`int` Return

0 on success, -1 on error.

Set up an elliptic path for the contour integral. A circle centered at `gamma` and radius `rho` is collapsed along the imaginary axis by a factor of `ellipse` to generate the path. The `M`-point trapezoidal rule is employed for the quadrature. In the case of `BLOSS_SYM_PATH`, the complex conjugate symmetry is utilized and the number of quadrature points is `M`/2, so that care must be taken in designing the `BLOSS_MPI_INVERT` parallel environment.

The quadrature is actually set up as follows:

$$\begin{aligned}\theta_j &= 2\pi(\mathtt{j}+\frac{1}{2})/\mathtt{M} && (20)\\ \mathtt{tau[j]} &= \cos\theta_j + i\sin\theta_j * \mathtt{ellipse} && (21)\\ \mathtt{omega[j]} &= \mathtt{rho} * \mathtt{tau[j]} + \mathtt{gamma} && (22)\\ \mathtt{weight[j]} &= \mathtt{rho} * (\cos\theta_j * \mathtt{ellipse} + i\sin\theta_j)/\mathtt{M} && (23)\end{aligned}$$

for `j = 0..M-1` (or `0..M/2-1` if `BLOSS_SYM_PATH` is used). In the moment-based method, the back-conversion function is also set.

4.1.5 Modulation term

■ `bloss_set_moment()`

```
int bloss_set_moment ( Bloss *om, int K, Bloss_moment_func *moment );
```

`Bloss *om` IN
    Bloss context.

`int K` IN
    Maximum order of the modulation term.

`Bloss_moment_func *moment` IN
    Pointer to the modulation function, `moment( int Kmax, complex_16 tau, complex_16 *res )`.

`int` Return
    0 on success (always).

Set the modulation function $m(\tau, k)$ used in Eq. (18). The maximum order `K` of the modulation term is also defined here. The shape of the filter function is modified according to $m(\tau, k)$. In the moment-based method, only the power modulation function (`bloss_moment_power()` below) is allowed as `moment`. The function `moment()` receives the maximum order `Kmax` and the shifted-and-scaled quadrature point `tau`, and should store `res[k]` $= m($`tau`$,$`k`$)$ for `k` $= 0, 1, \ldots,$ `Kmax` -1 on return. `Kmax = K` for the Rayleigh-Ritz method, and `Kmax = 2K` for the moment-based method.

■ `bloss_moment_power()`

```
void bloss_moment_power ( int Kmax, complex_16 tau, complex_16 *res );
```

The power modulation function, $m(\tau, k) = \tau^k$, to be used as the `moment` argument of `bloss_set_moment()`. It returns `res[k] = cpow(tau,k)`.

■ `bloss_moment_chebyshev()`

```
void bloss_moment_chebyshev ( int Kmax, complex_16 tau, complex_16 *res );
```

The Chebyshev polynomial modulation function, $m(\tau, k) = T_k(\tau)$, to be used as the `moment` argument of `bloss_set_moment()`. $T_k(z)$ is the k-th order Chebyshev polynomial of the first kind. It returns `res[k]` $= T_{\texttt{k}}($`tau`$)$.

4.1.6 SVD tolerance

■ `bloss_set_svd_tolerance()`

```
int bloss_set_svd_tolerance ( Bloss *om, double tol );
```

`Bloss *om` IN
    Bloss context.

`double tol` IN
    Cutoff threshold ϵ of the singular value.

`int` Return
    0 on success (always).

Set a cutoff threshold ϵ of the singular values. As mentioned in Sec. 3.4, an effective eigen-subspace is constructed by collecting finite singular value components from the filtered vectors. The components with singular values less than `tol` are ignored. Note that, rigorously speaking, the meaning of the singular values is not the same between the Rayleigh-Ritz method and the moment-based method, so that the magnitude of `tol` may be different.

4.1.7 Parallel environment

■ `bloss_set_comm()`

```
int bloss_set_comm ( Bloss *om, enum BLOSS_MPI mode, MPI_Comm worker );
```

`Bloss *om` IN
: Bloss context.

`enum BLOSS_MPI mode` IN
: Specify the phase, inversion or mat-vec, to which the divided communicator is assigned. Either `BLOSS_MPI_INVERT` or `BLOSS_MPI_MATMUL`.

`MPI_Comm worker` IN
: Communicator divided into workers.

`int` Return
: 0 on success, -1 on error.

Set how the base communicator is divided into workers at the inversion phase and the mat-vec phase. The MPI communicator `worker` may be generated by `MPI_Comm_split()` from the base communicator `comm`.

■ `bloss_prepare_comm_divide()`

```
int bloss_prepare_comm_divide ( Bloss *om, enum BLOSS_MPI mode, int div );
```

`Bloss *om` IN
: Bloss context.

`enum BLOSS_MPI mode` IN
: Specify the phase, inversion or mat-vec, for which the base communicator is divided. Either `BLOSS_MPI_INVERT` or `BLOSS_MPI_MATMUL`.

`int div` IN
: Number of workers.

`int` Return
: 0 on success, -1 on error.

Split the base communicator evenly into `div` workers for either the inversion phase or the mat-vec phase. The proximity of ranks in the base communicator is kept as much as possible.

4.2 Run context

■ `bloss_do()`

```
int bloss_do ( Bloss *om, int *task, int *L, complex_16 **V, complex_16 **Y,
    complex_16 *omega, MPI_Comm *worker );
```

`Bloss *om` IN
: Bloss context.

`int *task` OUT
: Task selector that indicates what action you should take next.

`int *L` OUT
: Size of the task.

`complex_16 **V` OUT
: Input of the task. The size of the array is `n`×`L`, where `n` is the dimension of the eigensystem to be solved. In the case of `BLOSS_SYM_PATH`, a pointer to the `double` array is actually given at the mat-vec phase. The array is stored in the column-major order (i.e. FORTRAN style).

`complex_16 **Y` IN
: Output of the task, where the result of the task should be stored. The shape of the array is identical to `V`. In the case of `BLOSS_SYM_PATH`, a pointer to the `double` array is actually given at the mat-vec phase.

`complex_16 *omega` OUT
: Input parameter of the task.

`MPI_Comm *worker` OUT
: The MPI communicator that can be used to process the task.

`int` Return
: 0 on success, -1 on error.

The function `bloss_do()` is a step-by-step driver of the Bloss interactive session. On return, `task` and other parameters are set up. You are expected to calculate `Y` = Op `V` and call `bloss_do()` again, where Op is a linear operator of either $(\texttt{omega}\,B - A)^{-1}$, A, or B. The number of columns of `V` (and `Y`) is given by `L`. Normally, only `om` and `Y` are necessary as inputs. You should not modify arguments other than `Y` between two calls of `bloss_do()`.

There are two kinds of tasks. One is a prerequisite task, for which you have to take an action. The other is a hook task, from which you can snatch progress information (or you can simply ignore it). The flow sequence of the tasks are depicted in Fig. 5. In the following, the possible values of `task` and the required response are described.

4.2.1 Prerequisites

`BLOSS_TASK_INVERT`
: The linear solution of $(\texttt{omega}\,B - A)$`Y` = `V` is requested. The `BLOSS_MPI_INVERT` communicator is set to `worker`, which may be used for the solution. `L` is the number of initial vectors. `L` and `omega` are bcasted over `worker`. The initial vectors assigned during the context setup are passed as `V`. If the number of workers is less than the number of quadrature points, several `BLOSS_TASK_INVERT` tasks with different `omega`s may be issued on the

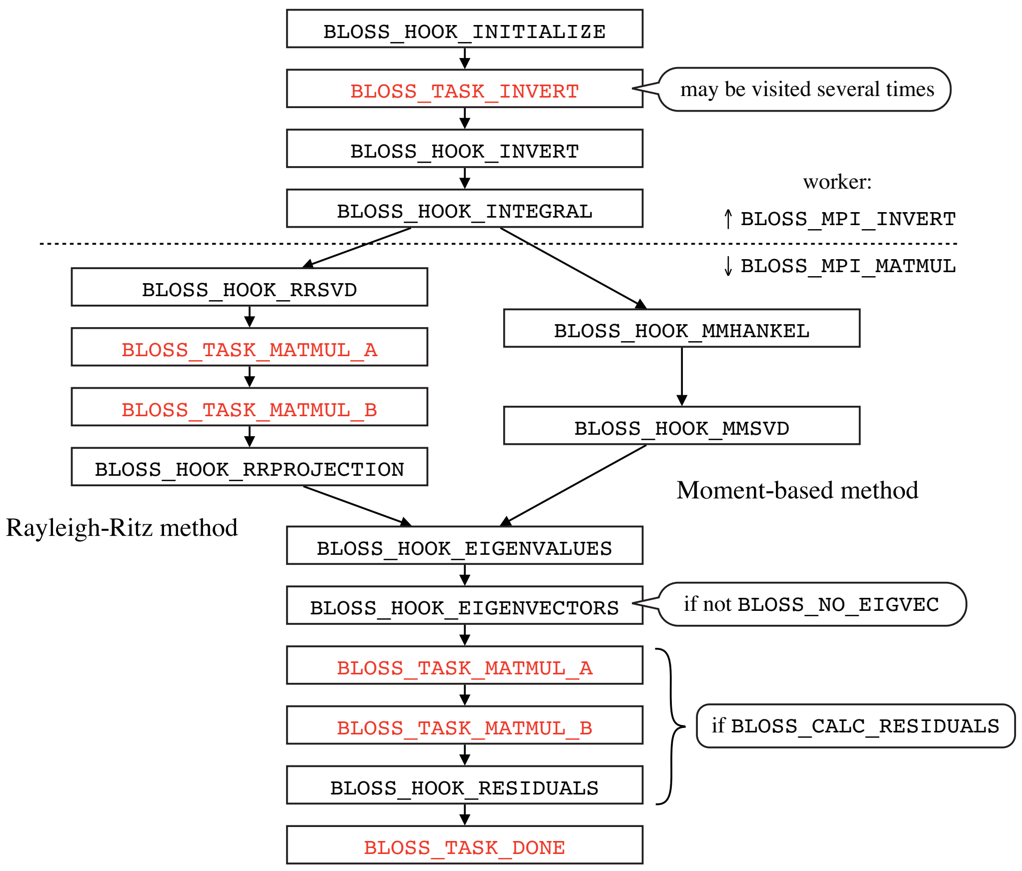

Figure 5: Bloss task sequence issued by `bloss_do()`.

same `worker`. In that case, `Y` contains the previous solution untouched. If more workers exist than quadrature points, some workers do not receive `BLOSS_TASK_INVERT`, and are kept idle during the inversion phase.

`BLOSS_TASK_MATMUL_A`

The matrix-vector multiplication of `Y` $= A$ `V` is requested. The `BLOSS_MPI_MATMUL` communicator is set to `worker`, which may be used for the operation. In the case of `BLOSS_SYM_PATH`, pointers to double arrays are actually returned as `V` and `Y`, so that a type cast may be necessary. `L` is bcasted over `worker`. Although `V` and `Y` are allocated on all ranks, `V` is set up only in rank 0 and `Y` is not zero-cleared. You should bcast `V` over `worker`, if necessary.

This task is issued at two stages: one at the Rayleigh-Ritz procedure, and the other at the residual calculation. A total of $\bar{n}_\Gamma$ vectors are divided into workers to perform the mat-vec operation in parallel. That is, if the number of workers exceeds $\bar{n}_\Gamma$, some workers are left idle. This task is always followed by `BLOSS_TASK_MATMUL_B`.

`BLOSS_TASK_MATMUL_B`

The matrix-vector multiplication of `Y` $= B$ `V` is requested. The `BLOSS_MPI_MATMUL` communicator is set to `worker`, which may be used for the operation. In the case of `BLOSS_SYM_PATH`, pointers to double arrays are actually returned in `V` and `Y`. `L` is bcasted over `worker`. Although `V` and `Y` are allocated on all ranks, `V` is set up only in rank 0 and `Y` is not zero-cleared. The same `L` and `V` are returned as in the last `BLOSS_TASK_MATMUL_A` task, so that bcasting `V` over `worker` may not be necessary.

`BLOSS_TASK_DONE`

The Bloss procedure finished successfully. Now you can move on to harvesting. But be careful! Even if `BLOSS_TASK_DONE` is returned, the results can be garbage. For example, if your estimate of $\bar{n}_\Gamma$ were too small, the calculated eigenvectors would be random linear combinations of the true eigenvectors. You should check if $LK > \bar{n}_\Gamma$ (i.e., S_R of Eq. (5) is overcomplete), and examine the residuals, the reliability indices, and other possible indicators of trouble.

`BLOSS_TASK_ERROR`
: Something has gone wrong. The context may not be furnished yet, the given parameters may be inconsistent, or even memory may be insufficient.

4.2.2 Hook points

`BLOSS_HOOK_INITIALIZE`
: Hook before starting a series of linear solver requests. The `BLOSS_MPI_INVERT` communicator is set in `worker`. The first set of `L` and `omega` are prepared and bcasted over `worker`. `L` is set to the number of initial vectors, or can be zero if the number of workers exceeds the number of quadrature points. `BLOSS_TASK_INVERT` request is not sent to such workers as `L`$= 0$. `V` is set to the initial vectors. `V` is not bcasted explicitly over `worker`, but the same `V` may already be assigned during the context setup. `Y` is allocated and zero-cleared on all ranks, unless you have assigned `BLOSS_WS_SOLUTION` by yourself via `bloss_set_workspace( )`. You may set an initial value to `Y` here, if iterative methods are to be used for the linear solver.

`BLOSS_HOOK_INVERT`
: Hook on finishing the invert requests. Here, you may release workspaces allocated for linear solvers. It is asynchronous: other `worker`s may still doing their tasks.

`BLOSS_HOOK_INTEGRAL`
: Hook on finishing contour integral and cleanup. The results are accumulated on rank 0 of the base communicator.

`BLOSS_HOOK_RRSVD`
: Hook after the singular value decomposition of S_R in the Rayleigh-Ritz method. The filtered basis vectors are orthonormalized by the singular value decomposition. The effective dimension of the filtered eigen-subspace $\bar{n}_\Gamma$ becomes available, and can be accessed via `bloss_get_neig()`. At this stage, the raw singular values `s` and the singular components U in Eq. (7) are stored in the `BLOSS_WS_SINGVALS` and `BLOSS_WS_EIGVECS` workspaces, respectively, which can be accessed via `bloss_get_workspace()`. This is available only on rank 0 of the base communicator.

  In the present version, only rank 0 of the base communicator performs SVD, which is a potential bottleneck.

`BLOSS_HOOK_RRPROJECTION`
: Hook after the Rayleigh-Ritz projection.

  The projected matrices $\mathbb{A}$ and $\mathbb{B}$ are now available on all rank 0 of `BLOSS_MPI_MATMUL` workers.

`BLOSS_HOOK_MMHANKEL`
: Hook after the construction of the Hankel matrices $\mathbb{A}$ and $\mathbb{B}$ (Eq. (11)) of the moment-based method.

  The Hankel matrices are available only on rank 0 of the base communicator.

`BLOSS_HOOK_MMSVD`

Hook after the singular value decomposition of $\mathbb{B}$ in the moment-based method. At this stage, the raw singular values `s` in Eq. (12) are stored in the `BLOSS_WS_SINGVALS` workspace, and can be accessed via `bloss_get_workspace()`. It is available only on rank 0 of the base communicator.

`BLOSS_HOOK_EIGENVALUES`

The reduced eigenvalue problem is solved, and eigenvalues are calculated. The `BLOSS_WS_SINGVALS` workspace is replaced by the reliability indices, which are available on rank 0 of the base communicator.

`BLOSS_HOOK_EIGENVECTORS`

Hook after the calculation of the eigenvectors. This hook is not visited if `BLOSS_NO_EIGVEC` is used.

The eigenvectors are calculated in parallel, using rank 0 of the `BLOSS_MPI_MATMUL` workers.

`BLOSS_HOOK_RESIDUALS`

Hook on finishing the residual calculation, which is visited if `BLOSS_CALC_RESIDUALS` is used. To calculate residuals, a pair of `BLOSS_TASK_MATMUL_A` and `BLOSS_TASK_MATMUL_B` is issued after `BLOSS_HOOK_EIGENVECTORS`. This hook comes after that.

4.3 Get results

■ `bloss_get_workspace()`

```
void *bloss_get_workspace ( Bloss *om, enum BLOSS_WORKSPACE id );
```

`Bloss *om` IN

Bloss context.

`enum BLOSS_WORKSPACE id` IN

Select a type for the results to be retrieved.

`void*` Return

Pointer to the result.

Allow access to the Bloss results stored internally in `om`. Most of them are available only on rank 0 of the base communicator. Possible values of `id` are as follows:

`BLOSS_WS_EIGVECS`

Pointer to the eigenvectors, which is an `n`$\times\bar{n}_\Gamma$ array of `complex_16` (or `double` if `BLOSS_SYM_PATH` is used). Vectors are stored in the column-major order (i.e. FORTRAN style). Use `bloss_get_neig()` to get $\bar{n}_\Gamma$. Not available if `BLOSS_NO_EIGVEC` is used.

The eigenvectors are normalized to $\|q\|_2 = 1$. Right eigenvectors are given for generalized eigenproblems.

`BLOSS_WS_EIGVALS`

Pointer to the eigenvalues, which is a `complex_16` (or `double` if `BLOSS_REAL_SYM` is used) array of length $\bar{n}_\Gamma$.

`BLOSS_WS_SINGVALS`

Pointer to the reliability indices. The workspace is a `double` array of length `L`$\times$`K`, of which top $\bar{n}_\Gamma$ elements are valid. The same workspace is used to store the raw singular values at the `BLOSS_HOOK_RRSVD` and `BLOSS_HOOK_MMSVD` stages.

`BLOSS_WS_RESIDUALS`
: Pointer to the residuals of eigenvectors, which is a `double` array of length $\bar{n}_\Gamma$. Only available if `BLOSS_CALC_RESIDUALS` is used. The residuals are calculated as $\|(\lambda B - A)q\|_2$ for eigenpairs (λ, q).

`BLOSS_WS_SOLUTION`
: Pointer to an array `Y` used in `bloss_do()`.

Please refer to `bloss_setup_ellipse()` for the parameters `n`, `L`, and `K`. You can also prepare storage areas by yourself, outside of Bloss, and set them for the workspace: see `bloss_set_workspace()`.

■ `bloss_get_neig()`

```
int bloss_get_neig ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`int` Return
: Number of eigenvalues, $\bar{n}_\Gamma$.

Returns the number of eigenvalues located inside (and on the periphery of) Γ. This is an effective dimension of the filtered eigen-subspace.

■ `bloss_get_nbase()`

```
int bloss_get_nbase ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`int` Return
: Raw dimension of the eigen-subspace, `L`×`K`.

Returns the number of basis vectors to span the eigen-subspace. If $\bar{n}_\Gamma = \mathtt{L} \times \mathtt{K}$, it is highly possible that the Bloss procedure will fail.

■ `bloss_get_rank()`

```
int bloss_get_rank ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`int` Return
: MPI rank in the base communicator.

Returns the MPI rank of the current process in the base communicator.

■ `bloss_invert_get_index()`

```
int bloss_invert_get_index ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`int` Return
: Index to the quadrature points to be calculated with the present worker.

Returns an index of the quadrature point (`i` of `omega[i]`) to be calculated with the present worker. It is available at the `BLOSS_TASK_INVERT` phase of `bloss_do()`, and only for rank 0 of `worker`.

■ `bloss_invert_get_rest()`

```
int bloss_invert_get_rest ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`int` Return
: Remaining number of quadrature points to be processed with the present worker.

Returns the remaining number of quadrature points statically assigned to the present worker. The index returned by `bloss_invert_get_index()` is exclusive. It is available at the `BLOSS_TASK_INVERT` phase of `bloss_do()`, and only for rank 0 of `worker`.

■ `bloss_residuals_get_lambda()`

```
void *bloss_residuals_get_lambda ( Bloss *om );
```

`Bloss *om` IN
: Bloss context.

`void*` Return
: Pointer to eigenvalues corresponding to eigenvectors passed at the mat-vec phase of the residual calculation.

Returns a pointer to eigenvalues currently under the residual calculation. It is available at the `BLOSS_TASK_MATMUL_A` phase of `bloss_do()` during the residual calculation, and only for rank 0 of `worker`.

The function is prepared for the one-step residual calculation. See Sec. 6.2 for the details.

4.4 Destroy context and misc.

■ `bloss_free()`

```
void bloss_free( Bloss *om );
```

`Bloss *om` IN
: Bloss context to be destroyed.

Destroys the Bloss context. All the work spaces allocated internally, including storage areas for eigenpairs, will be discarded. To allow the storage areas to survive beyond `bloss_free()`, use `bloss_detach_ptr()`.

■ `bloss_detach_ptr()`

```
int bloss_detach_ptr ( Bloss *om, void *p );
```

`Bloss *om` IN
Bloss context.

`void *p` IN
Pointer to be detached from the Bloss context.

`int` Return
0 on success, -1 on error.

Allows the pointer to survive beyond the destruction of the Bloss context. The Bloss context is equipped with an auto-release pool of pointers. Once a pointer `p` is added to the pool, `free(p )` is called on destruction of the context. Workspaces such as eigenvalues and eigenvectors are added to the pool internally, so that they are deallocated on `bloss_free()`. If you want those pointers to survive beyond the destruction of the Bloss context, you should use `bloss_detach_ptr()` to remove the pointer from the pool.

■ `bloss_attach_ptr()`

```
int bloss_attach_ptr ( Bloss *om, void *p );
```

`Bloss *om` IN
Bloss context.

`void *p` IN
Pointer to be attached to the Bloss context.

`int` Return
0 on success, -1 on error.

Allows the pointer `p` to be deallocated on the destruction of the Bloss context. That is, `free(p)` is called automatically on `bloss_free()`. Note that the deallocation is not performed recursively: if you are to attach a pointer to a complex object, you should attach not only the pointer itself, but also the pointers owned by the object.

■ `bloss_set_workspace()`

```
int bloss_set_workspace ( Bloss *om, enum BLOSS_WORKSPACE id, void *work );
```

`Bloss *om` IN
Bloss context.

`enum BLOSS_WORKSPACE id` IN
Select the workspace to be assigned.

`void *work` IN
Pointer to pre-allocated memory to be used as a workspace.

`int` Return
0 on success, -1 on error.

Assigns pre-allocated memory to a workspace used in the Bloss calculation. You should allocate enough memory to `work`, and leave it untouched during the calculation. Most of the workspaces should be aligned at a 16 byte boundary for performance reasons.

Usually, a sufficient amount of memory is allocated automatically, and you need not use `bloss_set_workspace()`. This is mainly intended for the FORTRAN interface, where workspaces should be brought from the FORTRAN world. Possible values of `id` and the required amount of memory are listed below. Please refer to `bloss_setup_ellipse()` for the parameters `n`, `L`, and `K`.

`BLOSS_WS_SOLUTION`
: Used as `Y` in `bloss_do()`. The required size is `n`×`L` of `complex_16` at the inversion phase. At the mat-vec phase, `n`×ceil($\bar{n}_\Gamma/n_{\rm wk}$) of `complex_16` (or `double` if `BLOSS_SYM_PATH` is used) is required, where $n_{\rm wk}$ is the number of `BLOSS_MPI_MATMUL` workers. If $n_{\rm wk}$ is larger than `K` (or `K/2` if `BLOSS_SYM_PATH`), `n`×`L` of `complex_16` may suffice. Note that the mat-vec phase is not visited in the moment-based method unless `BLOSS_CALC_RESIDUALS` is used.

`BLOSS_WS_EIGVECS`
: Storage area for the eigenvectors. It is also used to accumulate filtered vectors, and its subarray is used as `V` in `bloss_do()` during the mat-vec phase. Not used in the moment-based method if `BLOSS_NO_EIGVEC` is used. Required size is `n`×`L`×`K` of `complex_16` (or `double` if `BLOSS_SYM_PATH` is used).

`BLOSS_WS_EIGVALS`
: Storage area for the eigenvalues. Required size is `L`×`K` of `complex_16` (or `double` if `BLOSS_REAL_SYM` is used).

`BLOSS_WS_SINGVALS`
: Storage area for the reliability indices. Also used to store the singular values. Required size is `L`×`K` of `double`.

`BLOSS_WS_RESIDUALS`
: Storage area for the residuals. Required size is `L`×`K` of `double`.

■ `bloss_dump_summary()`

```
void bloss_dump_summary ( Bloss *om, FILE *fp );
```

`Bloss *om` IN
: Bloss context.

`FILE *fp` IN
: File pointer, to which the summary is dumped.

Outputs summary of the Bloss context.

■ `bloss_dump_quadrature()`

```
void bloss_dump_quadrature ( Bloss *om, FILE *fp );
```

`Bloss *om` IN
: Bloss context.

`FILE *fp` IN
: File pointer, to which the quadrature data are dumped.

Outputs the quadrature data for the contour integral. The `omega-weight-tau` triad is dumped in a text format.

5 FORTRAN interface

FORTRAN interfaces are prepared in the Bloss package. Just as in the C implementation, users have to write their own code to set up and communicate with the Bloss context. Most of the Bloss subroutines are parallel to the C APIs, though the running context part is a bit different. The FORTRAN code corresponding to the C tutorial (Fig. 2) is listed in Fig. 6. A complete code, including pre- and post-processes, is available at `examples/Fortran/tutorial.f`.

INPUT

`integer n`	dimension of the system
`complex*16 A(n*n), B(n*n)`	input matrices
`complex*16 gamma`	center of the contour path Γ
`real*8 rho`	radius of the contour path Γ
`integer M`	number of quadrature points (~ 32)
`integer L`	number of initial vectors (~ 8)
`integer K`	maximum order of modulation (~ 8)
`integer Comm`	base MPI communicator

All the INPUT data should be `MPI_Bcast`-ed over all the ranks of the communicator `Comm`.

OUTPUT

`integer neig`	number of eigenpairs obtained ($\leq$ `L*K`)
`complex*16 lambda(L*K)`	eigenvalues
`complex*16 q(n*L*K)`	eigenvectors

OUTPUT data are available only on rank 0 of `Comm`. The storage area `lambda(L*K)` and `q(n*L*K)` should be allocated on the caller side.

The Bloss context `om` is set up by calling `bloss_setup_ellipse()`. This subroutine is almost identical to its C counterpart, except that users have to supply workspaces `V`, `Y`, and `q`. Enough memory should be allocated for them beforehand. `V(n*L)` is a workspace to store the initial vectors, which is filled by random vectors in `bloss_setup_ellipse()`. The eigenvectors are stored in `q(n*L*K)`, which is also used to accumulate intermediate results. `Y(*)` is used to communicate with the Bloss context. The size of `Y` is `Y(n*L*K)` for safety, but if the number of processors is large enough, it can be as small as `Y(n*L)`.

After setting up the context, the interactive session is started by calling `bloss_do()`. In response to `bloss_do()`, users should store either `Y = (omega*B - A)`$^{-1}$`.V`, `Y = A.q(idx)`, or `Y = B.q(idx)`, depending on `task`, and call `bloss_do()` again. The leading dimension of `V`, `Y`, and `q(idx)` is `n`, and the number of columns is `num`. For these operations, users can utilize a small MPI communicator `worker`, which is generated by dividing the base communicator `Comm`. The Bloss procedure finishes if `task` is returned with 5963 (=`BLOSS_TASK_DONE`). Eigenvalues should be copied out from the Bloss context by `bloss_get_lambda()`, or they are lost on calling `bloss_free()` to destroy the context. Note that, different from the C example, eigenvectors stored in the workspace `q` are not touched on the destruction.

In the following, the FORTRAN APIs are documented. Because most of them are simple wrappers of the equivalent C APIs, only the differences will be described. Please refer to Sec. 4 for the missing details. Note that some of the C APIs, such as `bloss_set_moment()` and `bloss_set_convert_function()`, are not supported in FORTRAN. They are only accessible indirectly, via `bloss_set_workspace()` and `bloss_prepare_path_ellipse()`.

```
      subroutine tutorial( n, A, B, gamma, rho, M, L, K, neig, lambda,
     &      q, Comm )
      implicit none
      integer n, M, L, K, neig, Comm
      real*8 rho
      complex*16 A(*), B(*), gamma, lambda(*), q(*)
      !
      complex*16, allocatable :: V(:), Y(:)
      integer, parameter :: type = 0
      real*8, parameter :: ellipse = 1.0, tolerance = 1d-12
      !
      integer om, task, ierr, nproc, i, num, idx, worker
      complex*16 omega

      call MPI_Comm_size(Comm, nproc, ierr)

      ! Choose optimal size for Y:
      !   - at least L columns are required  (INVERT step)
      !   - can be as small as L * K / nproc (MATMUL step)
      i = L * K / nproc
      if (i * nproc .lt. L * K) i = i + 1
      if (i .lt. L) i = L
      allocate( V(n*L), Y(n*i) )

      call bloss_setup_ellipse( type, Comm, n, L, K, M, gamma, rho,
     &     ellipse, tolerance, V, Y, q, om, ierr)

      do
         call bloss_do( om, task, num, omega, worker, idx, ierr )

         if (task .eq. 1) then                 ! BLOSS_TASK_INVERT
            call linear_solve(n, A, B, omega, V, num, Y)
         else if (task .eq. 11) then           ! BLOSS_TASK_MATMUL_A
            call mat_vec( n, A, q(idx), num, Y)
         else if (task .eq. 12) then           ! BLOSS_TASK_MATMUL_B
            call mat_vec( n, B, q(idx), num, Y)
         else if (task .eq. 5963) then         ! BLOSS_TASK_DONE
            call bloss_get_rank(om, i, ierr)
            if (i .eq. 0) then
               call bloss_get_neig(om, neig, ierr)
               call bloss_get_lambda(om, lambda, ierr)
            end if
            call bloss_free(om)
            deallocate( V, Y )
            return
         end if
      end do

      end
```

Figure 6: FORTRAN code used to solve the tutorial example.

5.1 Setup context

5.1.1 Basics

■ `bloss_setup_ellipse( type, comm, n, L, K, M, gamma, rho, ellipse, tol, V, Y, q, om, ierr )`

```
integer             type, comm, n, L, K, M, om, ierr
real*8              rho, ellipse, tol
complex*16/real*8   q(n*L*K)
complex*16          gamma, V(n*L), Y(n*L*K)
```

`integer type` IN

Flag used to select the calculation mode. It is specified by adding the following values:

1	`BLOSS_MOMENT_METHOD`
2	`BLOSS_NO_EIGVEC`
4	`BLOSS_REAL_SYM`
8	`BLOSS_SYM_PATH`
16	`BLOSS_CALC_RESIDUALS`

For example, if you are to solve a complex system by using the moment-based method, and want to calculate residuals, you should set `type = 17`. See p. 11 for the details.

`complex*16 V(n*L)` IN

Workspace used to store the initial vectors. `V` is filled by random vectors, so that you only need to allocate enough memory for `V`.

`complex*16 Y(n*L*K)` IN

Workspace used to store results in response to the preceding `bloss_do()` call. Please refer to `BLOSS_WS_SOLUTION` of `bloss_set_workspace()` (p. 26) for the required memory. If the number of ranks in `comm` is large enough, the size can be as small as `Y(n*L)`.

`complex*16/real*8 q(n*L*K)` IN

Workspace used to store eigenvectors. It is also used to accumulate intermediate results. It should be `real*8` if `BLOSS_SYM_PATH` is used, and be `complex*16` otherwise.

`integer om` OUT

ID of the newly created Bloss context.

`integer ierr` OUT

0 on success, -1 on error.

Create and furnish the Bloss context. In addition to the equivalent C API, workspaces used in the Bloss procedure are explicitly specified here. These workspaces are used to communicate with the Bloss context during the interactive session.

The context is actually set up by calling several subroutines described below.

5.1.2 Create context

■ `bloss_setup( type, comm, om, ierr )`

```
integer     type, comm, om, ierr
```

`integer type` IN

Flag used to select the calculation mode. See `bloss_setup_ellipse()` for the details.

`integer om` OUT

ID of the newly created Bloss context.

`integer ierr` OUT

0 on success, -1 on error.

Create and initialize the Bloss context.

■ `bloss_set_workspace( om, n, L, K, V, Y, q, ierr )`

```
integer             om, n, L, K, ierr
complex*16/real*8   q(n*L*K)
complex*16          V(n*L), Y(n*L*K)
```

`integer om` IN

ID of the Bloss context.

`integer n` IN

Dimension of the eigensystem to be solved.

`integer L` IN

Number of initial vectors.

`integer K` IN

Maximum order of modulation.

`complex*16 V(n*L)` IN

Initial vectors.

`complex*16 Y(n*L*K)` IN

Workspace used to store results in response to `bloss_do()`.

`complex*16/real*8 q(n*L*K)` IN

Workspace used to store eigenvectors. It is also used to accumulate intermediate results. It should be `real*8` if `BLOSS_SYM_PATH` is used, and be `complex*16` otherwise.

`integer ierr` OUT

0 on success, -1 on error.

Assign workspaces for the Bloss context. It is not a simple wrapper of the C API of the same name, but a mixture of `bloss_set_initial_vector()`, `bloss_set_workspace()`, and `bloss_set_moment()`. You should allocate enough memory for the workspaces. The subroutine should be called on every rank of the base communicator, where identical initial vectors should be set in `V`. To prepare the initial vectors, you can use utility functions `bloss_fill_random_vectors_*()`.

`Y` is a workspace used to return results in response to the preceding `bloss_do()` call. The required size shown here (`Y(n*L*K)`) is for safety; please refer to `BLOSS_WS_SOLUTION` of `bloss_set_workspace()` (p. 26) for the minimal size. Basically, it can be as small as `Y(n*L)`, if the number of `BLOSS_MPI_MATMUL` workers is large enough.

At the end of the Bloss procedure, eigenvectors are returned in `q`, which is also used to accumulate intermediate results. An index to `q` is used in `bloss_do()` to specify a subarray, so that `q` is type-sensitive: it should be `real*8` if `BLOSS_SYM_PATH` is used, and be `complex*16` otherwise. The workspace `q` will not be used in the moment-based method if `BLOSS_NO_EIGVEC` is used.

The modulation function $m(\tau, k)$ (Eq. (18)) is set to a simple power function. Other modulation functions are not supported in FORTRAN (i.e., `bloss_set_moment()` is missing).

5.1.3 Initial vectors

■ `bloss_set_left_projector( om, U, ierr )`

```
integer      om, ierr
complex*16   U(*)
```

See p. 14.

■ `bloss_fill_random_vectors_D( n, L, V, ierr )`

```
integer      n, L, ierr
complex*16   V(n*L)
```

See p. 15.

■ `bloss_fill_random_vectors_Z( n, L, V, ierr )`

```
integer      n, L, ierr
complex*16   V(n*L)
```

See p. 15.

■ `bloss_set_random_seed_for_initial_vector( seed, ierr )`

```
integer      seed, ierr
```

See p. 15.

5.1.4 Contour path / quadrature

■ `bloss_set_path( om, M, omega, weight, tau, ierr )`

```
integer      om, M, ierr
complex*16   omega(M), weight(M), tau(M)
```

See p. 16. In the moment-based method, the resulting eigenvalues are shifted-and-scaled in the same way as the `omega`→`tau` conversion. Because the back-conversion function cannot be set in FORTRAN (`bloss_set_convert_function()` is missing), you should convert eigenvalues by yourself. `BLOSS_CALC_RESIDUALS` is also prohibited, because the true eigenvalues are unknown. Note that the above rule is not applied if you use `bloss_prepare_path_ellipse()` (or `bloss_setup_ellipse()`) to set up the path; the back-conversion function is set up there.

■ `bloss_prepare_path_ellipse( om, M, gamma, rho, ellipse, ierr )`

```
integer      om, M, ierr
real*8       rho, ellipse
complex*16   gamma
```

See p. 17.

5.1.5 SVD tolerance

■ `bloss_set_svd_tolerance( om, tol, ierr )`

```
integer      om, ierr
real*8       tol
```

See p. 18.

5.1.6 Parallel environment

■ `bloss_set_comm( om, mode, worker, ierr )`

```
integer      om, mode, worker, ierr
```

`integer mode` IN

Specify the phase, inversion or mat-vec, to which the divided communicator `worker` is assigned..

0	BLOSS_MPI_INVERT
1	BLOSS_MPI_MATMUL

See p. 19.

■ `bloss_prepare_comm_divide( om, mode, div, ierr )`

```
integer      om, mode, div, ierr
```

`integer mode` IN

Specify the phase, inversion or mat-vec, for which the base communicator is divided.

0	BLOSS_MPI_INVERT
1	BLOSS_MPI_MATMUL

See p. 19.

5.2 Run context

■ `bloss_do( om, task, L, omega, worker, idx, ierr )`

```
integer      om, task, L, worker, idx, ierr
complex*16   omega
```

`integer task` OUT

Task selector that indicates what action you should take next. Possible values are the following:

1	BLOSS_TASK_INVERT
11	BLOSS_TASK_MATMUL_A
12	BLOSS_TASK_MATMUL_B
51	BLOSS_HOOK_INITIALIZE
52	BLOSS_HOOK_INVERT
53	BLOSS_HOOK_INTEGRAL
54	BLOSS_HOOK_RRSVD

55	`BLOSS_HOOK_RRPROJECTION`
56	`BLOSS_HOOK_MMHANKEL`
57	`BLOSS_HOOK_MMSVD`
58	`BLOSS_HOOK_EIGENVALUES`
59	`BLOSS_HOOK_EIGENVECTORS`
60	`BLOSS_HOOK_RESIDUALS`
5963	`BLOSS_TASK_DONE`
-1	`BLOSS_TASK_ERROR`

See p. 20 for the details.

`integer idx` OUT

Index to the `q` workspace for which the mat-vec operation (`task = 11` or `12`) should be performed.

Step-by-step driver of the Bloss interactive session. Depending on `task`, you should calculate either:

```
task .eq. 1   ⇒ Y = (omega*B - A)^-1.V,
task .eq. 11  ⇒ Y = A.q(idx), or
task .eq. 12  ⇒ Y = B.q(idx),
```

and call `bloss_do()` again. The workspaces `V`, `Y`, and `q` should be set beforehand via `bloss_set_workspace()`. The leading dimension of `V`, `Y`, and `q(idx)` is `n`, and the number of columns is `L`. See p. 20 for other details.

5.3 Get results

■ `bloss_get_lambda( om, lambda, ierr )`

```
integer     om, ierr
real*8      lambda(L*K)
```

`integer om` IN

ID of the Bloss context.

`real*8 lambda(L*K)` OUT

Eigenvalues.

`integer ierr` OUT

0 on success, -1 on error.

Replace the top $\bar{n}_\Gamma$ elements of `lambda` with the calculated eigenvalues. You should allocate enough memory for `lambda`.

■ `bloss_get_sv( om, sv, ierr )`

```
integer     om, ierr
real*8      sv(L*K)
```

`integer om` IN

ID of the Bloss context.

`real*8 sv(L*K)` OUT

Reliability indices.

`integer ierr` OUT
    0 on success, -1 on error.

Replace elements of `sv` with the reliability indices, of which the top $\bar{n}_\Gamma$ elements are valid. If this is called at the `BLOSS_HOOK_RRSVD` and `BLOSS_HOOK_MMSVD` stages, `sv` is filled by the raw singular values. You should allocate enough memory for `sv`.

■ `bloss_get_residuals( om, residual, ierr )`

```
integer     om, ierr
real*8      residual(L*K)
```

`integer om` IN
    ID of the Bloss context.

`real*8 residual(L*K)` OUT
    Residuals of the eigenvectors.

`integer ierr` OUT
    0 on success, -1 on error.

Replace the top $\bar{n}_\Gamma$ elements of `residual` with the absolute residuals $\|(\lambda B - A)q\|_2$ of the eigenpairs. Only available if `BLOSS_CALC_RESIDUALS` is used. You should allocate enough memory for `residual`.

■ `bloss_get_neig( om, neig, ierr )`

```
integer     om, neig, ierr
```

See p. 24.

■ `bloss_get_nbase( om, nbase, ierr )`

```
integer     om, nbase, ierr
```

See p. 24.

■ `bloss_get_rank( om, rank, ierr )`

```
integer     om, rank, ierr
```

See p. 24.

■ `bloss_invert_get_index( om, idx, ierr )`

```
integer     om, idx, ierr
```

`integer idx` OUT
    Index to the quadrature points to be calculated with the present worker.

`integer ierr` OUT
    0 on success, -1 on error.

Returns an index of the quadrature point (`idx` of `omega(idx)`) to be calculated with the present worker. This is available at the `BLOSS_TASK_INVERT` phase of `bloss_do()`, and only for rank 0 of `worker`.

■ `bloss_invert_get_rest( om, num, ierr )`

```
integer     om, num, ierr
```

See p. 25.

■ `bloss_residuals_get_lambda( om, L, lambda, ierr )`

```
integer     om, L, ierr
real*8      lambda(L)
```

`integer L` IN
Number of eigenvalues to copy.

`real*8 lambda(L)` OUT
Eigenvalues.

`integer ierr` OUT
0 on success, −1 on error.

Replace the top `L` elements of `lambda` with the eigenvalues under the residual calculation. This is available at the `BLOSS_TASK_MATMUL_A` phase of `bloss_do()` during the residual calculation, and only for rank 0 of `worker`. You should allocate enough memory for `lambda`.

The subroutine is prepared for the one-step residual calculation. See Sec. 6.2 for the details.

5.4 Destroy context

■ `bloss_free( om, ierr )`

```
integer     om, ierr
```

`integer om` IN
ID of the Bloss context.

`integer ierr` OUT
0 on success, −1 on error.

Destroys the Bloss context associated with `om`. Memories allocated internally are deallocated, so that eigenvalues and residuals become inaccessible. Externally assigned workspaces, such as eigenvectors `q`, are not touched.

6 Advanced topics

6.1 Iterative refinement

After the `BLOSS_HOOK_INTEGRAL` hook point, you can set `BLOSS_TASK_INITIALIZE (= 0)` to `task` and call `bloss_do()` to restart the Bloss procedure. The restart feature may be used for iterative refinement. First, apply the zero-th order filter (`K=1`) on the initial vector `V`. At the `BLOSS_HOOK_INTEGRAL` hook point, a set of filtered vectors is stored in the eigenvector workspace (`BLOSS_WS_EIGVECS`), which can be set to `V` and enable restart of the Bloss procedure with a larger `K`. If complete factorization is employed for the inversion, and the `BLOSS_MPI_INVERT` parallel environment is set up appropriately, the LU factorization can be reused in the restarted procedure. In this case, the cost for the refinement will be negligible.

6.2 One-step residual calculation

During the residual calculation, `bloss_do()` issues `BLOSS_TASK_MATMUL_A` and `BLOSS_TASK_MATMUL_B` tasks in sequence. To calculate the residuals at one step, you can supply `Y` $= (\lambda B - A)$`V` for the former request. The eigenvalues λ corresponding to `V` are available via `bloss_residuals_get_lambda()`. To indicate that the residuals are calculated in the one-step manner, you should return `Y = NULL` in the subsequent `BLOSS_TASK_MATMUL_B` task (or return `idx = -1` in FORTRAN). Note that, in the Rayleigh-Ritz method, `BLOSS_TASK_MATMUL_*` tasks are issued both at the Rayleigh-Ritz procedure and the residual calculation. When using the one-step feature in the Rayleigh-Ritz method, users should be careful to switch the action between them.

6.3 Non-linear eigenproblem

The Bloss package is also capable of solving non-linear λ-matrix eigenproblems. The λ-matrix $M(\lambda)$ is a matrix whose elements are regular functions of λ, and its eigenproblem is to find eigenpairs (λ, q) that satisfy $M(\lambda)q = 0$. The λ-matrix eigenproblem can be solved by replacing $(zB - A)^{-1}$ in Eq. (1) by $M(z)^{-1}$. See [4] for the theories behind this. Unfortunately, the Rayleigh-Ritz method is not applicable, and only the moment-based method works. On running the Bloss context, `bloss_do()` returns a `BLOSS_TASK_INVERT` request, where you should solve M(`omega`)`Y` $=$ `V` for `Y`. If you set `BLOSS_CALC_RESIDUALS`, `bloss_do()` issues `BLOSS_TASK_MATMUL_A` and `BLOSS_TASK_MATMUL_B` requests, where you can utilize the one-step residual calculation feature described in the previous section. That is, for the `BLOSS_TASK_MATMUL_A` request, you should return `Y` $= M(\lambda)$`V`, and for the subsequent `BLOSS_TASK_MATMUL_B` request, return `Y = NULL`.

7 Acknowledgments

The development of the Bloss package was partly supported by KAKENHI, Grant-in-Aid for Scientific Research 21246018.

8 Conditions of use

You shall acknowledge (using following references [1] and [2]) the contribution of this package in any publication of material dependent upon the use of the package.

References

[1] Ikegami, T., Sakurai, T. and Nagashima, U.: A filter diagonalization for generalized eigenvalue problems based on the Sakurai-Sugiura projection method, *J. Comp. Appl. Math.*, Vol. 233, pp. 1927–1936 (2010).

[2] Ikegami, T. and Sakurai, T.: Contour integral eigensolver for non-Hermitian systems: a Rayleigh-Ritz-type approach, *Taiwanese J. Math.*, Vol. 14, pp. 825–837 (2010).

[3] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html`

[4] Ikegami, T., Sakurai, T. and Tadano, H.: Parallel eigensolver for large scale non-linear systems, Proceedings of ICNAAM 2010, Rhodes, 2010/09/20.