z-Pares Users' Guide

Release 0.9.5

Yasunori Futamura and Tetsuya Sakurai
University of Tsukuba

July 03, 2014

Contents

1 Introduction **1**
- 1.1 Features . . . 1
- 1.2 Dependences . . . 1

2 Getting started **3**
- 2.1 Installation . . . 3
- 2.2 Example . . . 4
- 2.3 Basic concepts in z-Pares . . . 6
- 2.4 Two-level MPI parallelism . . . 6
- 2.5 `zpares_prm` derived type . . . 7
- 2.6 `zpares` module . . . 7
- 2.7 Initialization and finalization . . . 8
- 2.8 Memory allocation . . . 8
- 2.9 Setting a circle . . . 9
- 2.10 Relative residual norm . . . 9
- 2.11 Reverse communication interface . . . 10
- 2.12 Reverse communicaton on two-level MPI parallelism . . . 11
- 2.13 Efficient implementations for specific problems . . . 15

3 General use of z-Pares **16**
- 3.1 Naming convention of z-Pares routines . . . 16
- 3.2 Input and output arguments in z-Pares routines . . . 17
- 3.3 Parameters in `zpares_prm` . . . 20
- 3.4 Extracting method for eigenpairs . . . 21
- 3.5 Reverse communicaton interface in complex Hermitian case . . . 22
- 3.6 Estimations of eigenvalue count . . . 23
- 3.7 Indicator for spurious eigenvalues . . . 23
- 3.8 Error diagnostics in `info` . . . 23

4 Advanced features **24**
- 4.1 User defined quadrature rule . . . 24

Chapter 1

Introduction

z-Pares is a package for solving generalized eigenvalue problems

$$A\boldsymbol{x} = \lambda B\boldsymbol{x},$$

where A and B are real or complex square matrices , and λ and $\boldsymbol{x}$ are an eigenvalue and an eigenvector, respectively. z-Pares is designed to compute a few eigenvalues and eigenvectors of sparse matrices. The symmetries and definitenesses of the matrices can be exploited suitably. z-Pares implements a complex moment based contour integral eigensolver. z-Pares computes eigenvalues inside a user-specified contour path and corresponding eigenvectors. The most important feature of z-Pares is two-level Message Passing Interface (MPI) distributed parallelism.

1.1 Features

The main features of z-Pares are described below.

- Implemented in Fortran 90/95
- Solves standard eigenvalue problems $A\boldsymbol{x} = \lambda\boldsymbol{x}$ and generalized eigenvalue problems $A\boldsymbol{x} = \lambda B\boldsymbol{x}$
- Computes eigenvalues located in an interval or a circle and corresponding (right) eigenvectors
- Both real and complex type are supported
- Single precision and double precision are supported
- Both sequatial and distributed parellel MPI builds are available
- Two-level distributed parallelism can be utilized using a pair of MPI communicators
- Reverse communication mechanism is used so that the package accept any matrix data structures
- Interfaces for dense and sparse CSR format are available (only with 1-level distributed parallelism)

1.2 Dependences

z-Pares depends on follwing packages:

- BLAS/LAPACK
- Message Passing Interface (MPI-2 standard)
- MUMPS 4.10.0 (optional)

BLAS/LAPACK should be installed and MPI is needed for the parallel version of z-Pares. MUMPS is required to use the sparse CSR interface.

Chapter 2

Getting started

2.1 Installation

Download z-Pares from

http://zpares.cs.tsukuba.ac.jp/download/zpares_0.9.5.tar.gz

then unarchive the tar.gz file as

```
% tar zxf zpares_0.9.5.tar.gz
```

In what follows we refer the extracted directry to as `$ZPARES_HOME`. Then

```
% cd $ZPARES_HOME
```

Copy a sample of `make.inc` in directry `$ZPARES_HOME/Makefile.inc/` to `$ZPARES_HOME/make.inc`. For example:

```
% cp Makefile.inc/make.inc.par make.inc
```

Then modify `make.inc` according to the user's preferense. If the user needs to make MPI version, specify

```
USE_MPI = 1
```

in `make.inc`. Otherwize (sequencial version),

```
USE_MPI = 0
```

If the user needs sparse CSR interface,

```
USE_MUMPS = 1
```

should be added. When `USE_MUMPS = 1`, before the user types command `make`, the user should install MUMPS available at

http://mumps.enseeiht.fr/

and specify the MUMPS installed directry path to `MUMPS_DIR`.

Now

```
% make
```

to build z-Pares.

Now you have `libzpares.a` in `$ZPARES_HOME/lib/` , and `zpares.mod` in `$ZPARES_HOME/include/`. If you set `USE_MUMPS = 1` in `make.inc`, you also have `libzpares_mumps.a` in `$ZPAERS_HOME/lib/`

For advanced users

If the user installed MUMPS with external ordering packages (e.g. METIS, SCOTCH), specify the paths to the package libraries to variable `MUMPS_DEPEND_LIBS` in `make.inc` in order to compile the examples.

2.2 Example

Here, we describe an example to briefly show how a user code using z-Pares looks like. The problem in this example is a generalized eigenvalue problem with

$$A = \begin{pmatrix} 1+\mathrm{i} & 1 & & & 0 \\ & 2+\mathrm{i} & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ 0 & & & & 100+\mathrm{i} \end{pmatrix} \in \mathbb{C}^{100\times 100}$$

and

$$B = \begin{pmatrix} I_{50} & & 0 \\ & 0 & \\ & & \ddots & \\ 0 & & 0 \end{pmatrix} \in \mathbb{R}^{100\times 100},$$

where i is the imaginary unit and I_{50} is the identity matrix of order 50. The finite eigenvalues are $\{1+\mathrm{i}, 2+\mathrm{i}, \ldots, 50+\mathrm{i}\}$. This example uses the z-Pares with the following conditions:

- A generalized eigenproblem with complex non-Hermitian matrices is solved
- Dense interface is used
- The higher-level parallelism (described in Section 2.4 (page 6)) is only used

We set a circle contour path so that it encloses eigenvalues $\{1+\mathrm{i}, 2+\mathrm{i}, \ldots, 12+\mathrm{i}\}$.

```
program main
  use zpares
  implicit none
  include 'mpif.h'
  integer, parameter :: mat_size = 100
  integer :: num_ev, info, i, j, L, N, M, Lmax, ncv, ierr, myrank
  double precision :: right
  double precision, allocatable :: res(:)
  complex(kind(0d0)) :: left
  complex(kind(0d0)), allocatable :: A(:,:), B(:,:), eigval(:), X(:,:)
  type(zpares_prm) :: prm

  call MPI_INIT(ierr)
```

```
   call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

   allocate(A(mat_size,mat_size), B(mat_size,mat_size))

   ! Setting the matrices
   A = (0d0, 0d0)
   B = (0d0, 0d0)
   do i = 1, mat_size
      A(i,i) = cmplx(i, 1d0)
      if ( i /= 1 )  A(i-1,i) = 1d0
      if ( i <= 50 )  B(i,i) = 1d0
   end do

   ! Initialization
   call zpares_init(prm)

   ! Change parameters from default values
   prm%high_comm = MPI_COMM_WORLD
   prm%L = 16
   prm%M = 6

   ! Get the number of column vectors
   ncv = zpares_get_ncv(prm)
   ! Memory allocation
   allocate(eigval(ncv), X(mat_size, ncv), res(ncv))

   ! Setting the circle
   left = cmplx(0d0, 1d0) ! The position of the left edge of the circle
   right = 12.5d0 ! The real part of the position of the right edge of the circle

   ! Call a z-Pares main subroutine
   call zpares_zdnsgegv &
   (prm, mat_size, A, mat_size, B, mat_size, left, right, num_ev, eigval, X, res, info)

   ! Finalization
   call zpares_finalize(prm)

   ! Show the result
   if ( myrank == 0 ) then
      write(*,*) 'This is an example for a non-Hermitian complex generalized'
      write(*,*) 'eigenvalue problem using a dense interface.'; write(*,*)
      write(*,*) 'Result : '
      write(*,'(A6,1X,"|",A41,1X,"|",A10)') 'Index', 'Eigenvalue', 'Residual'
      do i = 1, num_ev
         write(*,'(I6,2X,F18.15,1X,"+",1X,F18.15,1X,"i",2X,1pe10.1)') &
              i, real(eigval(i)), aimag(eigval(i)), res(i)
      end do
   end if

   call MPI_FINALIZE(ierr)
   deallocate(A, B, eigval, X, res)
end program main
```

The result will be:

```
 This is an example for a non-Hermitian complex generalized
 eigenvalue problem using a dense interface.
```

```
Result :
Index |                              Eigenvalue |  Residual
    1   0.999999999999999 +  1.000000000000001 i     2.4E-13
    2   1.999999999999999 +  0.999999999999999 i     1.1E-13
    3   3.000000000000001 +  1.000000000000001 i     1.3E-13
    4   4.000000000000002 +  1.000000000000001 i     1.4E-13
    5   5.000000000000013 +  1.000000000000000 i     2.0E-13
    6   5.999999999999999 +  0.999999999999998 i     6.0E-14
    7   7.000000000000010 +  1.000000000000000 i     2.4E-14
    8   7.999999999999995 +  0.999999999999997 i     2.7E-14
    9   9.000000000000012 +  1.000000000000000 i     6.9E-14
   10  10.000000000000002 +  0.999999999999998 i     4.8E-14
   11  11.000000000000018 +  1.000000000000006 i     6.7E-14
   12  12.000000000000021 +  0.999999999999998 i     1.3E-13
```

2.3 Basic concepts in z-Pares

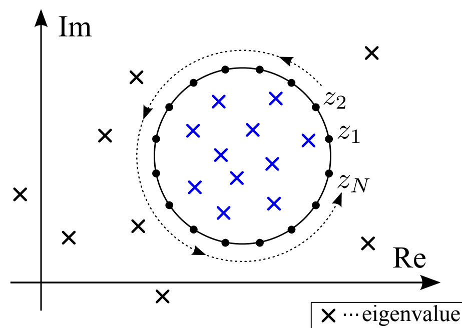

Figure 2.1: z-Pares computes eigenvalues located inside a contour path on the complex plane. (blue cross)

z-Pares implements an eigensolver which uses the contour integration and the complex moment. The eigensolver computes eigenvalues located inside a contour path on the complex plane. The corresponding eigenvectors are also computed. As described in Figure 2.1 numerical quadrature with N quadrature points is used to approximate the contour integral. The basis of the subspace which is used for extracting eigenpairs are computed by solving linear systems with mulitple right hand sides

$$(z_j B - A)Y_j = BV \quad (j = 1, 2, \dots, N)$$

for Y_j, where z_j is a quadrature points, and V and Y_j are n-by-L matrices. V is called source matrix and its column vector is called source vector.

Since the linear systems can be solved independently, the computations can be embarrassingly parallelized. Addtionaly, each linear system can be solved in parallel.

2.4 Two-level MPI parallelism

Above the parallelism of quadrature points, there is independent parallelism if multiple contour paths are given. Here we define three levels of parallelism:

- **Top level** : Parallelism of computations on contour paths

- **Middle level** : Parallelism of computaitons on quadrature points
- **Bottom level** : Parallelism of computations for solving linear systems

z-Pares utilizes the parallelism at the middle level and the bottom level using a pair of MPI communicators. The MPI communicator which manages the middle level and the bottom level is refered to as *the higher-level communicator*, and *the lower-level communicator*, respectively. Since the top-level parallelism can be implemented completely without communications, we have not added the implementation of this level to the feature of z-Pares. The users should manage the top level parallelism theirselves if needed.

The above descriptions are illustrated in Figure 2.2.

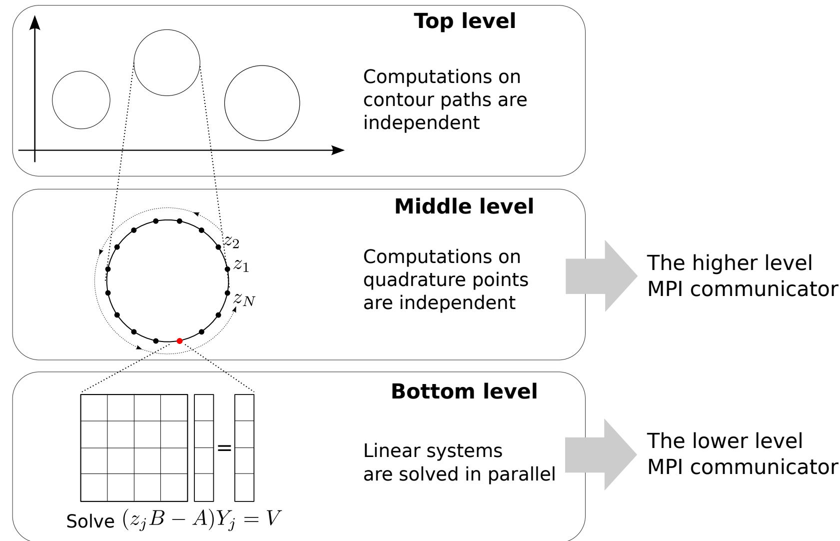

Figure 2.2: Three levels of parallelism and two-level MPI communicator

For the Rayleigh-Ritz procedure (described in the Section 3.4 (page 21)) and the residual calculations, matrix-vector multiplecations (mat-vec) of A and B for multiple vectors need to be done. The higher-level communicator manages the parallelization in performing mat-vec for different vectors. The lower-level communicator manages the parallelization for one mat-vec.

2.5 `zpares_prm` derived type

The derived type `zpares_prm` plays a central roll for the use of z-Pares. `zpares_prm` consists of components that represent several input and output parameters and inner variables. See Section 3.3 (page 20) for more details. In the rest of this users' guide, an entry of `zpares_prm` is refered to as `prm`.

2.6 `zpares` module

z-Pares subroutines can be accessed by using MODULE (feature from Fortran 90). To use z-Pares, the user needs to insert `use zpares` at the first line of a program unit as

```
subroutine user_sub
  use zpares
  implicit none
```

```
  ! user code
end subroutine
```

Note that, the user needs to add `$ZPARES_HOME/include/` to the include path when compring the user program.

If the user wants to use the sparse CSR MUMPS routines, the user needs to add `use zpares_mumps` in their code.

2.7 Initialization and finalization

Before calling any z-Pares main routines,

zpares_init

should be called with only one argement `zpares_prm` as like

```
call zpares_init(prm)
```

`zpares_init` initializes an entity of `zpares_prm`. This subroutine should be called before any modifications of the components of `zpares_prm`.

After the z-Pares main routine finishes, regardless of whether it succeed or not,

zpares_finalize

should be called with only one argement `zpares_prm` as like

```
call zpares_finalize(prm)
```

`zpares_finalize` deallocates the memory space for the internally managed variables.

2.8 Memory allocation

The arguments of z-Pares routine `eigval`, `X`, and `res` should be allocated before the user passes them to the routine. The user needs to allocate them with the return value of `zpares_get_ncv`. `zpares_get_ncv` can be called after setting the parameters in `prm`. Here is an example of the real valued problem in double precision.

```
subroutine user_sub
  use zpares
  implicit none

  type(zpares_prm) :: prm
  integer :: mat_size ! matrix size
  integer :: ncv
  double precision, allocatable :: eigval(:), X(:,:), res(:)
  ! Declare here

  call zpares_init(prm)
  ! The user can set parameters in prm here

  ncv = zpares_get_ncv(prm)
  ! The user should not change parameters in prm below
  allocate(eigval(ncv))
  allocate(X(mat_size, ncv))
```

```
allocate(res(ncv))
! Now the user can call a z-Pares subroutine
```

`zpares_get_ncv` returns the the value of `prm%Lmax * prm%M` in the current release. In the above code, `matrix_size` is the matrix size n. This size specifier for `X` is not the matrix size when the lower-level MPI parallelism is used. See Section 2.12.3 (page 13) for more details.

2.9 Setting a circle

The default shape of the contour is an ellipse. To indicate the position and the shape of the ellipse on the complex plane, the left and right edge are set in the input arguments `left` and `right`. The aspect ratio of ellipse can be set with `prm%asp_ratio`. The default value of `prm%asp_ratio` is 1.0 (precise circle). 2.3 illustrates the relations between the ellipse and the parameters.

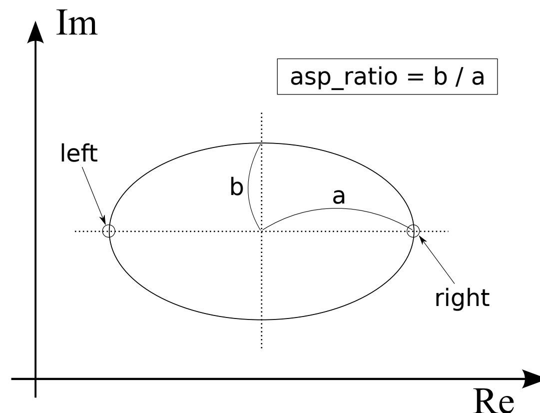

Figure 2.3: The ellipse and the parameters

The most frequent eigenvalue problems appear with real symmetric (or complex Hermitian) A, and real symmetric (or complex Hermitian) and positive definite B. In these cases the eigenvalues lies on the real axis. We provide special subroutines for these sub-class of eigenvalue problem. For these subroutines, `left` and `right` are replaced by real typed input argument `emin` and `emax` for simplicity. The ellipse encloses targeted eigenvalues is simply regarded as the search interval [`emin` , `emax`]. In this case, a vertically compressed ellipse provides better accuracy e.g. `prm%asp_ratio == 0.1`.

2.10 Relative residual norm

In z-Pares routines, relative residual norms are returned in `res`. The relative residual norms are defined as

$$\mathrm{res(i)} = \frac{\|A\boldsymbol{x}_i - \lambda_i B\boldsymbol{x}_i\|_2}{\|A\boldsymbol{x}_i\|_2 + |\lambda_i|\|B\boldsymbol{x}_i\|_2},$$

where λ and $\boldsymbol{x}_i$ are `eigval(i)` and `X(:,i)`, respectively.

If the user does not need the residual norms, specify `prm%calc_res = .false.`. If `prm%calc_res = .false.`, `res` will not be touched by the z-Pares routine and the matrix-vector multiplecations for computing the residuals will be avoided.

2.11 Reverse communication interface

z-Pares basically delegates tasks of

- Solving linear systems with multiple right hand sides $(z_j B - A)Y_j = BV$
- Performing matrix-vector multiplications of A and B

to the user, since efficient algorithm and matrix data structure are seriously problem dependent.

z-Pares delegates the tasks by using reverse communicaiton interface (RCI) rather than modern procedure pointer or external subroutine.

In the use of RCI, the user code communicate with the z-Pares subroutine in the following manner:

1. Reverse communication flag `prm%itask` is initilized with `zpares_init` before entering the loop of 2.
2. The z-Pares subroutine is repeatedly called until `prm%itask == ZPARES_TASK_FINISH`
3. In the loop of 2., the tasks indicated by `prm%itask` are completed with the user's implementation

By using RCI, the user does not have to define global, COMMON, or module variables to share the information (such as matrix data) with the subroutine given to the package, in contrust to manners using the procedure pointer or external subroutine. RCI is also used in eigensolver packages such as ARPACK and FEAST.

To briefly describe how a user code using RCI looks like, a skeleton code for solving complex non-Hermitian problem are given below.

```
do while ( prm%itask /= ZPARES_TASK_FINISH )
   call zpares_zrcigegv &
        (prm, nrow_local, z, mwork, cwork, left, right, num_ev, eigval, X, res, info)

   select case (prm%itask)
   case(ZPARES_TASK_FACTO)

      ! Here user factorizes (z*B - A)
      ! At the next return from zpares_zrcigegv,
      ! prm%itask==ZPARES_TASK_SOLVE with the same z is returned

   case(ZPARES_TASK_SOLVE)

      ! i = prm%ws; j = prm%ws+prm%nc-1
      ! Here user solves (z*B - A) X = cwork(:,i:j)
      ! The solution X should be stored in cwork(:,i:j)

   case(ZPARES_TASK_MULT_A)

      ! iw = prm%ws; jw = prm%ws+prm%nc-1
      ! ix = prm%xs; jx = prm%xs+prm%nc-1
      ! Here the user performs matrix-vector multiplications:
      ! mwork(:,iw:jw) = A*X(:,ix:jx)

   case(ZPARES_TASK_MULT_B)

      ! iw = prm%ws; jw = prm%ws+prm%nc-1
      ! ix = prm%xs; jx = prm%xs+prm%nc-1
      ! Here the user performs matrix-vector multiplications:
      ! mwork(:,iw:jw) = B*X(:,ix:jx)
```

```
    end select
end do
```

ZPARES_TASK_FINISH, ZPARES_TASK_FACTO, ZPARES_TASK_SOLVE, ZPARES_TASK_MULT_A and ZPARES_TASK_MULT_B are defined as module variables typed integer,parameter of the zpares module. Tasks delegated to the user are indicated with these values.

Implementing a linear solver is heavy task for users. To allow users to easily be get-started with z-Pares, we provide two interfaces for specific matrix data structure:

- Dense interface using LAPACK
- Sparse CSR interface using MUMPS

2.12 Reverse communicaton on two-level MPI parallelism

This section describes the manner of reverse communication on the two-level MPI parallelism by step-by-step instruction using a trivial toy problem.

Let A and B be,

$$A = \begin{pmatrix} 2 & & & & 0 \\ & 4 & & & \\ & & 6 & & \\ & & & \ddots & \\ 0 & & & & 1400 \end{pmatrix} \in \mathbb{R}^{700\times 700}$$

and

$$B = \begin{pmatrix} 2 & & & 0 \\ & 2 & & \\ & & \ddots & \\ 0 & & & 2 \end{pmatrix} \in \mathbb{R}^{700\times 700}.$$

Obviously, the eigenvalues of the generalized eigenvalue problem $A\boldsymbol{x} = \lambda B\boldsymbol{x}$ are $\{1, 2, \dots, 700\}$ and the eigenvectors are multiples of the unit vectors.

2.12.1 Sequential code

Now we get started with a sequential program. Assume that diagonal elements of A and B are stored in 1D array A and B. An integer variable mat_size is set to the matrix size n of A and B.

For a sequential code, X and working space should be allocated as

```
allocate(X(mat_size,ncv), mwork(mat_size,prm%Lmax), rwork(mat_size,prm%Lmax))
```

Reverse communication loop can be written as follows.

```
do while ( prm%itask /= ZPARES_TASK_FINISH )
   call zpares_zrcigegv &
        (prm, mat_size, z, mwork, cwork, left, right, num_ev, eigval, X, res, info)

   select case (prm%itask)
```

```
    case(ZPARES_TASK_FACTO)

       ! Do nothing

    case(ZPARES_TASK_SOLVE)

       ! i = prm%ws; j = prm%ws+prm%nc-1
       ! Here the user solves (z*B - A) X = cwork(:,i:j)
       ! The solution X should be stored in cwork(:,i:j)
       do k = 1, mat_size
          do jj = prm%ws, prm%ws+prm%nc-1
             cwork(k,jj) = cwork(k,jj) / (z*B(k) - A(k))
          end do
       end do

    case(ZPARES_TASK_MULT_A)

       ! iw = prm%ws; jw = prm%ws+prm%nc-1
       ! ix = prm%xs; jx = prm%xs+prm%nc-1
       ! Here the user performs matrix-vector multiplications:
       ! mwork(:,iw:jw) = A*X(:,ix:jx)
       do k = 1, mat_size
          do jj = 1, prm%nc
             jjw = prm%ws + jj - 1
             jjx = prm%xs + jj - 1
             mwork(k,jjw) = A(k)*X(k,jjx)
          end do
       end do

    case(ZPARES_TASK_MULT_B)

       ! iw = prm%ws; jw = prm%ws+prm%nc-1
       ! ix = prm%xs; jx = prm%xs+prm%nc-1
       ! Here the user performs matrix-vector multiplications:
       ! mwork(:,iw:jw) = B*X(:,ix:jx)
       do k = 1, mat_size
          do jj = 1, prm%nc
             jjw = prm%ws + jj - 1
             jjx = prm%xs + jj - 1
             mwork(k,jjw) = A(k)*X(k,jjx)
          end do
       end do

    end select
 end do
```

2.12.2 Parallel code with the higher-level parallelisim

For a parallel code with the higher-level parallelism, the reverse communication loop is the same as the sequential one. The user only has to set `prm%high_comm` to the global communicator (in most cases `MPI_COMM_WORLD`), and make sure that `prm%low_comm == MPI_COMM_SELF` (defalut value).

2.12.3 Parallel code with the lower-level parallelisim

Apart from the matrix data, the memory spaces for X, mwork and cwork are dominant (especially for X). Thus they should be problematic for large scale problems. To avoid this problem, z-Pares allows user to let the arrays X, mwork, cwork be row-wise distributed with the lower-level communicator. Now assume that we have 2 MPI processes. Let the number of the rows associated to each process be nrow_local. For example, nrow_local == 400 for rank 0 and nrow_local == 300 for rank 1. The sum of nrow_locals along all processors should be the matrix size, but the matrix size do not have to be divisible by nrow_local. Memory spaces for X and the working arrays should be allocated as

```
allocate(X(nrow_local,ncv), mwork(nrow_local,prm%Lmax), rwork(nrow_local,prm%Lmax))
```

Addtionaly, assume that the arrays X, mwork and cwork are row-wise distributed with the block distribution. The rank 0 manages the first 400 rows of the arrays and the rank 1 manages the remaining 300 rows of the arrays. For ease of coding, we also assumed that the rank 0 has the first 400 diagonal elements of A and B and the rank 1 has the remaining diagonals. In particular, $A_{i,i}(i = 1, 2, \dots, 400)$ stored in A(1:400) of rank-0 and $A_{i,i}(i = 401, 402, \dots, 700)$ stored in A(1:300) of rank-1. Likewise for B.

According to the above settings, the reverse communication loop can be written as follows.

```
if ( myrank == 0 ) then
   nrow_local = 400
else if ( myrank == 1 ) then
   nrow_local = 300
end if

! Here the user set the values of the distributed A and B
! and memory allocations for X, mwork and cwork

do while ( prm%itask /= ZPARES_TASK_FINISH )
   call zpares_zrcigegv &
      (prm, nrow_local, z, mwork, cwork, left, right, num_ev, eigval, X, res, info)

   select case (prm%itask)
   case(ZPARES_TASK_FACTO)

      ! Do nothing

   case(ZPARES_TASK_SOLVE)

      ! i = prm%ws; j = prm%ws+prm%nc-1
      ! Here the user solves (z*B - A) X = cwork(:,i:j)
      ! The solution X should be stored in cwork(:,i:j)
      do k = 1, nrow_local
         do jj = prm%ws, prm%ws+prm%nc-1
            cwork(k,jj) = cwork(k,jj) / (z*B(k) - A(k))
         end do
      end do

   case(ZPARES_TASK_MULT_A)

      ! iw = prm%ws; jw = prm%ws+prm%nc-1
      ! ix = prm%xs; jx = prm%xs+prm%nc-1
      ! Here the user performs matrix-vector multiplications:
      ! mwork(:,iw:jw) = A*X(:,ix:jx)
      do k = 1, nrow_local
         do jj = 1, prm%nc
```

```
            jjw = prm%ws + jj - 1
            jjx = prm%xs + jj - 1
            mwork(k,jjw) = A(k)*X(k,jjx)
         end do
      end do

   case(ZPARES_TASK_MULT_B)

      ! iw = prm%ws; jw = prm%ws+prm%nc-1
      ! ix = prm%xs; jx = prm%xs+prm%nc-1
      ! Here the user performs matrix-vector multiplications:
      ! mwork(:,iw:jw) = B*X(:,ix:jx)
      do k = 1, nrow_local
         do jj = 1, prm%nc
            jjw = prm%ws + jj - 1
            jjx = prm%xs + jj - 1
            mwork(k,jjw) = B(k)*X(k,jjx)
         end do
      end do

   end select
end do
```

In this case, we can see that `nrow_local` is passed to the second argument of the z-Pares routine instead of `mat_size`. Note that, the row distributions of `X`, `mwork` and `cwork` do not have to be the block distribution. It can be the cyclic, the block cyclic or a much more complicated distribution, provided that the manners of distributions of `X`, `mwork` and `cwork` agree. To use the lower-level parallelism without the higher level parallelism, the user has to set `prm%low_comm` to the global communicator (in most cases `MPI_COMM_WORLD`), and make sure that `prm%high_comm == MPI_COMM_SELF` (defalut value).

Of cource, for general matrices, computations for solving linear system and matrix-vector multiplecation require communications. The user has to manage the communications and the matrix data distribution with the lower-level communicator.

2.12.4 Parallel code with the two-level parallelisim

Now we consider to use both the high-level and lower-level parallelism. For a code with the two-level parallelism, the reverse communication loop is the same as that of the lower-level one.

The user needs to create two MPI-communicators that manage each level of parallelism by splitting the global communicator (in most cases `MPI_COMM_WORLD`). An example on the splitting is shown as follows.

```
n_procs_low = 8
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, n_procs, ierr)
high_color = mod(myrank, n_procs_low)
high_key = myrank
low_color = myrank / n_procs_low
low_key = myrank
call MPI_COMM_SPLIT(MPI_COMM_WORLD, high_color, high_key, prm%high_comm, ierr)
call MPI_COMM_SPLIT(MPI_COMM_WORLD, low_color, low_key, prm%low_comm, ierr)
```

Note that this code only works if `n_procs` is divisible by `n_procs_low`. Here, let `n_procs_high` and `n_procs_low` be

```
call MPI_COMM_SIZE(prm%high_comm, n_procs_high, ierr)
call MPI_COMM_SIZE(prm%low_comm, n_procs_low, ierr)
```

When the two-level parallelism is used, following conditions must be satisfied:

1. `n_procs_high*n_procs_low == n_procs` at the all MPI processes
2. the manners of row-distributions of `X`, `mwork` and `cwork` agree in the lower-level communicator
3. the manners in 2. of the all processes of the higher-level communicator should also agree.

2.13 Efficient implementations for specific problems

In the above descriptions, we have used the subroutines for complex non-Hermitian problems. In z-Pares, the effcienct implematations are given for exploiting specific features of the problem. Following features are regarded:

- Symmetry or hermiticity of matrix A and B
- Positive definiteness of B
- $B = I$ (Standard eigenvalue problem)

We recommend the user to let z-Pares regard these features by setting appropriate parameters to obtain maximum efficiency.

Chapter 3

General use of z-Pares

3.1 Naming convention of z-Pares routines

In the use of z-Pares, the user only uses one z-Pares main subroutine other than `zpares_init` and `zpares_finalize`. z-Pares main subroutines are named in the form of

zpares_TXXXYYZZ

T={s | c | d | z} (this means that **T** is replaced by s, c, d, or z) specifies the data type (real or complex) of the matrix data and the precision. **XXX**={rci | dns | mps} specifies the interface which is implemented by the routine. **YY**={ge | sy | he} specifies the symmetry and/or positive definiteness of the input matrices. **ZZ**={gv | ev} specifies the kind of eigenvalue problem which is solved (standard or generalized).

The correponding precisions and the matrix data types of **T**={s | c | d | z} are shown in Table 3.1. Symbols _REAL_, _COMPLEX__, and _MAT_TYPE_ represent the data types and are used in the following explanation about the data types of the variables. The user should replace them to the actual types listed in Table 3.1.

Table 3.1: Precisions and data types. double precision may be replaced by real(8) or real*8 and complex(kind(0d0)) may be replaced by complex(8), complex*16, or double complex

T	precision	A, B	_REAL_ type	_COMPLEX_ type	_MAT_TYPE_ type
s	single	real	real	complex	real
c	single	complex	real	complex	complex
d	double	real	double precision	complex(kind(0d0))	double precision
z	double	complex	double precision	complex(kind(0d0))	complex(kind(0d0))

The meanings of **XXX**={rci | dns | mps} are described in Table 3.2.

Table 3.2: Interfaces of z-Pares.

XXX	Description
rci	Reverse communication interface
dns	Dense interface using LAPACK
mps	Sparse CSR interface using MUMPS

The conditions that **YY**={ge | sy | he} assumes are shown in Table 3.3 .

Note here that, as LAPACK users expect, **YY**=sy is valid with **T**=s or **T**=d, and **YY**=he is valid with **T**=c or **T**=z.

The explanation of **ZZ**={gv | ev} is shown in Table 3.4.

Table 3.3: Matrix symmetry and positive definiteness.

YY	Property of matrix A	Property of matrix B
sy	Real symmetric	Real symmetric and positive definite
he	Complex Hermitian	Complex Hermitian and positive definite
ge	Other than above	

Table 3.4: Problem types.

ZZ	Description
ev	Standard eigenvalue problem : $Ax = \lambda x$
gv	Generalized eigenvalue problem : $Ax = \lambda Bx$

3.2 Input and output arguments in z-Pares routines

The form of the sequence of arguments of the z-Pares routines differ depending on **XXX**, **YY**, and **ZZ**.

If **YY** is `ge`, the sequence of input arguments is in the form of

zpares_TXXXgeZZ(prm, nrow_local, {XXX dependent sequence}, left, right, num_ev, eigval, X, res, info)

The descriptions of the arguments are shown in Table 3.5. {**XXX** dependent sequence} is an argument sequence which depends on **XXX**. The actual sequences are described in the subsections of this section.

Table 3.5: Common arguments for zpares_TXXXgeZZ.

Name	Description	Type	Input or output
prm	Derived type zpares_prm	zpares_prm	IN/OUT
nrow_local	Number of rows of X and/or the working spaces. See Section 2.12.3 (page 13)	integer	IN
left	Position of left edge of the search circle. See Section 2.9 (page 9).	_COMPLEX_	IN
right	Real part of position of right edge of the search circle. See Section 2.9 (page 9).	_REAL_	IN
num_ev	Number of calculated eigenvalues	integer	(IN)/OUT
eigval	Array of eigenvalues	_COMPLEX_, dimension(*)	OUT
X	Array of eigenvectors	_MAT_TYPE_, dimension(nrow_local,*)	(IN)/OUT
res	Array of relative residual norm	_REAL_, dimension(*)	OUT
info	Diagnostic information	integer	OUT

If **YY** is `sy` or `he`, the sequence of input arguments is in the form of

zpares_TXXX{sy|he}ZZ(prm, nrow_local, {XXX dependent sequence}, emin, emax, num_ev, eigval, X, res, info)

The descriptions of the arguments are shown in Table 3.6.

Note that the argument `num_ev` is basically OUTPUT. It is the number of approximate eigenvalues that have been computed, but is NOT the number of eigenvalues that are required by the user. The number of eigenvalues that are required by the user should be taken care by the setting of `prm%L`, `prm%Lmax` and `prm%M`. `prm%L` should be greater than or equal to the maximum multiplicity of eigenvalue inside the contour, and `prm%L * prm%M` should be greater

Table 3.6: Common arguments for zpares_TXXXsyZZ and zpares_TXXXheZZ.

Name	Description	Type	Input or output
prm	Derived type zpares_prm	zpares_prm	IN/OUT
nrow_local	Number of rows of X and/or the working spaces. See Section 2.12.3 (page 13)	integer	IN
emin	Minimum value of the search interval. See Section 2.9 (page 9).	_REAL_	IN
emax	Maximum value of the search interval. See Section 2.9 (page 9).	_REAL_	IN
num_ev	Number of calculated eigenvalues	integer	(IN)/OUT
eigval	Array of eigenvalues	_REAL_, dimension(*)	OUT
X	Array of eigenvectors	_MAT_TYPE_, dimension(nrow_local,*)	(IN)/OUT
res	Array of relative residual norms	_REAL_, dimension(*)	OUT
info	Diagnostic information	integer	OUT

than or equal to the number of eigenvalues inside the contour including the multiplicity. `prm%L <= prm%Lmax` should be maintained.

`num_ev` and `X` also become input arguments when the user already has approximate eigenvectors and gives them to the z-Pares routine (`prm%user_source == .true.`). In this case:

- `num_ev` should be the number of approximate eigenpairs
- `X` should contain approximate eigenvectors in leading `num_ev` columns

Note that `num_ev >= prm%L` should be maintained and in this case the estimation of the eigenvalue count in `prm%num_ev_est` becomes meaningless.

Real non-symmetric case

Only in case of **T**= {`s` | `d`} and **YY**=`ge` (real non-symmetric matrices), eigenvalues and eigenvectors will be returned with conjugate pairs if they are complex value. The computed eigenvalues are returned in complex typed 1D array `eigval`. Complex conjugate pairs of eigenvalues appear consecutively and the eigenvalue with the positive imaginary part appears first. The eigenvectors are returned in real typed 2D array `X`. `X` is returned, and the computed eigenvectors $\boldsymbol{x}_j$ of `eigval(j)` are represented in the following manner:

- If `eigval(j)` has zero imaginary part, then $\boldsymbol{x}_j$ = `V(:,j)`
- If `eigval(j)` and `eigval(j+1)` form a complex conjugate pair, then $\boldsymbol{x}_j$ = `V(:,j) + i*V(:,j+1)` and $\boldsymbol{x}_{j+1}$ = `V(:,j) - i*V(:,j+1)`, where `i` is the imaginary unit.

3.2.1 Arguments in reverse communication interface

If **XXX** is `rci`, the routine implements a reverse communication interface (RCI), For RCI routines, {**XXX** dependent sequence} is {z, mwork, cwork}. Table 3.7 describes the arguments.

Table 3.7: {XXX dependent sequence} for zpares_TrciYYZZ.

Name	Description	Type	Input or output
z	Quadrature point	_COMPLEX_	OUT
mwork	Working space for performing matrix-vector multiplecation	_MAT_TYPE_	-
cwork	Working space for solving linear system	_COMPLEX_	-

3.2.2 Arguments in dense interface

If **XXX** is `dns`, the routine implements a dense interface. The dense routines assume the matrices are given in a dense format (2D array) and solves the inner linear systems by LAPACK routines. For the dense routines, parallel implementations of the higher-level parallism is only supported. Thus `prm%low_comm` is always `MPI_COMM_SELF`.

If **YY**=`sy` or **YY**=`he`, and **ZZ**=`gv`, {**XXX** dependent sequence} is {UPLO, A, LDA, B, LDB}. Table 3.8 describes the arguments.

Table 3.8: {XXX dependent sequence} for zpares_Tdnssygv and zpares_Tdnshegv.

Name	Description	Type	Input or output
`UPLO`	`UPLO={'U'|'L'}`: {Upper | Lower} triangle parts of matrices are stored in `A` and `B`	`character`	IN
`A`	Array represents matrix A	`_MAT_TYPE_, dimension(LDA,*)`	IN
`LDA`	Leading dimension of `A`	`integer`	IN
`B`	Array represents matrix B	`_MAT_TYPE_, dimension(LDB,*)`	IN
`LDB`	Leading dimension of `B`	`integer`	IN

If **YY**=`ge`, {**XXX** dependent sequence} is {A, LDA, B, LDB}. Table 3.9 describes the arguments.

Table 3.9: {XXX dependent sequence} for zpares_Tdnsgegv.

Name	Description	Type	Input or output
`A`	Array represents matrix A	`_MAT_TYPE_, dimension(LDA,*)`	IN
`LDA`	Leading dimension of `A`	`integer`	IN
`B`	Array represents matrix B	`_MAT_TYPE_, dimension(LDB,*)`	IN
`LDB`	Leading dimension of `B`	`integer`	IN

For the standard eigenproblem routines (i.e. ZZ=`ev`), `B` and `LDB` should be omitted.

In the dense interfaces, argument `nrow_local` should be the matrix size n, since the lower-level parallelism is not supported.

3.2.3 Arguments in sparse CSR MUMPS interface

If **XXX** is `mps`, the routine implements a sparse CSR MUMPS interface. The sparse CSR MUMPS routines assumes matrices are given in Compressed Sparse Row (CSR) format and solves inner linear systems by MUMPS. For the sparse CSR MUMPS routines, parallel implementations of the higher-level parallism is only supported. Thus `prm%low_comm` is always `MPI_COMM_SELF`.

For sparse CSR MUMPS routines, if **ZZ**=`gv`, {**XXX** dependent sequence} is {rowptr_A, colind_A, val_A, rowptr_B, colind_B, val_B}. Table 3.10 describes the arguments.

For **YY**=`sy` and **YY**=`he`, only the entries of either upper or lower triangular parts of the matrices should be stored.

For the standard eigenproblem routines (i.e. **YY**=`ev`), `rowptr_B`, `colind_B` and `val_B` should be omitted.

In the sparse CSR routines, argument `nrow_local` should be the matrix size n, since the lower-level parallelism is not supported.

Table 3.10: {XXX specific sequence} for zpares_TmpsYYgv.

Name	Description	Type	Input or output
rowptr_A	Array of row pointers of A	integer, dimension(*)	IN
colind_A	Array of column indices of A	integer, dimension(*)	IN
val_A	Array of value of A	_MAT_TYPE_, dimension(*)	IN
rowptr_B	Array of row pointers of B	integer, dimension(*)	IN
colind_B	Array of column indices of B	integer, dimension(*)	IN
val_B	Array of value of B	_MAT_TYPE_, dimension(*)	IN

3.3 Parameters in zpares_prm

The derived type zpares_prm plays a central role in the use of z-Pares. It is passed to the first argument in a z-Pares main routine. Components of zpares_prm indicate input parameters for fine-tuning of the eigensolver or detailed information about the result. In the following subsections, we desribe the components of zpares_prm.

3.3.1 Integer input parameters

Table 3.11 shows integer input parameters.

Table 3.11: Integer input parameters in the components of zpares_prm.

Component	Description	Condition	Default value
N	Number of quadrature points	N > 0 and even value	32
L	Number of source vectors	L > 0	16
Lmax	Maxmum value of L	Lmax ≤ L	64
M	Maximum moment degree	0 < M ≤ N	16
imax	Maximum refinement iteration	imax ≥ 0	0
n_orth	Number of iteration for orthonormalization step	n_orth ≥ 0	3
extract	Extract method. See 3.4 (page 21).	ZPARES_EXTRACT_RR or ZPARES_EXTRACT_EM	ZPARES_EXTRACT_RR
Lstep	Step size for increasing L	Lstep > 0	8
high_comm	Higher-level communicator	-	MPI_COMM_SELF
low_comm	Lower-level communicator	-	MPI_COMM_SELF
write_unit	File unit for verbosing	A valid number	6 (Standard output)
verbose	Verbose level	{0|1|2}	0..
quad_type	See Section 4.1 (page 24) for description	ZPARES_QUAD_ELL_TRAP or ZPARES_QUAD_USER	ZPARES_QUAD_ELL_TRAP

The symbols in Table 3.11 that start with ZPARES_ are module variables typed integer,parameter of the zpares module.

3.3.2 Double precision real input parameters

Table 3.12 shows double precision input real parameters. These double precision parameters are used even if the user use a single precsion routine.

Table 3.12: Double precision real input parameters in the components of zpares_prm.

Component	Description	Condition	Default value
delta	Threshold for determining numerical rank	$0 \le$ delta ≤ 1	1d-12
spu_thres	Threshold for trancating eigenvalue with respect to spurious indicator	$0 \le$ spu_thres ≤ 1	1d-4
asp_ratio	Aspect ratio of ellipse	asp_ratio > 0	1.0

3.3.3 Logical input parameters

Table 3.13 shows logical input parameters.

Table 3.13: Logical input parameters in the componets of zpares_prm.

Component	If .true.	Default value
calc_res	Calculate residual norm	.true.
trim_out	Trim eigenvalues that are outside of the circle	.true.
trim_spu	Trim eigenvalues whose corresponding spurious indicators are smaller than spu_thres	.false.
trim_res	Trim eigenvalues whose corresponding residual norms are larger than tol	.false.
Hermitian	Assume both A and B are Hermitian	.false.
B_pos_def	Assume B is positive definite	.false.
sym_contour	See Section 4.1 (page 24) for description	.false.

Note that Hermitian and B_pos_def treated as .true., if **YY** = {sy | he}.

3.3.4 Output parameters

Table 3.14 and Table 3.15 show integer output parameters and double precsion output parameters. For the single precision routines, the single precsion results are upcasted to double precision.

Table 3.14: Integer output parameters in the components of zpares_prm.

Component	Description
num_ev_est	Estimation of the number of eigenvalues in the interval or the circle
iter	Number of refiment iterations
num_basis	Number of basis vectors used for projection.
L	Auto-tuned value of the source size

3.4 Extracting method for eigenpairs

z-Pares extracts eigenpairs from the approximate eigensubspace by two different ways:

- Rayleigh-Ritz procedure
- Hankel method

In the Rayleigh-Ritz procedure, linear combinations of the solutions of the linear systems used as the basis for the procedure. The Hankel method computes eigenpairs by solving a reduced generalized eigenvalue problem of (block) Hankel matrices. In our experience, the Rayleigh-Ritz procedure gives better accuracy than the Hankel method. But the Hankel method is still attractive because no matrix-vector multiplecation of A and B is needed

Table 3.15: Double precision, pointer typed output parameters in the components of zpares_prm.

Component	Description
`indi_spu(:)`	The i-th component is the indicator of spuriousness corresponds the i-th eigenvalue. The length is `num_ev`. See Section 3.7 (page 23).
`sigval(:)`	Singular values used for a low-rank approximation. The i-th component is the i-th largest singular value. The length is `prm%num_basis`

if the residual is not needed (`prm%calc_res == .false.`). The Rayleigh-Ritz procedure is enabled if `prm%extract == ZPARES_EXTRACT_RR` (default) and the Hankel method is enabled if `prm%extract == ZPARES_EXTRACT_EM`.

3.5 Reverse communicaton interface in complex Hermitian case

Unless **YY**=`he`, the reverse communication loop is written in the form described in Section 2.11 (page 10). If YY=`he`, the reverse communication loop should be written as following:

```
do while ( prm%itask /= ZPARES_TASK_FINISH )
   call zpares_zrcihegv &
        (prm, nrow_local, z, mwork, cwork, emin, emax, num_ev, eigval, X, res, info)

   select case (prm%itask)
   case(ZPARES_TASK_FACTO)

      ! Here the user factorizes (z*B - A)
      ! At the next return from zpares_zrcihegv,
      ! prm%itask==ZPARES_TASK_SOLVE with the same z

   case(ZPARES_TASK_SOLVE)

      ! i = prm%ws; j = prm%ws+prm%nc-1
      ! Here the user solves (z*B - A) X = cwork(:,i:j)
      ! The solution X should be stored in cwork(:,i:j)
      ! At the next return from zpares_zrcihegv,
      ! prm%itask==ZPARES_TASK_FACTO_H with the same z is returned

   case(ZPARES_TASK_FACTO_H)

      ! Here the user can factorize (z*B - A)^{H}
      ! In most cases, this task is unnecessary
      ! At the next return from zpares_zrcihegv,
      ! prm%itask==ZPARES_TASK_SOLVE_H with the same z is returned

   case(ZPARES_TASK_SOLVE_H)

      ! i = prm%ws; j = prm%ws+prm%nc-1
      ! Here the user solves (z*B - A) X = cwork(:,i:j)
      ! The solution X should be stored in cwork(:,i:j)

   case(ZPARES_TASK_MULT_A)

      ! iw = prm%ws; jw = prm%ws+prm%nc-1
      ! ix = prm%xs; jx = prm%xs+prm%nc-1
      ! Here the user performs matrix-vector multiplications:
      ! mwork(:,iw:jw) = A*X(:,ix:jx)
```

```
    case(ZPARES_TASK_MULT_B)

       ! iw = prm%ws; jw = prm%ws+prm%nc-1
       ! ix = prm%xs; jx = prm%xs+prm%nc-1
       ! Here the user performs matrix-vector multiplications:
       ! mwork(:,iw:jw) = B*X(:,ix:jx)

    end select
 end do
```

The symbols start with `ZPARES_TASK` are module variables typed `integer,parameter` of the `zpares` module.

3.6 Estimations of eigenvalue count

Along with eigenpairs z-Pares computes a stochastic estimation of the eigenvalue count. This value is used for auto-tuning for the parameter `prm%L`. The estimated eigenvalue count is returned in `prm%num_ev_est`. Note that the accuracy of the estimation is problem dependent. The error may become arbitrary large.

3.7 Indicator for spurious eigenvalues

z-Pares provides an indicator of the spuriousness of an eigenvalue. The indicators are returned in `prm%indi_spu(:)`. `prm%indi_spu(i)` indicates spuriousness of `eigval(i)`. A value of the indicator takes $[0, 1]$. The smaller indicator, the more spurious the eigenvalue is. The spurious eigenvalues can be discarded using threshold value `prm%spu_thres`.

3.8 Error diagnostics in `info`

The output parameter `info` of a z-Pares main routine tells diagnostics information on exit. It takes values listed in Table 3.16.

Table 3.16: Diagnostics information

Value	Description
ZPARES_INFO_SUCCESS	Successful exit
ZPARES_INFO_INVALID_PARAM	Invalid parameter
ZPARES_INFO_LAPACK_ERROR	LAPACK error
ZPARES_INFO_MPI_ERROR	MPI error

Chapter 4

Advanced features

4.1 User defined quadrature rule

The user can set an arbitrary quadrature rule by oneself. The subroutine which returns quadrature rule can be passed as optional argument at the end of argument list (next to `info`) of any z-Pares main routines.

The interface of the user defined subroutine should be in the form of

subroutine sub(mode, qmax, quad_idx, left, right, z, weight, zeta, lambda, info)

Table 4.1 describes the arguments. To enable the user defined quadrature rule, the user should set `prm%quad_type = ZPARES_QUAD_USER`.

Table 4.1: Interface definition of a subroutine represents an user defined quadrature rule.

Name	Description	Type	Input or output
`mode`	Mode specifier	`integer`	IN
`qmax`	Maximum value of `quad_idx`	`integer`	IN
`quad_idx`	Value of j	`integer`	IN
`left`	`left` which has been passed to the z-Pares main routine	`complex(kind(0d0))`	IN
`right`	`right` which has been passed to the z-Pares main routine	`double precsion`	IN
`z`	Quadrature point z_j	`complex(kind(0d0))`	OUT
`weight`	Quadrature weight w_j	`complex(kind(0d0))`	OUT
`zeta`	$\zeta_j := f(z_j)$	`complex(kind(0d0))`	OUT
`lambda`	μ as input, $\lambda := f^{-1}(\mu)$ as output	`complex(kind(0d0))`	IN/OUT

In z-Pares, approximate the contour integral by a N point numerical quadrature:

$$\frac{1}{2\pi\mathrm{i}}\oint_{\Gamma} f(z)^k (zB - A)^{-1} BV \mathrm{d}z \approx \sum_{j=1}^{N} w_j f(z_j)^k (z_j B - A)^{-1} BV \quad (k = 0, 1, \dots),$$

where Γ is a counter-clockwise contour path, $f(z)$ is a holomorphic function which maps `z` so that z is $|f(z)| \approx 1$, and z_j and w_j are quadrature points and weights ($j = 1, 2, \dots, N$).

The subroutine should behave differently according to input argument `mode`:

- If `mode == 0`

In this mode, the user can set the quadrature points and weights. j is passed in `quad_idx` by the z-Pares main routine. The corresponding z_j, w_j, and $\zeta_j := f(z_j)$ should be set in `z`, `weight`, and `zeta`. The output argument `lambda` does not have to be touched.

- If `mode == 1`

  This mode is significant for `prm%extract == ZPARES_EXTRACT_EM`. In this mode, the user should compute $\lambda := f^{-1}(\mu)$. μ comes in `lambda` as an input then λ should be passed in `lambda` as an output. The output arguments `z`, `weight`, and `zeta` do not have to be touched. If `prm%extract == ZPARES_EXTRACT_RR` it can be ignored.

If the components of each set of $\{z_j\}_{j=1}^{N}$, $\{w_j\}_{j=1}^{N}$, and $\{\zeta_j\}_{j=1}^{N}$ form complex conjugate pairs, we recommend the user to set `prm%sym_contour=.true.` for a possible efficiency. If `prm%sym_contour==.true.`, for each set $\{z_j\}_{j=1}^{N/2}$, $\{w_j\}_{j=1}^{N/2}$, and $\{\zeta_j\}_{j=1}^{N/2}$ should be the counterparts of the complex conjugate pairs.