nu–TRLan User Guide version 1.0

A high-performance software package for large-scale Harmitian eigenvalue problems[1]

October 2008

Ichitaro Yamazaki[2]
Kesheng Wu[3]
Horst Simon[3]

---

[2] University of California, Davis, One Shields Avenue, Davis, California 95616.
Email: ic.yamazaki@gmail.com

[3] Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, California 94720.
Email: {hdsimon,kwu}@lbl.gov

Table of Contents

1 Overview 1

2 Installation 2

3 Example program 3

4 trl_info structure 8
- 4.1 Initialization, `trl_init_info` 8
- 4.2 Starting vector options, `trl_set_iguess` 9
- 4.3 Checkpoint, `trl_set_checkpoint` 10
- 4.4 Performance statistics, `trl_set_debug` 11
- 4.5 Execution summary, `trl_print_info` and `trl_terse_info` . . . 11
- 4.6 Member variables 13

5 `trlan` subroutine 15
- 5.1 Interface 15
- 5.2 Matrix-vector multiply 15
- 5.3 Workspace 16
- 5.4 Error handling 17
- 5.5 Fortran interface 20

6 Solver parameters 23
- 6.1 Restart scheme 23
- 6.2 Maximum projection dimension 23
- 6.3 Solution accuracy 23

7 Acknowledgements and contact information 24

8 References 25

Index 27

1 Overview

The original software package TRLan, [TRLan User Guide], page 26, implements the thick-restart Lanczos method, [Wu and Simon 2001], page 26, for computing eigenvalues λ and their corresponding eigenvectors v of a symmetric matrix A:

$$Av = \lambda v.$$

Its effectiveness in computing the exterior eigenvalues of a large matrix has been demonstrated, [LBNL-42982], page 26. However, its performance strongly depends on the user-specified dimension of a projection subspace. If the dimension is too small, TRLan suffers from slow convergence. If it is too large, the computational and memory costs become expensive. Users must select an appropriate subspace dimension for each eigenvalue problem at hand in order to balance the solution convergence and costs. To free users from this difficult task, nu–TRLan, [LNBL-1059E], page 25, adjusts the subspace dimension at every restart such that optimal performance in solving the eigenvalue problem is automatically obtained. This document provides a user guide to the nu–TRLan software package.

The original TRLan software package was implemented in Fortran 90 to solve symmetric eigenvalue problems using static projection subspace dimensions. nu–TRLan was developed in C and extended to solve Hermitian eigenvalue problems. It can be invoked using either a static or an adaptive subspace dimension. In order to simplify its use for TRLan users, nu–TRLan has interfaces and features similar to those of TRLan:

- Solver parameters are stored in a single data structure called `trl_info`, Chapter 4 [`trl_info` structure], page 8.
- Most of the numerical computations are performed by `BLAS`, [BLAS], page 25, and `LAPACK`, [LAPACK], page 25, subroutines, which allow nu–TRLan to achieve optimized performance across a wide range of platforms.
- To solve eigenvalue problems on distributed memory systems, the message passing interface (MPI), [MPI forum], page 25, is used.

The rest of this document is organized as follows. In Chapter 2 [Installation], page 2, we provide an installation guide of the nu–TRLan software package. In Chapter 3 [Example], page 3, we present a simple nu–TRLan example program. In Chapter 4 [`trl_info` structure], page 8, and Chapter 5 [`trlan` subroutine], page 15, we describe the solver parameters and interfaces in detail. In Chapter 6 [Solver parameters], page 23, we discuss the selection of the user-specified parameters. In Chapter 7 [Contact information], page 24, we give the acknowledgements and contact information of the authors. In Chapter 8 [References], page 25, we list reference to related works.

2 Installation

All the source codes of the nu–TRLan software package are compressed into one file named `nutrlan.tar.gz`, which can be downloaded at

https://codeforge.lbl.gov/projects/trlan/.

After the source code is downloaded, it must be unpacked by invoking the following command:

```
% tar -xzf nutrlan.tar.gz
```

If your `tar` program does not support the flag `-z`, then the following commands can be used:

```
% gunzip -d nutrlan.tar.gz
% tar -xf nutrlan.tar
```

This will unpack the source code under the nu–TRLan top level directory '`nutrlan`'.

To install the package, you need a `C` compiler and the `BLAS` and `LAPACK` libraries. If the `BLAS/LAPACK` libraries that are optimized for your machine are not available, the required subroutines (not optimized for your machine) are included in the package under the sub-directory '`CBLAS`'. On a distributed memory machine, MPI, [MPI forum], page 25, is also required. The compiler and locations of these libraries on your machine must be specified in the file named '`Make.inc`', which can be found under the top directory '`nutrlan`'.

After '`Make.inc`' is modified to reflect the environments on your machine, nu–TRLan can be compiled into either the sequential or the parallel version of the library `libtrlan.a` using their respective commands from the top directory '`nutrlan`':

```
% make lib
```

or

```
% make plib
```

A number of example programs that use nu–TRLan are provided in the sub-directory '`examples`'. If `file_name` is the file name of the example that you want to test, then it can be compiled by:

```
% make file_name
```

For example, to compile the example program '`psimple`', [simple example], page 3, invoke the following command:

```
% make psimple
```

This will generate an executable called '`psimple`', which can then be run as:

```
% mpirun -np 2 ./psimple
```

For more information on the example programs, see the '`README`' file under the top directory '`nutrlan`', or see Chapter 3 [Example], page 3.

For further questions or comments or if you encounter errors in the installation procedure, please feel free to contact the authors by emailing to `ic.yamazaki@gmail.com` (Ichitaro Yamazaki), `kwu@lbl.gov` (Kesheng Wu), or `hdsimon@lbl.gov` (Horst Simon).

3 Example program

In this chapter, we provide a simple example that uses the nu–TRLan software package. The example computes the smallest 10 eigenvalues and corresponding eigenvectors of a diagonal matrix diag$(1, 2, 3, \ldots)$.

A user of nu–TRLan is required to provide a subroutine that computes the matrix-vector multiply with the coefficient matrix of the eigenvalue problem. This subroutine must have the same interface as the following subroutine that computes the matrix-vector multiply with the diagonal matrix diag$(1, 2, 3, \ldots)$:

```
void diag_op(const int nrow, const int ncol, const double *xin,
             const int ldx, double *yout, const int ldy, void *mvparam)
{
  int i, j, ioff, joff, doff, nrow, ncol, ldx, ldy;

  MPI_Comm_rank(MPI_COMM_WORLD, &i);
  doff = nrow*i;
  for( j=0; j<ncol; j++ )
  {
     ioff = j*ldx;
     joff = j*ldy;
     for( i=0; i<nrow; i++ )
        yout[joff+i] = (doff+i+1)*xin[ioff+i];
  }
}
```

This subroutine applies the matrix multiply to the vectors stored in `xin` and returns the results in `yout`. Beside these two vectors in the arguments, `nrow` specifies the numbers of rows, and `ncol` is the number of columns, of the vectors stored in `xin`. Furthermore, `ldx` and `ldy` are the leading dimensions of `xin` and `yout`, respectively. The last argument `mvparam` can be used to pass additional parameters to the subroutine, see [mvparam], page 14 for more details. Note that on a distributed memory system, `xin` and `yout` store the "local" vectors. Specifically, if the local problem size `pnrow` is `pnrow1`, `pnrow2`, and `pnrow3` on processes 1, 2, and 3, respectively, then `xin` and `yout` store the elements of the global vectors from the locations 0 to `pnrow1-1`, `pnrow1` to `pnrow1+pnrow2-1`, and `pnrow1+pnrow2` to `pnrow1+pnrow2+pnrow3-1`, respectively. In this example, the local problem size of every process is the same, i.e., `pnrow1=pnrow2=pnrow3=pnrow`, and process `i` has the elements from `i`×`nrow` to (`i+1`)×`nrow`. More information on the matrix-vector multiply can be found in Section 5.2 [matrix operation], page 15.

Using this matrix-vector multiply subroutine, the following program solves the example eigenvalue problem:

```
int main()
{
  static const int nrow=1000, lohi=-1, ned=10, maxlan=100, mev=10;
  static const double tol=1.4901/100000000;
  int i, check, lwrk=maxlan*(maxlan+10);
  double eval[mev], evec[mev*nrow], exact[mev];
  double res[lwrk], wrk[lwrk];
```

```
  trl_info info;
  if( MPI_Init(0,NULL) != MPI_SUCCESS ) {
     printf( "Failed to initialize MPI.\r\n" );
     return 0;
  }
  trl_init_info(&info, nrow, maxlan, lohi, ned, tol, 1, 2000, -1);
  for( i=0; i<mev;  i++ ) eval[i] = 0.0;
  for( i=0; i<nrow; i++ ) evec[i] = 1.0;
  trlan(diag_op, &info, nrow, mev, eval, evec, nrow, lwrk, res);
  trl_print_info(&info, 2*nrow);
  for( i=0; i<mev; i++ ) exact[i] = i+1;
  if( info.nec > 0 )
     i = info.nec;
  else
     i = mev - 1;
  trl_check_ritz(diag_op, &info, nrow, i, evec, nrow, eval,
                 &check, res, exact, i, wrk);
  MPI_Finalize();
}
```

The above example program first calls the subroutine `trl_init_info` to initialize the structure `info` of type `trl_info`. The interface to the subroutine is:

```
void trl_init_info(
       trl_info *info,   // pointer to the structure.
       int nrow,         // local problem size.
       int maxlan,       // max. number of basis vectors.
       int lohi,         // -1, compute smallest eigenvalues.
       int ned,          // number of desired eigenvalues.
       double tol,       // required solution accuracy.
       int restart,      // restart scheme.
       int mxmv,         // max. number of matrix operations.
       int mpicom        // -1, MPI_COMM_WORLD is duplicated.
       )
```

This subroutine `trl_init_info` must be called before any other nu–TRLan subroutines. In this example, the local problem size of every process is set to be the same, i.e., `nrow=1000`. Hence, the global problem size increases as more processes are used to run this example, i.e., the global problem size is `nprocs`×`nrow`, where `nprocs` is the number of processes. For more information on the structure `trl_info` and the subroutine `trl_init_info`, see Section 4.6 [`trl_info` structure], page 13, and Section 4.1 [`trl_init_info` subroutine], page 8, respectively.

The example program then invokes the main computational subroutine `trlan` that computes the eigenvalues and eigenvectors. The interface to this subroutine is as follows:

```
void trlan(
       void (*op)(const int,const int,double*,const int,
                  double*,const int,void*),
                          // matrix-vector multiply subroutine.
       trl_info *info,    // structure storing parameters.
```

```
        int nrow,           // local dimension of the problem.
        int mev,            // size of eval.
        double *eval,       // storage of eigenvalues.
        double *evec,       // storage of eigenvectors.
        int lde,            // leading dimension of evec.
        int lwrk,           // size of wrk.
        double *wrk         // workspace.
        )
```

In Chapter 5 [trlan interface], page 15, the interface to the subroutine `trlan` is described in detail.

After the completion of the subroutine `trlan`, an execution summary of `trlan` is printed by calling the subroutine `trl_print_info`. The interface to the subroutine is:

```
void trl_print_info(
        trl_info * info,   // structure storing parameters.
        int mvflop         // flops per matrix operation.
        )
```

The argument `info` is the pointer to the data structure storing the information on the current eigenvalue problem, and `mvflop` is the required number of floating-point operations (flops) per matrix-vector multiply. In this example, the matrix-vector multiply subroutine `diag_op` performs about 2×`nrow` flops. This information is then used to compute the total number of flops required to solve the eigenvalue problem. Here is an output for this example:

```
                                                Tue Oct  7 15:25:14 2008
TRLAN execution summary (exit status = 0) on PE 0
Number of SMALLEST eigenpairs            10 (computed)          10 (wanted)
Times the operator is applied:          587 (MAX:             2000 )
Problem size:                          1000 (PE:    0)        2000 (Global)
Convergence tolerance:            1.490e-08 (rel)        2.980e-05 (abs)
Maximum basis size:                     100
Restarting scheme:                        7
Number of re-orthogonalizations:        587
Number of (re)start loops:               35
Number of MPI processes:                  2
Number of eigenpairs locked:              3
time in OP:             0.0000e+00 sec
Re-Orthogonalization::  3.0000e-02 sec,   2.4765e+09 FLOP/S ( 7.4294e+07 FLOP)
Restarting::            3.0000e-02 sec,   1.0173e+09 FLOP/S ( 3.0520e+07 FLOP)
TRLAN on this PE:       6.0000e-02 sec,   0.0000e+00 FLOP/S ( 0.0000e+00 FLOP)
 -- Global summary --
                     Overall,           MATVEC,         Re-orth,        Restart,
Time(ave)            6.5000e-02,        5.0000e-03,     3.0000e-02,     2.5000e-02
Rate(tot)            8.5557e+08,        1.7610e+08,     2.4765e+09,     1.2208e+09
```

In this example run, two processes are used to run the example, thus the global problem size is 2000, where the local problem size `nrow` of each process is 1000. We note that since the global problem size increases as more processes are used to run this example, the times to solve this example eigenvalue problem can increase as more processors are used.

Finally, the computed approximate eigenvalues are printed by the subroutine `trl_check_ritz`, whose interface is:

```
void trl_check_ritz(
        void (*op)(cost int,const int,double*,const int,
```

```
                double*,const int,void*),
                        // matrix-vector multiply subroutine.
    trl_info *info,     // structure storing parameters.
    int nrow,           // local problem size.
    int ncol,           // number of computed eigenvalues.
    double *evec,       // computed eigenvectors.
    int ldevec,         // leading dimension of evec.
    double *eval,       // computed eigenvalues.
    int *check,         // check for solution convergence.
    double *res,        // residual norms of the eigenpairs (optional)
    double *exact,      // exact eigenvalues (optional)
    int lwrk,           // size of wrk
    double *wrk         // workspace (optional)
    )
```

The subroutine `trl_check_ritz` requires a workspace of size `nrow`+4×`ncol`. If the size of `wrk` is smaller than required, an additional workspace is internally allocated. Note that the computed eigenvalues `eval` and eigenvectors `evec`, and their residual norms `res` in the arguments of `trl_check_ritz` are returned by the subroutine `trlan` in the arguments `eval`, `evec`, and `wrk`, respectively, see [Example], page 3. Here is an output of `trl_check_ritz` for the example problem:

```
TRL_CHECK_RITZ:
          Ritz value       res norm   res diff  est error  diff w rq  act. error
    1.00000000000108      4.844e-13 -4.844e-13  2.347e-25 -1.085e-12  -1.085e-12
    2.00000000000043      5.272e-13 -5.272e-13  2.779e-25 -4.174e-13  -4.281e-13
    2.99999999999999      5.757e-13 -5.757e-13  3.314e-25  2.709e-14   7.105e-15
    3.99999999999993      6.630e-13 -6.622e-13  4.396e-25  8.216e-14   7.327e-14
    5.00000000000000      5.799e-13 -5.219e-13  3.363e-25 -1.066e-14   0.000e+00
    6.00000000000003      3.326e-12 -5.271e-14  1.106e-23 -6.217e-14  -3.375e-14
    6.99999999999996      1.596e-10 -1.901e-14  2.546e-20 -1.243e-14   4.352e-14
    7.99999999999989      6.939e-09 -1.101e-13  4.815e-17  9.504e-14   1.146e-13
    9.00000000000016      2.761e-07 -5.377e-14  7.621e-14 -1.847e-13  -1.563e-13
   10.00000000000117      1.022e-05 -5.892e-14  1.044e-10 -1.865e-13  -1.169e-12
```

Among the printed information, `res norm` is the actual residual norms of the eigenpairs (λ, v), i.e., $||Av - \lambda v||_2$, and `res diff` is the difference between the actual residual norm and the approximate residual norm `res` returned by `trlan`. `est error` is the estimated error norms computed from the residual norms and approximate eigenvalues. `diff w rq` is the difference between the computed eigenvalue λ and the value $v^T Av$, which is commonly referred to as the Rayleigh quotient, [Parlett 1998], page 25. Finally, if `exact` is provided in the argument, `act. error` shows the actual error in the computed eigenvalues.

On return from the subroutine `trl_check_ritz`, the argument `check` indicates results of internal solution convergence tests; `res diff` is less than 10^{-5}, `diff w rq` is less than `nrow`^{2}×`tol`, and `act. error` is less than 10×`nrow`^{2}×`tol`. If `check`= 0, this indicates all the internal checks are satisfied.

The above example shows how to compute eigenvalues of a symmetric matrix using nu–TRLan. For the solution of Hermitian eigenvalue problems, all the interfaces remain the same, except that complex numbers are stored in variables of type `trl_dcomplex`, whose format is compatible to that of the variable type `COMPLEX` of `LAPACK`. For example, the main computational subroutine has the interface:

```
void ztrlan(
void (*op)(const int, const int, trl_dcomplex*, const int,
   trl_dcomplex*, const int, void*), trl_info * info, int nrow,
   int mev, double *eval, trl_dcomplex * evec, int lde,
   trl_dcomplex * misc, int nmis, double *dwrk, int ldwrk)
```

and the `check_ritz` subroutine has the interface:

```
ztrl_check_ritz(
   void (*op)(const int, const int, trl_dcomplex*, const int,
   trl_dcomplex*, const int, void*), trl_info * info, int nrow,
   int ncol, trl_dcomplex * evec, int ldevec, double *eval,
   int *check, double *res, double *exact, int lwrk,
   trl_dcomplex * wrk)
```

A few examples that solve Hermitian eigenvalue problems using nu–TRLan are included under the '`examples`' directory. See the '`README`' file in the top directory '`nutrlan`' for more information. For the rest of this user guide, we will focus on the solutions of the symmetric eigenvalue problems. The extensions to the Hermitian problems are straight forward.

4 trl_info structure

To simplify the interfaces to nu–TRLan subroutines, all the required input parameters are stored in the `trl_info` structure. These parameters stored in `trl_info` can be manipulated through the following subroutines:

- `trl_init_info` initializes the parameters.
- `trl_set_debug` sets the monitored performance statistics.
- `trl_set_iguess` specifies the starting vector options.
- `trl_set_checkpoint` sets up the checkpoints.
- `trl_print_info` and `trl_terse_info` print the current parameters.

In this chapter, we will discuss these subroutines.

4.1 Initialization, `trl_init_info`

The subroutine `trl_init_info` initializes all the solver parameters and resets all the performance counters. It must be called before any other nu–TRLan subroutines. Its interface is

```
void trl_init_info(trl_info *info, int nrow, int maxlan,
                   int lohi, int ned, double tol,
                   int restart, int mxmv, int mpicom)
```

The arguments to the subroutine are:

`info: pointer to the structure.`
: On entry, `info` points to the structure to be initialized. The structure stores the solver parameters, some of which are specified by the rest of the arguments. See Section 4.6 [`trl_info` structure], page 13, for more information about the structure. On exit, `info` points to the initialized structure.

`nrow: local problem size.`
: On a distributed memory machine, the basis vectors are distributed by rows among the processes. The argument `nrow` specifies the number of rows of the basis vectors that are owned by this process. `nrow` may vary from process to process.

`maxlan: maximum projection subspace dimension.`
: When a static restart scheme is used, the iteration is restarted after `maxlan` basis vectors are computed. This determines the required workspace size and solution convergence rate of `trlan`. See Chapter 5 [`trlan` interface], page 15, and Section 5.3 [required workspace], page 16, for more information on the workspace requirement. If an adaptive restart scheme is used, `maxlan` specifies the upper-bound of the subspace dimension that is adjusted at every restart. See [restart scheme], page 9 and Section 6.2 [basis size], page 23, for more information on the restart schemes and the selection of the parameter `maxlan`.

`lohi: type of desired eigenvalues.`
: `lohi` indicates which end of the spectrum to compute. The choices are to compute either the smallest (`lohi` < 0) or the largest (`lohi` > 0) eigenvalues, or whatever converges first (`lohi` = 0).

`ned: number of desired eigenvalues.`
: nu–TRLan tries to compute `ned` eigenpairs with the given parameters.

`tol: required solution accuracy.`
: nu–TRLan computes approximate eigenvalues λ and their corresponding eigenvectors v of a symmetric or Hermitian matrix A. The relative residual norms of the convergent approximate eigenpairs (λ, v) are guaranteed to be less than `tol`, i.e., $\|Av - \lambda v\|_2 \leq \texttt{tol}||A||_2$, where $||A||_2$ is approximated by the largest absolute value of computed eigenvalues. If `tol` is a negative value, then it is set to be its default value, which is the square root of the machine precision. For example, on machines with 8-byte IEEE floating-point arithmetic, the default value is 2^{-26}.

`restart: restart scheme.`
: `restart` can be either $1, 2, \ldots, 8$. If `restart` is less than 1 or greater than 8, it is reset to be the default choice 7, which adaptively adjusts the projection subspace dimension at every restart, [LNBL-1059E], page 25. See Section 6.1 [restart scheme], page 23, for further discussion of this parameter.

`maxmv: maximum number of matrix operations.`
: The purpose of `maxmv` is to terminate the program in case of stagnation. If a negative value is provided, `maxmv` is set to be the default value `ned`×`ntot`, where `ntot` is the global problem size, i.e., `ntot` is the sum of `nrow` over all processors.

`mpicom: MPI communicator`
: `mpicom` is used only on a distributed memory system. If a negative value is provided, `trl_init_info` duplicates `MPI_COMM_WORLD` and uses the resulting communicator for its internal communication operations.

4.2 Starting vector options, `trl_set_iguess`

To start the iteration, nu–TRLan either uses a user-supplied starting vector, generates an arbitrary starting vector, or reads a set of checkpoint files. This *initial guess* option can be set by the subroutine `trl_set_iguess`, whose interface is:

```
void trl_set_iguess(trl_info *info, int nec, int iguess,
                    int ncps, char *oldcpf)
```

with the following arguments:

`info: structure to be updated.`
: The subroutine `trl_init_info`, [`trl_init_info` subroutine], page 8, must be called to initialize the data structure before `trl_set_iguess` is called.

`nec: number of convergent eigenvalues.`
: If `nec` is greater than zero, it specifies the number of the convergent approximate eigenpairs stored in the arrays `eval` and `evec` that are the arguments to the subroutine `trlan`, Chapter 5 [`trlan` subroutine], page 15. Specifically, the first `nec` elements of the array `eval` contain the convergent approximate eigenvalues, and the first `nec` columns of the array `evec` contain corresponding approximate eigenvectors. This allows `trlan` to resume the computation of eigenvalues and

eigenvectors from the previous runs of nu–TRLan. `trlan` can be restarted with an arbitrary number of vectors. However, these vectors have to satisfy the convergence criteria; see [convergence criteria], page 9. The subroutine `trl_init_info` sets `nec` to be zero.

`iguess: initial guess option.`
`iguess` specifies the initial guess option;

`>1:` nu–TRLan reads checkpoint files and uses their contents to resume the iteration. Checkpoint is explained in Section 4.3 [Checkpoint], page 10.

`1:` The user supplies a starting vector. The vector is stored in the first column of `evec`, which is an argument to the subroutine `trlan`.

`0:` nu–TRLan generates the starting vector, whose entries are all set to be one.

`<0:` nu–TRLan generates the starting vector, and applies a random perturbation to it.

`ncps: number of checkpoints.`
If `ncps` is greater than zero, this indicates that the checkpoint files are provided.

`oldcpf: leading portion of the checkpoint file name.`
The name of the checkpoint file is formed by appending the MPI process rank to `oldcpf`. On a sequential machine, the MPI process rank is zero. For example, if `oldcpf` is '`TRL_CHECKPOINT_`', the process with the MPI rank of 0 reads the checkpoint file named '`TRL_CHECKPOINT_0`'.

4.3 Checkpoint, `trl_set_checkpoint`

A checkpoint saves a state of the eigen solver such that `trlan` can be resumed from a previous run of `trlan`. All the necessary information to resume `trlan` is stored in checkpoint files. The subroutine `trl_set_iguess` specifies whether existing checkpoint files are used to resume the current iteration of `trlan`; see Section 4.2 [`trl_set_iguess` subroutine], page 9. On the other hand, to specify whether new checkpoint files will be written during the proceeding iterations, the following subroutine can be used:

```
void trl_set_checkpoint(trl_info *info, int cpflag, char *cpfile)
```

The arguments to this subroutine are:

`info: structure to be updated.`
The subroutine `trl_init_info` must be called before `trl_set_checkpoint` is called.

`cpflag: number of checkpoints.`
If `cflag` is greater than zero, the checkpoint file is updated `cpflag` number of times in the user-specified maximum number of iterations, `maxmv`, which is set by the subroutine `trlan`; see Chapter 5 [`trlan` subroutine], page 15. Only the most recent checkpoint will be available in the file. Each MPI process writes its own checkpoint in binary at the end of a restart process. Because of this, these checkpoint files can be read only on the same type of machines using the same

number of MPI processes. If `cpflag` is less than or equal to zero, no checkpoint files are written. `trl_init_info` sets `cpflag` to be zero.

`cpfile: leading portion of the checkpoint file.`
The name of the checkpoint file is formed by concatenating the MPI process rank at the end of `cpfile`. `trl_init_info` sets `cpfile` to be '`TRL_CHECKPOINT_`' by default.

4.4 Performance statistics, `trl_set_debug`

nu–TRLan allows a user to monitor the solution convergence. The information that can be monitored include the elements of the projected tridiagonal matrix, current approximate eigenvalues, their residual norms, and levels of the orthogonality among the basis vectors. This information is written to a separate *debug file* by each MPI process. The name of the file and how much information is monitored are controlled by the subroutine `trl_set_debug`. The interface to this subroutine is:

```
void trl_set_debug(trl_info *info, int msglvl, char *filename)
```

Short descriptions of the arguments are as follow:

`info: structure to be updated.`
The subroutine `trl_init_info` must be called before `trl_set_debug` is called.

`msglvl: message level.`
This parameter controls how much information is monitored. With a larger `msglvl`, more information is monitored, $0 \leq$`msglvl`≤ 10. The subroutine `trl_init_info` sets `msglvl` to be zero as the default value to indicate that nothing is monitored.

`filename: leading part of the debug file name.`
As with the checkpoint files used by the subroutine `trl_set_iguess`, see [`trl_set_iguess` subroutine], page 9, the debug file names are generated by concatenating `filename` with the MPI rank of this process.

4.5 Execution summary, `trl_print_info` and `trl_terse_info`

nu–TRLan includes the subroutine `trl_print_info` that allows a user to examine the current state of the eigensolver. This is useful for testing the solution convergence at the completion of the subroutine `trlan`; see Chapter 5 [`trlan` subroutine], page 15. The interface to `trl_print_info` is:

```
void trl_print_info(trl_info *info, int mvflop)
```

with the arguments:

`info: structure storing the current state.`
The structure stores the information such as the number of desired and convergent eigenvalues, number of matrix-vector multiply performed, and the CPU time spent in each phases of nu–TRLan. See [`trl_print_info` output], page 5, for an example of output.

`mvop: number of flops per matrix-vector multiply.`
It indicates the number of floating-point operations (flops) performed by this process during one matrix-vector multiply. This information is then used to

compute the total number of flops required to solve the eigenvalue problem. If `mvop` is not given, the relevant fields are left blank in the output.

For an example of using `trl_print_info`, see Chapter 3 [Example], page 3.

nu–TRLan also contains an additional subroutine, `trl_terse_info`, that prints a summary of the information stored in the `trl_info` structure. Its interface is:

```
void trl_terse_info(trl_info *info, FILE* ofp)
```

The arguments to the subroutine are:

`info: pointer to the structure.`
: The structure contains the information about the current state of nu–TRLan.

`ofp: a pointer to a file stream.`
: The subroutine `trl_terse_info` can output to any valid output file. This is different from `trl_print_info`, whose output file is set by the subroutine `trl_set_debug`; see Section 4.4 [`trl_set_debug` subroutine], page 11.

An example of output from this subroutine is:

```
MAXLAN:       100, Restart:         7,   NED: -      10,      NEC:        10
MATVEC:       587,  Reorth:       587, Nloop:        35,  Nlocked:         3
Ttotal:  0.060000,    T_op:  0.000000, Torth:  0.030000,   Tstart:  0.030000
```

The description of the printed information is as follows.

`MAXLAN:` user-specified maximum dimension of projection subspace. It is set by the subroutine `trl_init_info`; see [`trl_init_info` subroutine], page 8.

`Restart:` restart scheme, $0, 1, \ldots, 8$. It is set by `trl_init_info`.

`NED:` number of desired eigenvalues. It also specifies at which end of the spectrum the approximate eigenvalues were computed, i.e., at the largest (`+`), at the smallest (`-`), or both ends of the spectrum (`0`). It is set by `trl_init_info`.

`NEC:` number of convergent eigenvalues.

`MATVEC:` number of the times that the matrix-vector multiply is applied.

`Reorth:` number of reorthogonalization, i.e., each time the Gram-Schmidt procedure is called, this counter is incremented by one.

`Nloop:` number of outer iterations, i.e., number of restarts.

`Nlocked:` number of the approximate eigenpairs (λ, v) that are locked because their residual norms are small ($||Av - \lambda v||_2 \leq \epsilon ||A||_2$, where ϵ is the machine precision).

`Ttotal:` total time in seconds spent by nu–TRLan.

`T_op:` time in seconds spent performing the matrix-vector multiply.

`Torth:` time in seconds spent performing the reorthogonalizations.

`Tstart:` time in seconds spent in restart.

Note that the subroutine `trl_terse_info` prints only the local information, i.e., the flops performed and time spent by this processor, while `trl_print_info` also prints the global summary information; see [`trl_print_info` output], page 5 for an output from `trl_print_info`.

4.6 Member variables

As discussed earlier in this chapter, all the required solver parameters are stored in the `trl_info` structure. We list below all the member variables of `trl_info`. The variables are of type integer unless otherwise specified.

`npes` number of processors.

`my_pe` rank of the MPI process that owns this structure.

`mpicom` MPI communicator. See [mpicom], page 9

`nloc` local problem size. See [nrow], page 8

`ntot` global problem size, i.e., `ntot` is the sum of `nrow` over all processors.

`lohi` index to indicate which end of the spectrum to compute. See [lohi], page 8.

`restart` restart scheme. See Section 6.1 [restart scheme], page 23

`rfact` factor used with the restart scheme 8 to specify the projection subspace dimension. See Section 6.1 [restart scheme], page 23.

`ned` number of desired eigenvalues. See [ned], page 8

`nec` number of convergent eigenvalues. See [tol], page 9.

`locked` number of locked eigenvalues. See [locked], page 12.

`guess` initial guess option. See Section 4.2 [initial guess], page 9.

`tol` required solution accuracy. `tol` is of type double. See [tol], page 9.

`matvec` number of times that the matrix-vector multiply was applied.

`nloop` number of outer-loop iterations or restarts.

`north` number of times that the Gram-Schmidt procedure was invoked to perform reorthogonalization.

`nrand` number of times `trlan` generated random vectors in the attempt to recover from an invariant subspace.

`clk_rate` clock rate of the machine. `clk_rate` is of type `clock_t`.

`clk_max` maximum clock ticks. `clk_max` is of type `clock_t`.

`clk_tot, clk_op, clk_orth, and clk_res (of type clock_t);`
`tick_t, tick_o, tick_h, and tick_r (of type double):`
time spent performing matrix-vector multiply (`clk_op` and `tick_o`), reorthogonalization (`clk_orth` and `tick_h`), and restart (`clk_res` and `tick_r`), and the total time spent by `trlan` (`clk_tot` and `tick_t`). The four counters `clk_op`, `clk_orth`, `clk_res`, and `clk_tot` are used to accumulate the clock ticks returned from the intrinsic subroutine `clock`. Once the clock ticks become too large to store in the integer counters, their values are added to their corresponding counters of type double, and the integer counters are reset to be zero. Specifically, when the value of the integer counter decreases with the addition of new tick counts, then it is assumed to be too large to store in the integer counter. We also assume that `clock` wraps around when the tick returned by `clock` is smaller than the previously-recorded tick.

`flop, flop_h, and flop_r;`
`rflp, rflp_h, and rflp_r (of type double):`
: numbers of flops performed for reorthogonalization (`flop_h` and `rflp_h`) and for restart (`flop_r` and `rflp_r`), and the total flops (`flop` and `rflp`) excluding those used by matrix-vector multiply, i.e., the matrix-vector multiply subroutine is supplied by the user; see [matrix-vector multiply], page 11. When the numbers of flops become too large to store in the integer counters, they are added to their corresponding counters of type double, and the integer counters are reset to be zero (see the description of the counters used to store clock ticks for more information).

`tmv;`
`crat, tres, and trgt (of type double):`
: convergence factor of the approximate eigenvalues. The variable `crat` is the convergence factor over the previous restart-loop, i.e., `crat` $= e^{\log(||r_{\mathbf{matvec}}||_2/||r_{\mathbf{tmv}}||_2)/(\mathbf{matvec}-\mathbf{tmv})}$, where $||r_i||_2$ is the residual norm of the current target eigenvalue `trgt` after i matrix-vector multiplies, and `matvec` is the current number of times that the matrix-vector multiply is applied. The residual norm of the target at the previous restart, $||r_{\mathbf{tmv}}||_2$, is stored in `tres`, i.e., `tmv` is the number of matrix-vector multiplies at the previous restart. After `crat` is updated, `trgt` is set to be the next target eigenvalue, `tres` is set to be the current residual norm of the target, and `tmv` is set to be `matvec`.

`anrm (of type double):`
: estimate norm of the coefficient matrix. This is the largest absolute value of approximate eigenvalues computed. This value is primarily used in the convergence test, see [tol], page 9.

`stat:` current state of nu–TRLan. In Section 5.4 [error handling], page 17, this is described in detail.

`mvparam:` additional parameters passed to the matrix-vector multiply subroutine. See Section 5.2 [matrix-vector multiply], page 15 for more information.

5 `trlan` subroutine

In this section, we describe the main subroutine `trlan` that computes the approximate eigenpairs of symmetric matrices.

5.1 Interface

The main computation kernel of the nu–TRLan package is the subroutine `trlan`. The interface to the subroutine is:

```
void trlan(void (*op)(const int,const int,double*,const int,double*,
           const int,void*), trl_info *info, int nrow, int mev,
           double *eval, double *evec, int lde, int lwrk, double *wrk)
```

Most of the arguments of this subroutine are explained in Chapter 3 [Example], page 3. For completeness, we list them here.

`op:` matrix-vector multiply subroutine. In Section 5.2 [matrix-vector multiply], page 15, we discuss the interface to this subroutine.

`info:` pointer to the structure of type `trl_info`. See Chapter 4 [`trl_info` structure], page 8, for more details.

`nrow:` number of rows owned by this process. See [nrow], page 8, in the arguments of the `trl_init_info` stubroutine for more information.

`mev:` number of elements in array `eval` and number of columns in array `evec`, i.e., the maximum number of eigenpairs that can be stored in `eval` and `evec`.

`eval:` array used to store the computed approximate eigenvalues.

`evec:` array used to store the computed approximate eigenvectors.

`lde:` leading dimension of array `evec`. `lde` is expected to be at least as large as `nrow`.

`lwrk:` number of elements in the workspace `wrk`. If the user does not supply any workspace, it must be set to be a non-positive integer.

`wrk:` optional workspace. If sufficient amount of workspace is not provided, additional workspace is internally allocated. See Section 5.3 [workspace], page 16 for more details. `trlan` will return the residual norms of the converged eigenpairs in the first `nec` elements of `wrk`, where `nec` is the number of the convergent eigenpairs.

Recall that when `nec` in the arguments to the subroutine `trl_set_iguess` is greater than zero, the first `nec` elements of `eval` contain the convergent eigenvalues, and the first `nec` columns of `evec` contain the corresponding eigenvectors. See [`trl_set_iguess` subroutine], page 9 for more information.

5.2 Matrix-vector multiply

One of the arguments to the subroutine `trlan` is the pointer to the subroutine that computes the matrix-vector multiply with the coefficient matrix of the eigenvalue problem. This subroutine must have the following interface:

```
void diag_op(const int nrow, const int ncol, double *xin,
             const int ldx, double *yout, const int ldy,
             void *mvparam)
```

The arguments to the subroutine are:

`nrow:` local problem size. See [nrow_trlan], page 15, in the arguments of the `trlan` stubroutine for more information.

`ncol:` number of columns in the arrays `xin` and `yout`.

`xin:` array storing the input vectors to be multiplied.

`ldx:` leading dimension of the array `xin`.

`yout:` array to store the results of the matrix-vector multiply.

`ldy:` leading dimension of the array `yout`.

`mvparam:` optional argument for passing additional parameters to this subroutine. See [mvparam], page 14 for more information.

An example of the matrix-vector multiply is presented in Chapter 3 [example], page 3. There are a number of packages that can be referenced when implementing your own matrix-vector multiply subroutine. We provide below a short list of such packages:

`ACTS` http://acts.nersc.gov/

`NETLIB` http://www.netlib.org

`PETSc` http://www.mcs.anl.gov/petsc/

`SPARSKIT` http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html

5.3 Workspace

Inside the subroutine `trlan`, there are three workspaces `evec`, `base`, and `misc`. Their usages are as follows:

1. The user always provides the array `evec` as an argument to the subroutine `trlan`. The size of `evec` is `lde`×`mev`. `evec` is used to input the initial vectors and output the approximate eigenvectors. See Chapter 5 [trlan subroutine], page 15, for more information.
2. The array `base` is used to store the basis vectors when there is no more space in `evec`. Given the maximum basis size `maxlan`, the size of `base` is `(maxlan + 1 - mev)`×`nrow`. If `wrk` of a sufficient size is provided to the subroutine `trlan`, it is used to store `base`.
3. The array `misc` is used as internal workspaces. For example, it is used to store the projected matrix, the eigenvalues, and eigenvectors of the projected matrix, and is used as workspace for lower-level subroutines, including those from `LAPACK/BLAS` libraries. Its size should be at least `maxlan`×`(maxlan +10)`. Some subroutines might run faster with a larger `misc`. If `wrk` of a sufficient size is provided, it is used to store `misc`.

If the user provides a workspace `wrk` to `trlan`, then its size `lwrk` is checked to see if `misc` or `base` or both can fit in the workspace. If additional workspaced is required, a workspace of appropriate size is internally allocated.

5.4 Error handling

This section lists error codes that can be returned by the subroutine `trlan`. We also discusses possible remedies to the errors.

`0` This indicates a successful completion of `trlan`. However, it is possible that some of the desired eigenpairs have not yet converged. Check `nec` (the number of convergent eigenpairs) to see how many desired eigenpairs have converged; see Section 4.5 [`trl_print_info` subroutine], page 11, or [nec], page 13, for more information.

If some of the desired eigenpairs have not converged, the possible solutions are:

- If checkpoint files were written, resume the iteration by calling `trlan` with the checkpoint files; see Section 4.3 [Checkpoint], page 10, for more details.
- If the checkpoint files were not written, but the approximate eigenvectors are available, then make a linear combination of the eigenvectors and rerun `trlan` with the resulting vector as the starting vector; see [`trl_set_iguess` subroutine], page 9. In addition, generate checkpoint files for future use; see Section 4.3 [Checkpoint], page 10.
- Increase the maximum basis size (`maxlan`) and rerun `trlan`; see Section 4.1 [`trl_init_info` subroutine], page 8.
- Increase the maximum number of iterations allowed (`maxmv`) and rerun `trlan`, see Section 4.1 [`trl_init_info` subroutine], page 8.
- Use a different restart scheme; see Section 6.1 [restart scheme], page 23.

`-1` The value of `nrow` in the argument to the subroutine `trlan` does not match with the local problem size `nloc` stored in the `trl_info` structure, [nloc], page 13.

SOLUTION: Make sure that the subroutine `trl_init_info` is called before `trlan`, and the arguments to both subroutines are correct for the intended eigenvalue problem; see Section 4.1 [`trl_init_info` subroutine], page 8, and Chapter 5 [`trlan` subroutine], page 15.

`-2` In the arguments to `trlan`, the leading dimension `lde` of the array `evec` is smaller than the local problem size `nloc` of the `trl_info` structure.

SOLUTION: Make sure that the subroutine `trl_init_info` is called before `trlan`, and the arguments to both subroutines are correct for the intended eigenvalue problem; see Section 4.1 [`trl_init_info` subroutine], page 8. Allocate the array `evec` with the leading dimension larger or equal to `nrow` specified by the `trl_init_info` subroutine.

`-3` In the argument to `trlan`, the array size `mev` of `eval` is too small to store the desired eigenvalues.

SOLUTION: Increase the size of array `eval` and number of columns in `evec`, see Chapter 5 [`trlan` subroutine], page 15. Check the number `ned` of desired eigenvalues in the arguments to the `trl_init_info` subroutine; see [`trl_inif_info` subroutine], page 8

`-4` nu–TRLan failed to allocate workspace of size `maxlan`×`(maxlan+10)`, which is used to store the projected matrix.

SOLUTION:

- Reduce the size of `maxlan`; see Chapter 5 [`trlan` subroutine], page 15.
- If additional workspace is available, give nu–TRLan more workspace.
- If possible, increase the swap file/partition size.

`-5` nu–TRLan failed to allocate memory to store the Lanczos basis vectors. The size of the required workspace is `(maxlan + 1 - mev)`×`nrow`.

SOLUTION: See solutions for error code `-4`.

`-11` nu–TRLan does not have enough workspace to perform the Gram-Schmidt procedure for reorthogonalization. This happens when the lower-level subroutine `trlanczos` is directly called with an insufficient workspace.

SOLUTION: Increase the workspace to `trlanczos`.

`-12` nu–TRLan does not have enough workspace to compute eigenvalues of a symmetric tridiagonal projected matrix. This happens when the lower-level subroutine `trlanczos` is directory called with an insufficient workspace.

SOLUTION: Increase the workspace to `trlanczos`.

`-101` The reorthogonalization subroutine does not have enough workspace. This happens when the lower-level subroutine `trl_orth` is directly called with an insufficient workspace.

SOLUTION: Increase the workspace to `trl_orth`.

`-102` The computation of the residual norm overflowed or underflowed.

SOLUTION:

- This can happen when the workspace is not as large as the user indicated. Check the sizes of workspaces including the space to store the eigenvalues and eigenvectors; see Chapter 5 [`trlan` subroutine], page 15.
- If the initial number of convergent eigenvalues (`nec`) is not zero, make sure that the convergent eigenvalues are stored in the first `nec` elements of `eval`, and the corresponding eigenvalues are stored in the first `nec` columns of `evec`; see [`trl_set_iguess` subroutine], page 9.

`-112` nu–TRLan failed to generate an orthogonal transformation to reduce the projected matrix into a tridiagonal matrix, i.e., `LAPACK` subroutine `dsytrd`/`ssytrd` failed.

SOLUTION: Make sure `LAPACK` is installed correctly. See suggestions for error `-102`.

`-113` nu–TRLan failed to apply the orthogonal transformation to reduce the projected matrix into a tridiagonal matrix, i.e., `LAPACK` subroutine `dorgtr`/`sorgtr` failed.

SOLUTION: See solutions to error `-112`.

`-121` There was not sufficient workspace to compute the eigenvalues of a tridiagonal matrix. This error occurs when the actual size of workspace `wrk` passed to `trlan` is not `lwrk`; see Chapter 5 [`trlan` subroutine], page 15.

SOLUTION: See solutions to error `-112`.

`-122` nu–TRLan failed to compute the eigenvalues of a tridiagonal matrix, i.e., `LAPACK` subroutine `dstqrb` failed.

SOLUTION: See solutions to error `-112`.

`-131` There was not sufficient workspace to compute the eigenvectors of a tridiagonal matrix. This error occurs when a workspace of incorrect size is provided.

SOLUTION: See solution to error `-112`.

`-132` nu–TRLan failed to compute the eigenvectors of a tridiagonal matrix. Specifically, `LAPACK` subroutine `dstein/sstein` failed.

SOLUTION: See solutions to error `-112`.

`-141` There was not sufficient workspace to compute the eigenvectors of a tridiagonal matrix.

SOLUTION: See solutions to error `-102`.

`-142` nu–TRLan failed to compute the eigenvalues of a tridiagonal matrix, i.e., the `LAPACK` subroutine `dsyev/ssyev` failed.

SOLUTION: See solution to error `-112`.

`-143/144` nu–TRLan could not match the computed eigenvalues selected to be saved with the eigenvalues found by `dsyev/ssyev`.

SOLUTION: See solutions to error `-112`.

`-201` The Gram-Schmidt procedure was called with an insufficient workspace. This happens when the lower-level subroutine `trl_cgs` is directly called.

SOLUTION: Increase the workspace size for `trl_cgs`. If you did not call `trl_cgs` directly, make sure workspace size `lwrk` matches the actual size of `wrk` when calling `trlan`.

`-202/203` The Gram-Schmidt process failed to orthogonalize a new vector against the previous basis vectors. This indicates two possible sources of the problem: either the previous basis vectors are not orthogonal, or the newly-generated random vector belongs to the space spanned by the previous basis vectors.

SOLUTION: Initialize each process with a different random number seed. If this does not fix the problem, see the solutions to error `-102`.

`-204` The vector norm after orthogonalization is not a valid floating-point number.

SOLUTION: See solutions to error `-102`.

`-211` The leading dimension of the array `evec` in the argument to `trlan` is not large enough to store the vectors in a checkpoint file.

SOLUTION: Make sure the checkpoint files are for the same problem and are generated on the same type of machine using the same number of processors; see Section 4.3 [Checkpoint], page 10.

`-212` A checkpoint file could not be opened for reading.

SOLUTION: Make sure the checkpoint files exist; see Section 4.3 [Checkpoint], page 10 or solutions for `-211`.

`-213` The array size stored in a checkpoint file is different from that specified by the user.

SOLUTION: See solutions for `-211`.

`-214` The number of the vectors stored in a checkpoint file is greater than `maxlan`.

SOLUTION: Increase `maxlan` in the argument to the subroutine `trlan`; see Chapter 5 [`trlan` subroutine], page 15.

`-215` An error was encountered while reading a checkpoint file.

SOLUTION: See solutions for `-211`.

`-216` An error was encountered while closing a checkpoint file.

SOLUTION: This error can be ignored in many cases. Consult your system administrator.

`-221` A checkpoint file could not be opened for writing.

SOLUTION: Make sure the checkpoint file is not being used for other tasks, and that you have permission to write files in the directory where the program is running.

`-222` An error was encountered while writing a checkpoint file.

SOLUTION: Make sure there is enough space on the disk to store the checkpoint files.

`-223` An error was encountered while closing a checkpoint file.

SOLUTION: This error can be ignored in many cases. Consult your system administrator.

For further information, please feel free to contact the authors; see Chapter 7 [contact information], page 24.

5.5 Fortran interface

To provide an interface for Fortran programs, nu-TRLan include a fortran module `trlan_info_t`, which stores the same information stored in the `C` structure `trlan_info`; see Section 4.6 [`trl_info` structure], page 13. Furthermore, the Fortran interface to the matrix-vector multiply subroutine is defined as

```
typedef void (*trl_matvec) (int *pnrow, int *pncol, double *x, int *pldx,
                            double *y, int *pldy);
```

The arguments to the subroutine are:

`pnrow:` pointer to the local problem size. See [nrow_trlan], page 15, in the arguments of the `trlan` stubroutine for more information.

`pncol:` pointer to the number of columns in the arrays `xin` and `yout`.

`xin:` array storing the input vectors to be multiplied.

`pldx:` pointer to the leading dimension of the array `xin`.

`yout:` array to store the results of the matrix-vector multiply.

`pldy:` pointer to the leading dimension of the array `yout`.

See Section 5.2 [matrix-vector multiply], page 15 for the C interface to this subroutine.

To build the nu–TRLan library with the Fortran interface to the matrix-vector multiply subroutine, the library needs to be rebuilt with `-DTRL_FORTRAN_COMPATIBLE`. This can be done by adding the following line in your `Make.inc` file:

```
CFLAGS = -DTRL_FORTRAN_COMPATIBLE
```

and invoking the following command from the top directory '`nutrlan`':

```
% make clean
% make lib
```

Finally, the Fortran module can be compiled by invoking the following command from the top directory:

```
% make ftrlan
```

This will generate the module `trlan_info_t` under the sub-directory '`FORTRAN`'.

There are a couple of Fortran examples using nu–TRLan included under the sub-directory '`example`'. Here, we present a simple Fortran example program whose parallel `C` version is presented in Chapter 3 [example], page 3:

```
 Program simple
 Use trlan_info
 Implicit None
 Integer, Parameter :: nrow=1000, lohi=-1, mev=100,
+                      ned=10, maxlan=200, restart=1
 Real(8) :: eval(mev), evec(nrow, mev), exact(mev)
 Real(8), DIMENSION(:), ALLOCATABLE :: res, wrk
 Type(trlan_info_t) :: info
 Integer :: i, lwrk, check
 External diag_op
 lwrk = maxlan*(maxlan+10)
 ALLOCATE( res(lwrk) )
 ALLOCATE( wrk(lwrk) )
 Call trl_init_info(info,%VAL(nrow),%VAL(maxlan),%VAL(lohi),
+       %VAL(ned),%VAL(1.4901D-8),%VAL(restart),%VAL(2000000))
 Call trl_set_iguess(info,%VAL(0),%VAL(1),%VAL(0),%VAL(0))
 eval(1:nrow)   = 0.0D0
 evec(1:nrow,1) = 1.0D0
 Call trlan(diag_op, info, %VAL(nrow), %VAL(mev), eval, evec,
+           %VAL(nrow), %VAL(lwrk), res )
 Call trl_print_info(info, %VAL(3*nrow))
 Do i = 1, mev
    exact(i) = i*i
 End Do
 i = info%nec
 Call trl_check_ritz(diag_op, info, %VAL(nrow),
+       %VAL(i),evec(:,1:i),%val(nrow),eval(1:i),
+       check, res, exact, %val(lwrk), wrk )
 DEALLOCATE( res,wrk )
 End Program simple
```

```
Subroutine diag_op(nrow, ncol, xin, ldx, yout, ldy)
Implicit None
Integer, Intent(in) :: nrow, ncol, ldx, ldy
Real(8), Dimension(ldx*ncol), Intent(in) :: xin
Real(8), Dimension(ldy*ncol), Intent(out) :: yout
Integer :: i, j, ioff, joff
Do j = 1, ncol
  ioff = (j-1)*ldx
  joff = (j-1)*ldy
  Do i = 1, nrow
     yout(joff+i) = i*xin(ioff+i)
  End Do
End Do
End Subroutine diag_op
```

This example program can be compiled using the following command from the '`example`' sub-directory:

```
% make fsimple
```

This will generate an executable called '`fsimple`', which can then be run as

```
% ./fsimple
```

For more information on Fortran example programs, see the '`README`' file under the top directory '`nutrlan`'.

6 Solver parameters

The performance of nu–TRLan depends on a few user-specified parameters; see Chapter 4 [`trl_info` structure], page 8. Optimal values of the parameters vary with the eigenvalue problem and the target machine. In this chapter, we will discuss the selection of these parameters.

6.1 Restart scheme

Effective restart schemes for the Lanczos method are still active researches area. nu–TRLan implements the same six restart schemes that are implemented in the original TRLan software package. For information on these restart schemes, see [TRLan User Guide], page 26. In addition to these six schemes, nu–TRLan also implements restart schemes 7 and 8. The advantage of using these two additional restart schemes is that the dimension of the projection subspace is adjusted at every restart. See [LNBL-1059E], page 25, for more information on restart scheme 7. Restart scheme 8 is a modification of scheme 7, where the next projection subspace is set to be `k`×`rfact`, see [rfact], page 13. The default restart scheme is 7.

6.2 Maximum projection dimension

If restart scheme 7 or 8 is used, the dimension of the projection subspace is adjusted at every restart. Hence, the maximum basis size `maxlan` (see [`trl_init_info` subroutine], page 8) should be set to be the maximum number of vectors that can be stored in the available memory space on the machine. When the static subspace dimension is used, the selection of optimal basis size is a difficult task. See [TRLan User Guide], page 26 for some recommendations.

6.3 Solution accuracy

In nu–TRLan, an approximate eigenpair (λ, v) is said to be converged when $||Av - \lambda v|| <$`tol`$||A||$, where `tol` is a user-specified solution accuracy; see [`trl_init_info` subroutine], page 8 When `tol` is set to be 10^{-k}, k digits of accuracy is typically achieved by the convergent eigenpairs.

7 Acknowledgements and contact information

This work was supported in part by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources from the National Energy Research Scientific Computing Center, which is supported by the Office of Energy Research of the U.S. Department of Energy.

The authors of nu–TRLan and this document can be contacted at the following email addresses: ic.yamazaki@gmail.com (Ichitaro Yamazaki), kwu@lbl.gov (Kesheng Wu), and hdsimon@lbl.gov (Horst Simon).

8 References

1. D. Calvettis, L. Reichel, and D. Sorensen. *An implicitly restarted Lanczos method for large symmetric eigenvalue problems.* Electronic Transactions on Numerical Analysis, 2:1–21, 1994.
2. The Basic Linear Algebra Subroutines. http://www.netlib.org/blas/.
3. M. Crouzeix, B. Philippe, and M. Sadkane, *The Davidson method.* SIAM J. Sci. Comput., 15:62–76, 1994.
4. J. Cullum and R. A. Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations: Theory*, volume 3 of *Progress in Scientific Computing.* Birkhauser, Boston, 1985.
5. J. Cullum and R. A. Willoughby. *Lanczos algorithms for large symmetric eigenvalue computations: Programs*, volume 4 of *Progress in Scientific Computing.* Birkhauser, Boston, 1985.
6. J. W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, 1997.
7. G. H. Golub and C. F. van Loan. *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1996.
8. C. Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators.* J. Res. Nati. Bur. Stand., 45:255–281, 1950.
9. The Linear Algebra Package. www.netlib.org/lapack/.
10. C. Lehouc, D. Sorensen, and C. Yang. *ARPACK User Guide: solution of large scale eigenvalue problems with implicitly restarted Arnoldi methods.* SIAM, Philadelphia, 1998.
11. Message Passing Interface (MPI) forum. http://www.mpi-forum.org/.
12. P. B. Morgan. *On restarting the Arnoldi method for large nonsymmetric eigenvalue problems.* Mathematics of Computation. 65:1213–1230, 1996
13. B. N. Parlett. *The symmetric eigenvalue problem.* SIAM, Philadelphia, PA, 1998.
14. A. Ruhe. *Rational Krylov sequence methods for eigenvalue computation.* Lin. Alg. Appl., 58:391–405, 1984.
15. Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, 1993.
16. H. D. Simon. *The Lanczos algorithm with partial reorthogonalization.* Math. Comp., 42:115–142, 1984.
17. A. Stathopoulos and J. R. McCombs. *PRIMME: Preconditioned iterative multimethod eigensolver.* http://www.cs.wm.edu/~andreas/software/doc.pdf, 2005.
18. A. Stathopoulos, Y. Saad, and K. Wu. *Dynamic thick restarting of the Davidson and the implicitly restarted Arnoldi methods.* SIAM J. Sci. Comput.. 19:227–245, 1998.
19. G. W. Steward. *Adjusting the Rayleigh Quotient in semiorthogonal Lanczos methods.* SIAM J. Scient. Comput., 24:201–207, 2002.
20. I. Yamazaki, Z. Bai, H. Simon, L.-W. Wang, and K. Wu. *Adaptive projection subspace dimension for the thick-restart Lanczos method.* Technical Report LBNL-1059E, Lawrence Berkeley National Laboratory, 2008

21. K. Wu and H. Simon. *Thick-restart Lanczos method for symmetric eigenvalue problems.* SIAM J. Matrix Anal. Appl., 22(2):602–616, 2001.
22. K. Wu and H. Simon. *Dynamic Retarting Schemes For Thick-Restart Lanczos Method.* Technical Report 42982, Lawrence Berkeley National Laboratory, 1999.
23. K. Wu and H. Simon. *TRLan User Guide.* Technical Report 42953, Lawrence Berkeley National Laboratory, 1999.

Index

C

Checkpoints 10
Contact information 24
Convergence factor 14

E

Eigen pairs, converged 9
Eigen pairs, locked 12
Error handling 17
Example 3

I

Installation 2

M

Matrix-vector multiply 15
Maximum projection dimension 23

P

Performance statistics, flops 13
Performance statistics, output 11
Performance statistics, setup 11
Performance statistics, time 13

R

References 25
Restart schemes 23

S

Solution accuracy 23

T

`trl_info` structure 13
`trl_info` structure, initialization 8
`trl_info` structure, performance statistics 11
`trl_info` structure, printing 11
`trlan` subroutine 14

W

Workspace 16
Workspace for Gram-Schmidt 19
Workspace, input 15