TRLIB Documentation

Release 0.1

Felix Lenders

Jul 19, 2023

Contents

TRLib is a library that provides various methods related to the trust region subproblem and in particular implements the Generalized Lanczos Method [Gould1999] for iterative solution of the trust region subproblem:

A detailed description of the problem and the method as well as the implementation can be found in [Lenders2016].

CHAPTER 1

Contents:

1.1 Installation

1.1.1 Dependencies

You have to make sure that the following requirements are provided:

- BLAS
- LAPACK
- CMake

The following dependencies are optional:

- **for the unittests:**
  - Check
- **for the documentation:**
  - sphinx with read the docs theme
  - numpydoc
- **for the python interface:**
  - Python Header, TRLIB works with Python 2 and 3 and compiles for the versions it finds
  - Cython
  - NumPy
  - SciPy
- **for the matlab interface:**
  - MATLAB with mex compiler and header files

1.1.2 Ubuntu/Debian Packages

To install all dependencies in a Ubuntu/Debian environment:

If you want to use Python 3:

```
sudo apt-get install cmake check build-essential python3-dev python3-numpy python3-
↪scipy cython3 liblapack-dev libblas-dev python3-sphinx python3-sphinx-rtd-theme
```

If you want to use Python 2:

```
sudo apt-get install cmake check cython build-essential python-dev python-numpy␣
↪python-scipy cython liblapack-dev libblas-dev python-sphinx python-sphinx-rtd-theme
```

1.1.3 Compilation

TRLIB is set up to create out of source builds using CMake. First create a build directory and change to that:

mkdir build cd build

Set up CMake in this directory:

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

Instead of `Release` you may also choose `Debug` which disables compiler optimization and enables debugging.

You may want change settings as described below in CMake, especially you have to turn on compilation of the python/matlab interface if you wish. After that execute:

```
ccmake .
```

Press `c` to reconfigure with changes and `q` to exit.

You can now compile TRLIB, generate the documentation and run the tests by executing:

```
make
make test
make doc
make install
```

Depending on the installation location, you might have to execute `make install` as super user:

```
sudo make install
```

1.1.4 CMake Options

Option	default	Description
TRLIB_MEASURE_TIME	OFF	measure time for trlib function calls
TRLIB_MEASURE_SUBTIME	OFF	measure time for blas and lapack function calls
TRLIB_BUILD_MEX	OFF	build matlab interface
TRLIB_BUILD_PYTHON2	OFF	build python 2 interface
TRLIB_BUILD_PYTHON3	OFF	build python 3 interface

1.2 Matlab Interface

1.2.1 Installation

Build trlib as declared in installation with CMake flag `TRLIB_BUILD_MEX` set to `ON`. At the moment, there is nothing in CMake to install the interface. You have to ensure that `$TRLIB_BUILD_DIR/bindings/matlab` is found by your Matlab Installation.

1.2.2 Usage

From Matlab you can access trlib via the functions `trlib`, `trlib_options`, `trlib_problem`, `trlib_set_hotstart` and `trlib_solve`.

`trlib` is a convenience function above the others that does everything in one step:

```
>> Hess = diag(sparse(linspace(-1, 100, 10000)));
>> grad = ones(10000, 1);
>> [x, flag] = trlib(Hess, grad, 0.1);
>> norm(x)
```

In this example we solve the indefinite problem

$$\min_{x \in \mathbb{R}^{10000}} \frac{1}{2} \sum_{i=1}^{10000} \left((-1 + 101 \frac{i-1}{9999}) x_i^2 + x_i \right) \quad \text{s.t. } \|x\| \leq 0.1$$

1.2.3 Hotstart

It is possible to hotstart the solution in the case that only the radius in the norm constraint changes. This is typically the case in applications in nonlinear programming after a rejected step. To employ hotstart after solution, you have to store the *TR* cell structure that contains data of the previous solution process and call *trlib_set_hotstart* before the next solution:

```
>> Hess = diag(sparse(linspace(-1, 100, 10000)));
>> grad = ones(10000, 1);
>> [x, flag, prob, TR] = trlib(Hess, grad, 0.1);
>> norm(x)
>> trlib_set_hotstart(TR);
>> [x, flag, prob, TR] = trlib_solve(prob, 0.05, TR);
>> norm(x)
```

1.2.4 Matlab API Documentation

For a documentation of the matlab API use the `help` command in matlab on the functions `trlib`, `trlib_options`, `trlib_problem`, `trlib_set_hotstart` and `trlib_solve`.

1.3 Python Interface

1.3.1 Installation

We don't have a `setup.py` yet but use CMake to build the interface. Ensure that either `TRLIB_BUILD_PYTHON2` or `TRLIB_BUILD_PYTHON3` is set to `ON` before compilation, depending on the python version you like to use. At the moment you have to ensure that `$TRLIB_DIR/build` is contained in your `$PYTHON_PATH`.

1.3.2 Usage

Use the function *trlib_solve()* to solve a trust region subproblem und *umin()* to solve an unconstrained non-linear optimization problem with a standard trust-region algorithm.

1.3.3 Functions

trlib_solve(*hess*, *grad*, *radius*, *invM = lambda x: x*, *TR=None*, *reentry=False*, *verbose=0*, *ctl_invariant=0*, *convexify=1*, *earlyterm=1*)

Solves trust-region subproblem

$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^T H x + g^T x \quad \text{s.t. } \|x\|_M \leq r$

with a projected CG/Lanczos method.

Parameters

- **hess** (*{sparse matrix, dense matrix, LinearOperator}*) – hessian matrix/operator *H* with shape (n,n)
- **grad** (*array*) – gradient *g* with shape (n,1)
- **radius** (*float*) – trust-region radius *r*
- **invM** (*{sparse matrix, dense matrix, LinearOperator}, optional*) – inverse of matrix/operator defining trust-region constraint norm, default: identity, acts as preconditioner in CG/Lanczos
- **TR** (*dict, optional*) – TR output of previous call for hotstarting
- **reentry** (*boolean, optional*) – set this to *True*, if you want to resolve with all data fixed but changed trust-region radius, provide *TR* of previous call
- **verbose** (*int, optional*) – verbosity level
- **ctl_invariant** (*int, optional*) – flag that determines how to treat hard-case, see C API of *trlib_krylov_min()*
- **convexify** (*int, optional*) – flag that determines if resolving with convexified should be tried if solution seems unrealistic
- **earlyterm** (*int, optional*) – flag that determines if solver should terminate early prior to convergence if it seems unlikely that progress will be made soon

Returns

(sol, TR): solution vector and trust-region instance data, needed for warmstart

- TR['ret'] gives return code of *trlib_krylov_min()*,
- TR['obj'] gives objective funtion value,

- TR['lam'] lagrange multiplier

Return type (array, dict)

Example Solve a sample large-scale problem with indefinite diagonal hessian matrix:

```
>>> import scipy.sparse
>>> import numpy as np
>>> H = scipy.sparse.diags(np.linspace(-1.0, 100.0, 1000),0)
>>> g = np.ones(1000)
>>> x, TR = trlib.trlib_solve(H, g, 1.0)
>>> np.linalg.norm(x)
1.0000000000000002
>>> x, TR = trlib.trlib_solve(H, g, .5, reentry=True, TR=TR)
0.50000000000000011
```

umin (*obj*, *grad*, *hessvec*, *x*, *tol=1e-5*, *eta1=1e-2*, *eta2=.95*, *gamma1=.5*, *gamma2=2.*, *itmax=-1*, *verbose=1*)

Standard Trust Region Algorithm for Unconstrained Optimization Problem:

$\min_{x \in \mathbb{R}^n} f(x)$

This implements Algorithm 6.1 of [Gould1999] with slight modifications:

- check for descent
- aggresive trust region reduction upon failed step if next iteration will have the same subproblem solution

Parameters

- **obj** (*function*) – callback that computes $x \mapsto f(x)$
- **grad** (*function*) – callback that computes $x \mapsto \nabla f(x)$
- **hessvec** (*function*) – callback that computes $(x, d) \mapsto \nabla^2 f(x) \cdot d$
- **x** (*array*) – starting point
- **tol** (*float, optional*) – convergence tolerance, convergence achieved if $\|\nabla f(x)\|_2 \leq \texttt{tol}$
- **eta1** (*float, optional*) – tolerance to discard step: $\rho = \frac{\text{actual reducion}}{\text{predicted reduction}} \leq \eta_1$
- **eta2** (*float, optional*) – tolerance to enlarge trust region: $\rho = \frac{\text{actual reducion}}{\text{predicted reduction}} \geq \eta_2$
- **gamma1** (*float, optional*) – reduction factor for trust region radius upon failed step
- **gamma2** (*float, optional*) – blow-up factor for trust region radius upon succesfully accepted step
- **itmax** (*int, optional*) – maximum number of iterations
- **verbose** (*int, optional*) – verbosity level

Returns last point, solution in case of convergence

Return type array

Example Compute the minimizer of the extended Rosenbrock function in R^10:

```
>>> import trlib
>>> import numpy as np
>>> import scipy.optimize
>>> trlib.umin(scipy.optimize.rosen, scipy.optimize.rosen_der,
        scipy.optimize.rosen_hess_prod, np.zeros(5))
```

(continues on next page)

(continued from previous page)

```
it   obj          ||g||        radius     step       rho           ?
↪nhv
  0 +4.0000e+00 4.0000e+00 4.4721e-01 4.4721e-01 -3.9705e+00  -    2
  1 +4.0000e+00 4.0000e+00 2.2361e-01 2.2361e-01 +6.4377e-01  +    0
  2 +3.7258e+00 1.0252e+01 2.2361e-01 4.5106e-02 +1.0011e+00  +    1
  3 +3.4943e+00 2.3702e+00 4.4721e-01 4.4721e-01 -1.6635e+00  -    3
  4 +3.4943e+00 2.3702e+00 2.2361e-01 2.2361e-01 +6.3041e-01  +    0
  5 +3.1553e+00 1.0513e+01 2.2361e-01 3.2255e-02 +1.0081e+00  +    1
  6 +2.9844e+00 3.0912e+00 4.4721e-01 4.4721e-01 -8.6302e-01  -    3
  7 +2.9844e+00 3.0912e+00 2.2361e-01 2.2361e-01 +8.2990e-01  +    0
  8 +2.5633e+00 8.4640e+00 2.2361e-01 4.1432e-02 +1.0074e+00  +    2
  9 +2.4141e+00 3.2440e+00 4.4721e-01 4.4721e-01 -3.3029e-01  -    4
 10 +2.4141e+00 3.2440e+00 2.2361e-01 2.2361e-01 +7.7786e-01  +    0
 11 +1.9648e+00 6.5362e+00 2.2361e-01 2.2361e-01 +1.1319e+00  +    4
 12 +1.4470e+00 5.1921e+00 4.4721e-01 4.0882e-01 +1.7910e-01  +    5
 13 +1.3564e+00 1.6576e+01 4.4721e-01 7.9850e-02 +1.0404e+00  +    2
 14 +6.8302e-01 3.7423e+00 8.9443e-01 4.0298e-01 -4.9142e-01  -    5
 15 +6.8302e-01 3.7423e+00 3.6268e-01 3.6268e-01 +8.1866e-02  +    0
 16 +6.5818e-01 1.3817e+01 3.6268e-01 4.8671e-02 +1.0172e+00  +    1
 17 +3.1614e-01 5.9764e+00 7.2536e-01 1.3202e-02 +1.0081e+00  +    2
 18 +2.8033e-01 1.4543e+00 1.4507e+00 3.5557e-01 -8.6703e-02  -    5
 19 +2.8033e-01 1.4543e+00 3.2001e-01 3.2001e-01 +3.4072e-01  +    0
it   obj          ||g||        radius     step       rho           ?
↪nhv
 20 +2.3220e-01 1.0752e+01 3.2001e-01 2.0244e-02 +1.0101e+00  +    1
 21 +1.2227e-01 5.0526e+00 6.4002e-01 1.1646e-02 +1.0073e+00  +    2
 22 +9.6271e-02 1.6755e+00 1.2800e+00 1.4701e-03 +9.9992e-01  +    1
 23 +9.5040e-02 6.1617e-01 2.5601e+00 3.3982e-01 -3.6365e-01  -    5
 24 +9.5040e-02 6.1617e-01 3.0584e-01 3.0584e-01 +1.3003e-01  +    0
 25 +8.6495e-02 9.2953e+00 3.0584e-01 1.1913e-02 +1.0071e+00  +    1
 26 +3.0734e-02 4.0422e+00 6.1167e-01 7.7340e-03 +1.0056e+00  +    2
 27 +1.7104e-02 1.1570e+00 1.2233e+00 9.7535e-04 +1.0004e+00  +    1
 28 +1.6540e-02 3.6078e-01 2.4467e+00 2.0035e-01 +5.5171e-01  +    5
 29 +8.6950e-03 3.5281e+00 2.4467e+00 3.7360e-03 +1.0023e+00  +    1
 30 +2.0894e-03 1.4262e+00 4.8934e+00 2.4287e-03 +1.0018e+00  +    2
 31 +5.8038e-04 3.5326e-01 9.7868e+00 4.3248e-04 +1.0004e+00  +    2
 32 +5.1051e-04 7.3221e-02 1.9574e+01 4.3231e-02 +9.9862e-01  +    5
 33 +1.4135e-05 1.4757e-01 3.9147e+01 2.0304e-04 +1.0001e+00  +    3
 34 +7.1804e-07 1.3804e-02 7.8294e+01 1.5512e-03 +1.0008e+00  +    5
 35 +2.2268e-11 1.8517e-04 1.5659e+02 2.0956e-06 +1.0000e+00  +    5
 36 +3.7550e-21 1.2923e-10
```

1.4 Julia Interface

1.4.1 Installation

Ensure that `TRLIB_INSTALL_DIR/lib` is part of your `LD_LIBRARY_PATH` environment variable.

You include the Julia interface by adding:

```
include("TRLIB_DIR/bindings/julia/trlib.jl")
using trlib
```

to your Julia code.

1.4.2 Usage

The interface allows to solve the trust region problem

$$\min_{x\in\mathbb{R}^n} \tfrac{1}{2}x^T Hx + x^T g \quad \text{s.t. } \|x\| \leq \text{radius}$$

respective

$$\min_{x\in\mathbb{R}^n} \tfrac{1}{2}x^T Hx + x^T g \quad \text{s.t. } \|x\|_M \leq \text{radius}$$

with $\|x\|_M = \sqrt{x^T M x}$.

The module provides a type `trlib_data` to hold the necessary data of a trust region problem and a function `trlib_solve` to solve the trust region problem.

To instaniate the data holding a trust region problem instance, execute:

```
TR = trlib.trlib_data(H, g)
```

respective:

```
TR = trlib.trlib_data(H, g, invM)
```

where *H* is such data the action *H * p* is defined, yielding Hp and *invM* such that *invM * p* is defined yielding $M^{-1}p$.

You can then solve the problem with:

```
trlib.trlib_solve(TR, radius)
```

and get the solution as *TR.sol*, the lagrange multiplier as *TR.lam* and the objective value as *TR.obj*.

To hotstart the solution process with changed radius, just execute *trlib_solve* again.

Example Solve a sample large-scale problem with indefinite diagonal hessian matrix:

```
julia> include("TRLIB_DIR/bindings/julia/trlib.jl");
julia> using trlib;
julia> H = spdiagm(linspace(-1.0, 100.0, n));
julia> g = ones(1000);
julia> TR = trlib.trlib_data(H, g);
julia> trlib.trlib_solve(TR, 1.0);
julia> norm(TR.sol)
(1.0000000000000002,2.9355512148709044,-15.283315647553387)
julia> trlib.trlib_solve(TR, 0.5);
julia> norm(TR.sol), TR.lam, TR.obj
(0.5000000000000001,28.860019828697034,-11.01602177675002)
```

1.5 C Example

Using the C interface to solve a trust region subproblem is done by repeated calls to *trlib_krylov_min()*, using a reverse communication pattern.

This is explained in detail in the API doc of *trlib_krylov_min()*, here we show an example how to solve a trust region subproblem and resolve it with a different radius:

First we include the necessary headers and provide declarations for the linear algebra routines provided by BLAS and LAPACK:

```
#include <stdlib.h>
#include "math.h"
#include "string.h"
#include "trlib.h"

// blas
void daxpy_(trlib_int_t *n, trlib_flt_t *alpha, trlib_flt_t *x, trlib_int_t *incx,␣
↪trlib_flt_t *y, trlib_int_t *incy);
void dscal_(trlib_int_t *n, trlib_flt_t *alpha, trlib_flt_t *x, trlib_int_t *incx);
void dcopy_(trlib_int_t *n, trlib_flt_t *x, trlib_int_t *incx, trlib_flt_t *y, trlib_
↪int_t *incy);
trlib_flt_t dnrm2_(trlib_int_t *n, trlib_flt_t *x, trlib_int_t *incx);
trlib_flt_t ddot_(trlib_int_t *n, trlib_flt_t *x, trlib_int_t *incx, trlib_flt_t *y,␣
↪trlib_int_t *incy);
// lapack
void dgemv_(char *trans, trlib_int_t *m, trlib_int_t *n, trlib_flt_t *alpha, trlib_
↪flt_t *a, trlib_int_t *lda, trlib_flt_t *x, trlib_int_t *incx, trlib_flt_t *beta,␣
↪trlib_flt_t *y, trlib_int_t *incy);
```

Then we define a data structure to be used in the reverse communication with *trlib_krylov_min()*:

```
struct trlib_qpdata {
     trlib_int_t n;                    ///< dimension of problem
     trlib_int_t maxiter;              ///< maximum number of Krylov subspace␣
↪iterations
     trlib_int_t *iwork;               ///< integer work space
     trlib_flt_t *fwork;               ///< floating point workspace
     trlib_flt_t *gradient;            ///< gradient of QP
     trlib_int_t hotstart;             ///< flag that determines if hotstarted or not
     void (*hv_cb)(const int n, const double *, double *);
                                       ///< callback to compute hessian vector␣
↪product
     trlib_int_t iter;                 ///< iteration counter
     trlib_flt_t *g;                   ///< gradient of Krylov iteration
     trlib_flt_t *gm;                  ///< previous gradient of Krylov iteration
     trlib_flt_t *p;                   ///< direction
     trlib_flt_t *Hp;                  ///< hessian product
     trlib_flt_t *Q;                   ///< matrix with Lanczos directions
 };
```

The following function fills useful defaults in such a data structure once the problem data are known:

```
int prepare_qp(trlib_int_t n, trlib_int_t maxiter,
       const double *gradient,
       void (*hv_cb)(const int n, const double *, double *),
       struct trlib_qpdata *data) {
    data->n = n;
    data->maxiter = maxiter;
    data->gradient = (double *)gradient;
    data->hv_cb = hv_cb;
    data->hotstart = 0;
    trlib_int_t iwork_size, fwork_size, h_pointer;
    trlib_krylov_memory_size(maxiter, &iwork_size, &fwork_size, &h_pointer);
    data->iwork = malloc(iwork_size*sizeof(trlib_int_t));
    data->fwork = malloc(fwork_size*sizeof(trlib_flt_t));
    data->g = malloc(n*sizeof(double));
    data->gm = malloc(n*sizeof(double));
```

(continues on next page)

(continued from previous page)

```
    data->p = malloc(n*sizeof(double));
    data->Hp = malloc(n*sizeof(double));
    data->Q = malloc((maxiter+1)*n*sizeof(double));
    data->iter = 0;
    return 0;
}
```

This function cleans up the data after the solution process has been finished:

```
int destroy_qp(struct trlib_qpdata *data) {
    free(data->iwork);
    free(data->fwork);
    free(data->g);
    free(data->gm);
    free(data->p);
    free(data->Hp);
    free(data->Q);
    return 0;
}
```

This function does the actual solution process by answering the reverse communication requests:

```
int solve_qp(struct trlib_qpdata *data, trlib_flt_t radius, double *sol, double *lam)
↪{

    // some default settings
    trlib_int_t equality = 0;
    trlib_int_t maxlanczos = 100;
    trlib_int_t ctl_invariant = 0;
    trlib_int_t refine = 1;
    trlib_int_t verbose = 1;
    trlib_int_t unicode = 0;
    trlib_flt_t tol_rel_i = -2.0;
    trlib_flt_t tol_abs_i = 0.0;
    trlib_flt_t tol_rel_b = -3.0;
    trlib_flt_t tol_abs_b = 0.0;
    trlib_flt_t obj_lo = -1e20;
    trlib_int_t convexify = 1;
    trlib_int_t earlyterm = 1;

    trlib_int_t ret = 0;

    trlib_int_t n = data->n;
    trlib_int_t init = 0, inc = 1, itp1 = 0;
    trlib_flt_t minus = -1.0, one = 1.0, z = 0.0, half = .5;
    if(!data->hotstart) {
        init = TRLIB_CLS_INIT;
        trlib_krylov_prepare_memory(data->maxiter, data->fwork);
    }
    else { init = TRLIB_CLS_HOTSTART; }

    trlib_flt_t v_dot_g = 0.0, p_dot_Hp = 0.0, flt1, flt2, flt3;
    trlib_int_t action, ityp;

    trlib_int_t iwork_size, fwork_size, h_pointer;
    trlib_krylov_memory_size(data->maxiter, &iwork_size, &fwork_size, &h_pointer);
```

(continues on next page)

(continued from previous page)

```
    while(1) {
        ret = trlib_krylov_min(init, radius, equality, data->maxiter, maxlanczos,
                tol_rel_i, tol_abs_i, tol_rel_b, tol_abs_b,
                TRLIB_EPS*TRLIB_EPS, obj_lo, ctl_invariant, convexify, earlyterm,
                v_dot_g, v_dot_g, p_dot_Hp, data->iwork, data->fwork,
                refine, verbose, unicode, "", stdout, NULL,
                &action, &(data->iter), &ityp, &flt1, &flt2, &flt3);
        init = 0;
        switch(action) {
            case TRLIB_CLA_INIT:
                memset(sol, 0, n*sizeof(trlib_flt_t)); memset(data->gm, 0,␣
↪n*sizeof(trlib_flt_t));
                dcopy_(&n, data->gradient, &inc, data->g, &inc);
                v_dot_g = ddot_(&n, data->g, &inc, data->g, &inc);
                // p = -g
                dcopy_(&n, data->g, &inc, data->p, &inc); dscal_(&n, &minus, data->p,␣
↪&inc);
                // Hp = H*p
                data->hv_cb(n, data->p, data->Hp);
                p_dot_Hp = ddot_(&n, data->p, &inc, data->Hp, &inc);
                // Q(0:n) = g
                dcopy_(&n, data->g, &inc, data->Q, &inc);
                // Q(0:n) = g/sqrt(<g,g>)
                flt1 = 1.0/sqrt(v_dot_g); dscal_(&n, &flt1, data->Q, &inc);
                break;
            case TRLIB_CLA_RETRANSF:
                itp1 = data->iter+1;
                // s = Q_i * h_i
                dgemv_("N", &n, &itp1, &one, data->Q, &n, data->fwork+h_pointer, &inc,
↪ &z, sol, &inc);
                break;
            case TRLIB_CLA_UPDATE_STATIO:
                // s += flt1*p
                if (ityp == TRLIB_CLT_CG) { daxpy_(&n, &flt1, data->p, &inc, sol, &
↪inc); };
                break;
            case TRLIB_CLA_UPDATE_GRAD:
                if (ityp == TRLIB_CLT_CG) {
                    // Q(iter*n:(iter+1)*n) = flt2*g
                    dcopy_(&n, data->g, &inc, data->Q+(data->iter)*n, &inc);
                    dscal_(&n, &flt2, data->Q+(data->iter)*n, &inc);
                    // gm = g; g += flt1*Hp
                    dcopy_(&n, data->g, &inc, data->gm, &inc);
                    daxpy_(&n, &flt1, data->Hp, &inc, data->g, &inc);
                }
                if (ityp == TRLIB_CLT_L) {
                    // s = Hp + flt1*g + flt2*gm
                    dcopy_(&n, data->Hp, &inc, sol, &inc);
                    daxpy_(&n, &flt1, data->g, &inc, sol, &inc);
                    daxpy_(&n, &flt2, data->gm, &inc, sol, &inc);
                    // gm = flt3*g
                    dcopy_(&n, data->g, &inc, data->gm, &inc); dscal_(&n, &flt3, data-
↪>gm, &inc);
                    // g = s
                    dcopy_(&n, sol, &inc, data->g, &inc);
                }
                v_dot_g = ddot_(&n, data->g, &inc, data->g, &inc);
```

(continues on next page)

(continued from previous page)

```
                break;
            case TRLIB_CLA_UPDATE_DIR:
                if (ityp == TRLIB_CLT_CG) {
                    // p = -g + flt2 * p
                    dscal_(&n, &flt2, data->p, &inc);
                    daxpy_(&n, &minus, data->g, &inc, data->p, &inc);
                }
                if (ityp == TRLIB_CLT_L) {
                    // p = flt1*g
                    dcopy_(&n, data->g, &inc, data->p, &inc);
                    dscal_(&n, &flt1, data->p, &inc);
                }
                // Hp = H*p
                data->hv_cb(n, data->p, data->Hp);
                p_dot_Hp = ddot_(&n, data->p, &inc, data->Hp, &inc);
                if ( ityp == TRLIB_CLT_L) {
                    // Q(iter*n:(iter+1)*n) = p
                    dcopy_(&n, data->p, &inc, data->Q+(data->iter)*n, &inc);
                }
                break;
            case TRLIB_CLA_CONV_HARD:
                itp1 = data->iter+1;
                trlib_flt_t *temp = malloc(n*sizeof(trlib_flt_t));
                // temp = H*s
                data->hv_cb(n, sol, temp);
                // temp = H*s + g
                daxpy_(&n, &one, data->gradient, &inc, temp, &inc);
                // temp = H*s + g + flt1*s
                daxpy_(&n, &flt1, sol, &inc, temp, &inc);
                v_dot_g = ddot_(&n, temp, &inc, temp, &inc);
                free(temp);
                break;
            case TRLIB_CLA_NEW_KRYLOV:
                printf("Hit invariant Krylov subspace. Please implement proper␣
↪reorthogonalization!");
                break;
            case TRLIB_CLA_OBJVAL:
                trlib_flt_t *temp = malloc(n*sizeof(trlib_flt_t));
                // temp = H*s
                data->hv_cb(n, sol, temp);
                // temp = .5*H*s + g
                daxpy_(&n, &half, data->gradient, &inc, temp, &inc);
                // g_dot_g = .5 s^T H*s + g^T s
                g_dot_g = ddot_(&n, temp, &inc, sol, &inc);
                free(temp);
        }
        if( ret < 10 ) { break; }
    }
    *lam = data->fwork[7];

    if(!data->hotstart) { data->hotstart = 1; }
    return ret;
}
```

This function defines for a test example the function that computes hessian vector products:

```
/* Test the driver program to solve a 3D problem with two different radii
 */

void hessvec(const int n, const double *d, double *Hd) {
    Hd[0] = d[0] + 4.0*d[2];
    Hd[1] = 2.0*d[1];
    Hd[2] = 4.0*d[0] + 3.0*d[2];
}
```

This main routine initializes data to solve a trust region subproblem with radius 2.0 and hotstart it afterwards to resolve with radius 1.0 from the previous solution:

```
int main () {

    trlib_int_t n = 3;          // number of variables
    trlib_int_t maxiter = 10*n; // maximum number of CG iterations

    // gradient of QP
    double *g = malloc(n*sizeof(double));
    g[0] = 5.0; g[1] = 0.0; g[2] = 4.0;

    // get datastructure for QP solution and prepare it
    struct trlib_qpdata data;
    prepare_qp(n, maxiter, g, &hessvec, &data);

    // allocate memory for solution
    double *sol = malloc(n*sizeof(double));
    double lam = 0.0;

    // solve QP with trust region radius 2.0
    double radius = 2.0;
    printf("Attempting to solve trust region problem with radius %f\n", radius);
    solve_qp(&data, radius, sol, &lam);
    printf("Got lagrange multiplier %f and solution vector [%f %f %f]\n\n", lam,␣
↪sol[0], sol[1], sol[2]);

    // resolve QP with trust region radius 1.0
    radius = 1.0;
    printf("Attempting to solve trust region problem with radius %f\n", radius);
    solve_qp(&data, radius, sol, &lam);
    printf("Got lagrange multiplier %f and solution vector [%f %f %f]\n\n", lam,␣
↪sol[0], sol[1], sol[2]);

    // clean up
    destroy_qp(&data);
    free(g); free(sol);

    return 0;

}
```

1.6 CAPI trlib_krylov

1.6.1 Functions

trlib_int_t **trlib_krylov_min**(trlib_int_t *init*, trlib_flt_t *radius*, trlib_int_t *equality*, trlib_int_t *itmax*, trlib_int_t *itmax_lanczos*, trlib_flt_t *tol_rel_i*, trlib_flt_t *tol_abs_i*, trlib_flt_t *tol_rel_b*, trlib_flt_t *tol_abs_b*, trlib_flt_t *zero*, trlib_flt_t *obj_lo*, trlib_int_t *ctl_invariant*, trlib_int_t *convexify*, trlib_int_t *earlyterm*, trlib_flt_t *g_dot_g*, trlib_flt_t *v_dot_g*, trlib_flt_t *p_dot_Hp*, trlib_int_t **iwork*, trlib_flt_t **fwork*, trlib_int_t *refine*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_int_t **action*, trlib_int_t **iter*, trlib_int_t **ityp*, trlib_flt_t **flt1*, trlib_flt_t **flt2*, trlib_flt_t **flt3*)

Solves trust region subproblem: Computes minimizer to

$\min \frac{1}{2}\langle s, Hs\rangle + \langle g_0, s\rangle$ subject to the trust region constraint $\|s\|_M \leq r$, where

- H is available via matrix-vector products $v \mapsto Hv$,
- $\|s\|_M = \sqrt{\langle s, Ms\rangle}$ with M positive definite,
- M is available via matrix-vector products of its inverse, $v \mapsto M^{-1}v$.

The minimizer is a global minimizer (modulo floating point), as long as the hard case does not occur. The hard case is characterized by the fact that the eigenspace corresponding to the smallest eigenvalue is degenerate.

In that case a global minimizer in the Krylov subspaces sampled so far is returned, sampling the complete search space can be forced by setting `ctl_invariant` to *TRLIB_CLC_EXP_INV_GLO*, but if this is desired factorization based method will most likely be much more efficient.

A preconditioned Conjugate Gradient / Lanczos method is used, that possibly employs a reduction to a tridiagnoal subproblem that is solved using Moré-Sorensens method by explicitly factorizing the tridiagonal matrix, calling *trlib_tri_factor_min()*.

The method builds upon algorithm 5.1 in [Gould1999], details of the implementation can be found [Lenders2016]:

Convergence

The stopping criterion is based on the gradient of the Lagrangian. Convergence in iteration i is reported as soon as

- interior case: $\|g_{i+1}\|_{M^{-1}} \leq \max\{\texttt{tol\_abs\_i}, \eta_i \|g_0\|_{M^{-1}}\}$.
- boundary case: $\|g_{i+1}\|_{M^{-1}} \leq \max\{\texttt{tol\_abs\_b}, \eta_b \|g_0\|_{M^{-1}}\}$.
- hard case: `ctl_invariant` determines if this is the sole stopping criterion or if further invariant subspaces are going to be explored. See the documentation of this option.

Here $\eta_i = \begin{cases} \texttt{tol\_rel\_i} & \texttt{tol\_rel\_i} > 0 \\ \min\{0.5, \sqrt{\|g_{i+1}\|_{M^{-1}}}\} & \texttt{tol\_rel\_i} = -1 \\ \min\{0.5, \|g_{i+1}\|_{M^{-1}}\} & \texttt{tol\_rel\_i} = -2 \end{cases}$, $\eta_b = \begin{cases} \texttt{tol\_rel\_b} & \texttt{tol\_rel\_b} > 0 \\ \min\{0.5, \sqrt{\|g_{i+1}\|_{M^{-1}}}\} & \texttt{tol\_rel\_b} = -1 \\ \min\{0.5, \|g_{i+1}\|_{M^{-1}}\} & \texttt{tol\_rel\_b} = -2, . \\ \max\{10^{-6}, \min\{0.5, \sqrt{\|g_{i+1}\|_{M^{-1}}}\}\} & \texttt{tol\_rel\_b} = -3 \\ \max\{10^{-6}, \min\{0.5, \|g_{i+1}\|_{M^{-1}}\}\} & \texttt{tol\_rel\_b} = -4 \end{cases}$

Choice of η_i and η_b depend on the application, the author has good overall experience in unconstrained optimization with `tol_rel_i` $= -2$ and `tol_rel_b` $= -3$. Some remarks to keep in mind when choosing the tolerances:

- Choosing fixed small values for η_i and η_b, for example `tol_rel_i` $= 10^{-8}$ and `tol_rel_b` $= 10^{-6}$ lead to quick convergence in a NLP algorithm with lowest number of function evaluations but at a possible excessive amount of matrix-vector products as this also solves problems accurately far away from the solution.
- Choosing $\eta \sim O(\sqrt{\|g\|_{M^{-1}}})$ leads to superlinear convergence in a nonlinear programming algorithm.
- Choosing $\eta \sim O(\|g\|_{M^{-1}})$ leads to quadratic convergence in a nonlinear programming algorithm.
- It is questionable whether it makes sense to solve the boundary with very high accuracy, as the final solution of a nonlinear program should be interior.

Calling Scheme

This function employs a reverse communication paradigm. The functions exits whenever there is an action to be performed by the user as indicated by the `action`. The user should perform this action and continue with the other values unchanged as long as the return value is positive.

User provided storage

The user has to manage 5/6 vectors refered by the names g, g_-, v, s, p, *Hp* and a matrix Q with `itmax` columns to store the Lanczos directions p_i. The user must perform the actions on those as indicated by the return value.

In the case of trivial preconditioner, $M = I$, it always holds $v = g$ and the vector v is not necessary.

s holds the solution candidate.

Note that the action $s \leftarrow Qh$ will only sometimes be requested in the very final iteration before convergence. If memory and not computational load is an issue, you may very well save `iter`, `ityp`, `flt1`, `flt2` `flt2` instead of q_j and when $s \leftarrow Qs$ is requested simultaniously recompute the directions q_j and update the direction s once q_j has been computed as $s \leftarrow h_j q_j$ with initialization $s \leftarrow 0$.

Furthermore the user has to provide a integer and floating point workspace, details are described in `iwork` and `fwork`.

Resolves

Reentry with new radius

You can efficiently hotstart from old results if you have a new problem with *decreased* trust region radius. Just set *status* to *TRLIB_CLS_HOTSTART*. Furthermore hotstarting with increased trust region radius should be trivial as you should be able to just increase the radius and take off the computation from the previous stage. However as this is an untypical scenario, it has not been tested at all.

Reentry to get minimizer of regularized problem

You can reenter to compute the solution of a convexified trust region problem. You may be interested to this once you see the effect of bad conditioning or also as a cheap retry in a trust region algorithm before continuing to the next point after a discarded step.

In this case, call the function with *status* set to *TRLIB_CLS_HOTSTART_P*.

Reentry to get unconstrained minimizer of constant regularized problem

After a successful solve you can compute the norm of the unconstrained minimizer to the problem with regularized hessian $H + \beta M$ (β large enough such that $H + \beta M$ is positive definite) in the previously expanded Krylov subspace. You will certainly be interested in this if you want to implement the TRACE algorithm described in [Curtis2016].

In this case, call the function with *status* set to *TRLIB_CLS_HOTSTART_R* and `radius` set to β.

On exit, the `obj` field of `fwork` yields the norm of the solution vector.

If you not only want the norm but also the unconstrained minimizer, you can get it by one step of *TRLIB_CLA_RETRANSF()*. However since this is usually not the case, you are not asked to do this in the reverse communication scheme.

Reentry to find suitable TRACE regularization parameter

For the aforementioned TRACE algorithm, there is also the need to compute β such that $\sigma_\mathrm{l} \leq \frac{\beta}{\|s(\beta)\|} \leq \sigma_\mathrm{u}$, where $\|s(\beta)\|$ denotes the norm of the regularized unconstrained minimizer.

To get this, set `status` to *TRLIB_CLS_HOTSTART_T* and `radius` to an initial guess for β, `tol_rel_i` to σ_l and `tol_rel_b` to σ_u. The field `obj` of `fwork` contains the desired solution norm on exit and `flt1` is the desired regularization parameter β.

If you not only want the norm but also the unconstrained minimizer, you can get it by one step of *TRLIB_CLA_RETRANSF*. However since this is usually not the case, you are not asked to do this in the reverse communication scheme.

Reentry with new gradient

You can efficiently hotstart from old results if you have a new problem with changed g_0 where the previous sampled Krylov subspace is large enough to contain the solution to the new problem, convergence will *not* be checked:

- get pointer `gt` to negative gradient of tridiagonal problem in *fwork* with *trlib_krylov_gt()*
- store `gt[0]` and overwrite `gt` $\leftarrow -Q_i^T$`grad`
- set `status` to *TRLIB_CLS_HOTSTART_G* and start reverse communication process
- to reset data for new problem make sure that you restore `gt[0]` and set `gt[1:] = 0` for those elements previously overwritten

Hard case

If you want to investigate the problem also if the hard case, you can either sample through the invariant subspaces (`ctl_invariant` set to *TRLIB_CLC_EXP_INV_GLO*) or solve the problem with a gradient for which the hard does not occur and then hotstart with the actual gradient.

Parameters

- **init** (trlib_int_t, input) – set to *TRLIB_CLS_INIT* on first call, to *TRLIB_CLS_HOTSTART* on hotstart with smaller radius and otherwise to `0`.
- **radius** (trlib_flt_t, input) – trust region radius
- **equality** (trlib_int_t, input) – set to `1` if trust region constraint should be enforced as equality
- **itmax** (trlib_int_t, input) – maximum number of iterations
- **itmax_lanczos** (trlib_int_t, input) – maximum number of Lanczos type iterations. You are strongly advised to set this to a small number (say 25) unless you know better. Keep in mind that Lanczos type iteration are only performed when curvature information is flat and Lanczos may amplify rounding errors without reorthogonalization. If you allow a lot of Lanczos type iterations consider reorthogonalizing the new direction against the vector storage.
- **tol_rel_i** (trlib_flt_t, input) – relative stopping tolerance for interior solution
- **tol_abs_i** (trlib_flt_t, input) – absolute stopping tolerance for interior solution
- **tol_rel_b** (trlib_flt_t, input) – relative stopping tolerance for boundary solution

- **tol_abs_b** (trlib_flt_t, input) – absolute stopping tolerance for boundary solution
- **zero** (trlib_flt_t, input) – threshold that determines when $\langle p, Hp \rangle$ is considered to be zero and thus control eventual Lanczos switch
- **obj_lo** (trlib_flt_t, input) – lower bound on objective, returns if a point is found with function value $\leq$ obj_lo. Set to a positive value to ignore this bound.
- **ctl_invariant** (trlib_int_t, input) –
  - set to *TRLIB_CLC_NO_EXP_INV* if you want to search only in the first invariant Krylov subspace
  - set to *TRLIB_CLC_EXP_INV_LOC* if you want to continue to expand the Krylov subspaces but terminate if there is convergence indication in the subspaces sampled so far.
  - set to *TRLIB_CLC_EXP_INV_GLO* if you want to continue to expand the Krylov subspaces even in the case of convergence to a local minimizer within in the subspaces sampled so far. Reverse communication continues as long as ctl_invariant is set to *TRLIB_CLC_EXP_INV_GLO*, so you should reset ctl_invariant to either *TRLIB_CLC_EXP_INV_LOC* or *TRLIB_CLC_EXP_INV_LOC* if there is no reason to continue, for example because you cannot find a new nonzero random vector orthogonal to the sampled directions if action is *TRLIB_CLA_NEW_KRYLOV*.
- **convexify** (trlib_int_t, input) – set to *1* if you like to monitor if the tridiagonal solution and the backtransformed solution match and if not resolve with a convexified problem, else set to *0*
- **earlyterm** (trlib_int_t, input) – set to *1* if you like to terminate in the boundary case if it unlikely that much progress will be made fast but no convergence is reached, else set to *0*
- **g_dot_g** (trlib_flt_t, input) – dot product $\langle g, g \rangle$
- **v_dot_g** (trlib_flt_t, input) – dot product $\langle v, g \rangle$
- **p_dot_Hp** (trlib_flt_t, input) – dot product $\langle p, Hp \rangle$
- **iwork** (trlib_int_t, input/output) – integer workspace for problem, internal memory layout described in table below
  - on first call, provide allocated memory for iwork_size entries provided by *trlib_krylov_memory_size()*
  - leave untouched between iterations

start	end(exclusive)	description
0	1	internal status flag
1	2	iteration counter
2	3	flag that indicates if H is positive definite in sampled Krylov subspace
3	4	flag that indicates if interior solution is expected
4	5	flag that indicates if warmstart information to `leftmost` is available
5	6	index to block with smallest leftmost
6	7	flag that indicates if warmstart information to λ_0 is available
7	8	flag that indicates if warmstart information to λ is available
8	9	iteration in which switched to Lanczos iteration, `-1`: no switch occured
9	10	return code from *`trlib_tri_factor_min()`*
10	11	`sub_fail` exit code from subrotines called in *`trlib_tri_factor_min()`*
11	12	number of newton iterations needed in *`trlib_tri_factor_min()`*
12	13	last iteration in which a headline has been printed
13	14	kind of iteration headline that has been printed last
14	15	status variable if convexified version is going to be resolved
15	16	number of irreducible blocks
16	17 + `itmax`	decomposition into irredecubile block, `irblk` for *`trlib_tri_factor_min()`*

- **`fwork`** (`trlib_flt_t`, input/output) – floating point workspace for problem, internal memory layout described in table below
  - on very first call, provide allocated memory for at least `fwork_size` entries that has been initialized using `trlib_prepare_memory()`, `fwork_size` can be obtained by calling *`trlib_krylov_memory_size()`*
  - can be used for different problem instances with matching dimensions and itmax without reinitialization
  - leave untouched between iterations

start	end (exclusive)	description
0	1	stopping tolerance in case of interior solution
1	2	stopping tolerance in case of boundary solution
2	3	dot product $\langle v, g \rangle$ in current iteration
3	4	dot product $\langle p, Hp \rangle$ in current iteration
4	5	ratio between projected CG gradient and Lanczos direction in current iteration
5	6	ratio between projected CG gradient and Lanczos direction in previous iteration
6	7	Lagrange multiplier λ_0 for trust region constraint
7	8	Lagrange multiplier λ for trust region constraint
8	9	objective function value in current iterate
9	10	$\langle s_i, Mp_i \rangle$
10	11	$\langle p_i, Mp_i \rangle$
11	12	$\langle s_i, Ms_i \rangle$
12	13	σ_i
13	14	max Rayleigh quotient, $\max_i \frac{\langle p, Hp \rangle}{\langle p, Mp \rangle}$

Table 1 – continued from previous page

start	end (exclusive)	description
14	15	min Rayleigh quotient, $\min_i \frac{\langle p,Hp\rangle}{\langle p,Mp\rangle}$
15	16 + itmax	$\alpha_i, i \geq 0$, step length CG
16 + itmax	17 + 2 itmax	$\beta_i, i \geq 0$, step update factor CG
17 + 2 itmax	18 + 3 itmax	neglin for *trlib_tri_factor_min()*, just given by $-\gamma_0 e_1$
18 + 3 itmax	19 + 4 itmax	solution h_0 of tridiagonal subproblem provided as sol by *trlib_tri_factor_min*
19 + 4 itmax	20 + 5 itmax	solution h of tridiagonal subproblem provided as sol by *trlib_tri_factor_min(*
20 + 5 itmax	21 + 6 itmax	$\delta_i, i \geq 0$, curvature in Lanczos, diagonal of T in Lanczos tridiagonalization process
21 + 6 itmax	22 + 7 itmax	diagonal of Cholesky of $T_0 + \lambda_0 I$
22 + 7 itmax	23 + 8 itmax	diagonal of Cholesky of $T + \lambda I$
23 + 8 itmax	23 + 9 itmax	$\gamma_i, i \geq 1$, norm of gradients in Lanczos; provides offdiagonal of T in Lanczos tridiagonal
23 + 9 itmax	23 + 10 itmax	offdiagonal of Cholesky factorization of $T_0 + \lambda_0 I$
23 + 10 itmax	23 + 11 itmax	offdiagonal of Cholesky factorization of $T + \lambda I$
23 + 11 itmax	24 + 12 itmax	ones for *trlib_tri_factor_min()* and *trlib_eigen_inverse()*
24 + 12 itmax	25 + 13 itmax	leftmost for *trlib_tri_factor_min()*
25 + 13 itmax	26 + 14 itmax	regdiag for *trlib_tri_factor_regularize_posdef()*
26 + 14 itmax	27 + 15 itmax	history of convergence criteria values
27 + 15 itmax	27 + 15 itmax + fmz	fwork for *trlib_tri_factor_min()*, fmz is given by *trlib_tri_factor_m*

- **refine** (trlib_int_t, input) – set to 1 if iterative refinement should be used on solving linear systems, otherwise to 0
- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, all values are multiples of nanoseconds, provide zero allocated memory of length *trlib_krylov_timing_size()*

block	description
0	total duration
1	timing of *trlib_tri_factor_min()*

- **action** (trlib_int_t, output) – The user should perform the following action depending on action and ityp on the vectors he manages, see the table below. The table makes use of the notation explained in the *User provided storage* above and the following:
  - i: iter
  - q_j: j-th column of Q
  - Q_i: matrix consisting of the first $i+1$ columns of Q, $Q_i = (q_0, \ldots, q_i)$
  - h_i: vector of length $i+1$ stored in fwork at start position h_pointer provided by *trlib_krylov_memory_size()*
  - $p \leftarrow \perp_M Q_j$: optionally M-reorthonormalize p against Q_j
  - $g \leftarrow$ rand $\perp Q_j$ find nonzero random vector that is orthogonal to Q_j

– Note that *TRLIB_CLA_NEW_KRYLOV* is unlikely and only occurs on problems that employ the hard case and only if `ctl_invariant` $\neq$ *TRLIB_CLC_NO_EXP_INV*. If you want to safe yourself from the trouble implementing this and are confident that you don't need to expand several invariant Krylov subspaces, just ensure that `ctl_invariant` is set to *TRLIB_CLC_NO_EXP_INV*.

action	ityp	command
TRLIB_CLA_TERMINATE	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	do nothing
TRLIB_CLA_REFINE_SOL	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	$s \leftarrow Q_i h_i$
TRLIB_CLA_INIT	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	$s \leftarrow 0$, $g \leftarrow g_0$, $g_- \leftarrow 0$, $v \leftarrow M^{-1}g$, $p \leftarrow -v$, $Hp \leftarrow Hp$, `g_dot_g` $\leftarrow \langle g, g\rangle$, `v_dot_g` $\leftarrow \langle v, g\rangle$, `p_dot_Hp` $\leftarrow \langle p, Hp\rangle$, $q_0 \leftarrow \frac{1}{\sqrt{\texttt{v\_dot\_g}}} v$
TRLIB_CLA_UPDATE_SOL	*TRLIB_CLT_CG*	$s \leftarrow s + \texttt{flt1}\, p$
TRLIB_CLA_UPDATE_SOL	*TRLIB_CLT_L*	do nothing
TRLIB_CLA_UPDATE_GRAD	*TRLIB_CLT_CG*	$q_i \leftarrow \texttt{flt2}\, v$, $g_- \leftarrow g$, $g \leftarrow g + \texttt{flt1}\, Hp$, $v \leftarrow M^{-1}g$, `g_dot_g` $\leftarrow \langle g, g\rangle$, `v_dot_g` $\leftarrow \langle v, g\rangle$
TRLIB_CLA_UPDATE_GRAD	*TRLIB_CLT_L*	$s \leftarrow Hp + \texttt{flt1}\, g + \texttt{flt2}\, g_-$, $g_- \leftarrow \texttt{flt3}\, g$, $g \leftarrow s$, $v \leftarrow M^{-1}g$, `v_dot_g` $\leftarrow \langle v, g\rangle$
TRLIB_CLA_UPDATE_DIR	*TRLIB_CLT_CG*	$p \leftarrow \texttt{flt1}\, v + \texttt{flt2}\, p$ with $\texttt{flt1} = -1$, $Hp \leftarrow Hp$, `p_dot_Hp` $\leftarrow \langle p, Hp\rangle$
TRLIB_CLA_UPDATE_DIR	*TRLIB_CLT_L*	$p \leftarrow \texttt{flt1}\, v + \texttt{flt2}\, p$ with $\texttt{flt2} = 0$, $p \leftarrow \perp_M Q_{i-1}$, $Hp \leftarrow Hp$, `p_dot_Hp` $\leftarrow \langle p, Hp\rangle$, $q_i \leftarrow p$
TRLIB_CLA_NEW_KRYLOV	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	$g \leftarrow$ `rand` $\perp Q_{i-1}$, $g_- \leftarrow 0$, $v \leftarrow M^{-1}g$, `v_dot_g` $\leftarrow \langle v, g\rangle$, $p \leftarrow \frac{1}{\sqrt{\texttt{v\_dot\_g}}} v$, `p_dot_Hp` $\leftarrow \langle p, Hp\rangle$, $q_{i+1} \leftarrow p$
TRLIB_CLA_CONV_HARD	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	`v_dot_g` $\leftarrow \langle Hs + g_0 + \texttt{flt1}\, Ms, M^{-1}(Hs + g_0) + \texttt{flt1}\, s\rangle$
TRLIB_CLA_OBTAIN_PV	*TRLIB_CLT_CG*, *TRLIB_CLT_L*	`g_dot_g` $\leftarrow \frac{1}{2}\langle s, Hs\rangle + \langle g, s\rangle$

- **iter** (`trlib_int_t`, output) – iteration counter to tell user position in vector storage
- **ityp** (`trlib_int_t`, output) – iteration type, see `action`
- **flt1** (`trlib_flt_t`, output) – floating point value that user needs for actions
- **flt2** (`trlib_flt_t`, output) – floating point value that user needs for actions
- **flt3** (`trlib_flt_t`, output) – floating point value that user needs for actions

Returns

status flag with following meaning:

- *TRLIB_CLR_CONTINUE* no convergence yet, continue in reverse communication
- *TRLIB_CLR_CONV_BOUND* successful exit with converged solution on boundary, end reverse communication process
- *TRLIB_CLR_CONV_INTERIOR* successful exit with converged interior solution, end reverse communication process
- *TRLIB_CLR_APPROX_HARD* succesful exit with approximate solution, hard case occured, end reverse communication process

- *TRLIB_CLR_NEWTON_BREAK* exit with breakdown in Newton iteration in *trlib_tri_factor_min()*, most likely converged to boundary solution
- *TRLIB_TTR_HARD_INIT_LAM* hard case encountered without being able to find suitable initial λ for Newton iteration, returned approximate stationary point that maybe suboptimal
- *TRLIB_CLR_ITMAX* iteration limit exceeded, end reverse communication process
- *TRLIB_CLR_UNBDBEL* problem seems to be unbounded from below, end reverse communication process
- *TRLIB_CLR_FAIL_FACTOR* failure on factorization, end reverse communication process
- *TRLIB_CLR_FAIL_LINSOLVE* failure on backsolve, end reverse communication process
- *TRLIB_CLR_FAIL_NUMERIC* failure as one of the values `v_dot_g` or `p_dot_Hp` are not a floating point number
- *TRLIB_CLR_UNLIKE_CONV* exit early as convergence seems to be unlikely
- *TRLIB_CLR_PCINDEF* preconditioner apperas to be indefinite, end reverse communication process
- *TRLIB_CLR_UNEXPECT_INT* unexpected interior solution found, expected boundary solution, end reverse communication process
- *TRLIB_CLR_FAIL_TTR* failure occured in *trlib_tri_factor_min()*, check `iwork[7]` and `iwork[8]` for details
- *TRLIB_CLR_FAIL_HARD* failure due to occurence of hard case: invariant subspace encountered without local convergence and request for early termination without exploring further invariant subspaces

Return type `trlib_int_t`

trlib_int_t **trlib_krylov_prepare_memory**(trlib_int_t *itmax*, trlib_flt_t **fwork*)

Prepares floating point workspace for :c:func::*trlib_krylov_min*

Initializes floating point workspace `fwork` for *trlib_krylov_min()*

Parameters

- **itmax** (`trlib_int_t`, input) – maximum number of iterations
- **fwork** (`trlib_flt_t`, input/output) – floating point workspace to be used by *trlib_krylov_min()*

Returns `0`

Return type `trlib_int_t`

trlib_int_t **trlib_krylov_memory_size**(trlib_int_t *itmax*, trlib_int_t **iwork_size*, trlib_int_t **fwork_size*, trlib_int_t **h_pointer*)

Gives information on memory that has to be allocated for :c:func::*trlib_krylov_min*

Parameters

- **itmax** (`trlib_int_t`, input) – maximum number of iterations
- **iwork_size** (`trlib_int_t`, output) – size of integer workspace iwork that has to be allocated for *trlib_krylov_min()*
- **fwork_size** (`trlib_int_t`, output) – size of floating point workspace fwork that has to be allocated for *trlib_krylov_min()*

- **h_pointer** (trlib_int_t, output) – start index of vector h that has to be used in reverse communication in action *TRLIB_CLA_RETRANSF*

Returns 0

Return type trlib_int_t

trlib_int_t **trlib_krylov_timing_size** (void)
size that has to be allocated for timing in *trlib_krylov_min()*

Returns 0

Return type trlib_int_t

trlib_int_t **trlib_krylov_gt** (trlib_int_t *itmax*, trlib_int_t **gt_pointer*)
Gives pointer to negative gradient of tridiagonal problem

Parameters

- **itmax** (trlib_int_t, input) – itmax maximum number of iterations
- **gt_pointer** (trlib_int_t, output) – pointer to negative gradient of tridiagonal subproblem

Returns 0

Return type trlib_int_t

1.6.2 Definitions

TRLIB_CLR_CONV_BOUND
0

TRLIB_CLR_CONV_INTERIOR
1

TRLIB_CLR_APPROX_HARD
2

TRLIB_CLR_NEWTON_BREAK
3

TRLIB_CLR_HARD_INIT_LAM
4

TRLIB_CLR_FAIL_HARD
5

TRLIB_CLR_UNBDBEL
6

TRLIB_CLR_UNLIKE_CONV
7

TRLIB_CLR_CONTINUE
10

TRLIB_CLR_ITMAX
-1

TRLIB_CLR_FAIL_FACTOR
-3

TRLIB_CLR_FAIL_LINSOLVE
-4

TRLIB_CLR_FAIL_NUMERIC
-5

TRLIB_CLR_FAIL_TTR
-7

TRLIB_CLR_PCINDEF
-8

TRLIB_CLR_UNEXPECT_INT
-9

TRLIB_CLT_CG
1

TRLIB_CLT_L
2

TRLIB_CLA_TRIVIAL
0

TRLIB_CLA_INIT
1

TRLIB_CLA_RETRANSF
2

TRLIB_CLA_UPDATE_STATIO
3

TRLIB_CLA_UPDATE_GRAD
4

TRLIB_CLA_UPDATE_DIR
5

TRLIB_CLA_NEW_KRYLOV
6

TRLIB_CLA_CONV_HARD
7

TRLIB_CLA_OBJVAL
8

TRLIB_CLS_INIT
1

TRLIB_CLS_HOTSTART
2

TRLIB_CLS_HOTSTART_G
3

TRLIB_CLS_HOTSTART_P
4

TRLIB_CLS_HOTSTART_R
5

TRLIB_CLS_HOTSTART_T
6

TRLIB_CLS_VEC_INIT
7

TRLIB_CLS_CG_NEW_ITER
8

TRLIB_CLS_CG_UPDATE_S
9

TRLIB_CLS_CG_UPDATE_GV
10

TRLIB_CLS_CG_UPDATE_P
11

TRLIB_CLS_LANCZOS_SWT
12

TRLIB_CLS_L_UPDATE_P
13

TRLIB_CLS_L_CMP_CONV
14

TRLIB_CLS_L_CMP_CONV_RT
15

TRLIB_CLS_L_CHK_CONV
16

TRLIB_CLS_L_NEW_ITER
17

TRLIB_CLS_CG_IF_IRBLK_P
18

TRLIB_CLS_CG_IF_IRBLK_C
19

TRLIB_CLS_CG_IF_IRBLK_N
20

TRLIB_CLC_NO_EXP_INV
0

TRLIB_CLC_EXP_INV_LOC
1

TRLIB_CLC_EXP_INV_GLO
2

TRLIB_CLT_CG_INT
0

TRLIB_CLT_CG_BOUND
1

TRLIB_CLT_LANCZOS
2

TRLIB_CLT_HOTSTART
3

1.7 CAPI trlib_tri_factor

1.7.1 Functions

trlib_int_t **trlib_tri_factor_min**(trlib_int_t *nirblk*, trlib_int_t **irblk*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_flt_t **neglin*, trlib_flt_t *radius*, trlib_int_t *itmax*, trlib_flt_t *tol_rel*, trlib_flt_t *tol_newton_tiny*, trlib_int_t *pos_def*, trlib_int_t *equality*, trlib_int_t **warm0*, trlib_flt_t **lam0*, trlib_int_t **warm*, trlib_flt_t **lam*, trlib_int_t **warm_leftmost*, trlib_int_t **ileftmost*, trlib_flt_t **leftmost*, trlib_int_t **warm_fac0*, trlib_flt_t **diag_fac0*, trlib_flt_t **offdiag_fac0*, trlib_int_t **warm_fac*, trlib_flt_t **diag_fac*, trlib_flt_t **offdiag_fac*, trlib_flt_t **sol0*, trlib_flt_t **sol*, trlib_flt_t **ones*, trlib_flt_t **fwork*, trlib_int_t *refine*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_flt_t **obj*, trlib_int_t **iter_newton*, trlib_int_t **sub_fail*)

Solves tridiagonal trust region subproblem

Computes minimizer to

$\min \frac{1}{2}\langle h, Th\rangle + \langle g, h\rangle$ subject to the trust region constraint $\|h\| \leq r$, where $T \in \mathbb{R}^{n\times n}$ is symmetric tridiagonal.

Let $T = \begin{pmatrix} T_1 & & \\ & \ddots & \\ & & T_\ell \end{pmatrix}$ be composed into irreducible blocks T_i.

The minimizer is a global minimizer (modulo floating point).

The algorithm is the Moré-Sorensen-Method as decribed as Algorithm 5.2 in [Gould1999].

Convergence

Exit with success is reported in several cases:

- interior solution: the stationary point $Th = -g$ is suitable, iterative refinement is used in the solution of this system.
- hard case: the smallest eigenvalue $-\theta$ was found to be degenerate and $h = v + \alpha w$ is returned with v solution of $(T + \theta I)v = -g$, w eigenvector corresponding to $-\theta$ and α chosen to satisfy the trust region constraint and being a minimizer.
- boundary and hard case: convergence in Newton iteration is reported if $\|h(\lambda)\| - r \leq$ `tol_rel` r with $(T + \lambda I)h(\lambda) = -g$, Newton breakdown reported if $|d\lambda| \leq$ `macheps` $|\lambda|$.

Parameters

- **nirblk** (`trlib_int_t`, input) – number of irreducible blocks ℓ, ensure $\ell > 0$
- **irblk** (`trlib_int_t`, input) – pointer to indices of irreducible blocks, length `nirblk+1`:
  - `irblk[i]` is the start index of block i in `diag` and `offdiag`
  - `irblk[i+1] - 1` is the stop index of block i
  - `irblk[i+1] - irred[i]` the dimension n_ℓ of block i

  - `irblk[nirred]` the dimension of T

- **diag** (`trlib_flt_t`, input) – pointer to array holding diagonal of T, length `irblk[nirblk]`
- **offdiag** (`trlib_flt_t`, input) – pointer to array holding offdiagonal of T, length `irblk[nirblk]`
- **neglin** (`trlib_flt_t`, input) – pointer to array holding $-g$, length `n`
- **radius** (`trlib_flt_t`, input) – trust region constraint radius r
- **itmax** (`trlib_int_t`, input) – maximum number of Newton iterations
- **tol_rel** (`trlib_flt_t`, input) – relative stopping tolerance for residual in Newton iteration for `lam`, good default may be `macheps` (*TRLIB_EPS*)
- **tol_newton_tiny** (`trlib_flt_t`, input) – stopping tolerance for step in Newton iteration for `lam`, good default may be $10\texttt{macheps}^{.75}$
- **pos_def** (`trlib_int_t`, input) – set `1` if you know T to be positive definite, otherwise `0`
- **equality** (`trlib_int_t`, input) – set `1` if you want to enfore trust region constraint as equality, otherwise `0`
- **warm0** (`trlib_int_t`, input/output) – set `1` if you provide a valid value in `lam0`, otherwise `0`
- **lam0** (`trlib_flt_t`, input/output) – Lagrange multiplier such that $T_0 + \texttt{lam0}I$ positive definite and $(T_0 + \texttt{lam0}I)\texttt{sol0} = \texttt{neglin}$ - on entry: estimate suitable for warmstart - on exit: computed multiplier
- **warm** (`trlib_int_t`, input/output) – set `1` if you provide a valid value in `lam`, otherwise `0`
- **lam** (`trlib_flt_t`, input/output) – Lagrange multiplier such that $T + \texttt{lam}I$ positive definite and $(T + \texttt{lam}I)\texttt{sol} = \texttt{neglin}$ - on entry: estimate suitable for warmstart - on exit: computed multiplier
- **warm_leftmost** (`trlib_int_t`, input/output) –
  - on entry: set `1` if you provide a valid value in leftmost section in `fwork`
  - on exit: `1` if leftmost section in `fwork` holds valid exit value, otherwise `0`
- **ileftmost** (`trlib_int_t`, input/output) – index to block with smallest leftmost eigenvalue
- **leftmost** (`trlib_flt_t`, input/output) – array holding leftmost eigenvalues of blocks
- **warm_fac0** (`trlib_int_t`, input/output) – set `1` if you provide a valid factoricization in `diag_fac0`, `offdiag_fac0`
- **diag_fac0** (`trlib_flt_t`, input/output) – pointer to array holding diagonal of Cholesky factorization of $T_0 + \texttt{lam0}I$, length `irblk[1]` - on entry: factorization corresponding to provided estimation `lam0` on entry - on exit: factorization corresponding to computed multiplier `lam0`
- **offdiag_fac0** (`trlib_flt_t`, input/output) – pointer to array holding offdiagonal of Cholesky factorization of $T_0 + \texttt{lam0}I$, length `irblk[1]-1` - on entry: factorization corresponding to provided estimation `lam0` on entry - on exit: factorization corresponding to computed multiplier `lam0`

- **warm_fac** (trlib_int_t, input/output) – set 1 if you provide a valid factoricization in diag_fac, offdiag_fac
- **diag_fac** (trlib_flt_t, input/output) – pointer to array of length n that holds the following: - let $j =$ ileftmost and $\theta = -$leftmost[ileftmost] - on position $0, \ldots,$ irblk[1]: diagonal of factorization of $T_0 + \theta I$ - other positions have to be allocated
- **offdiag_fac** (trlib_flt_t, input/output) – pointer to array of length n-1 that holds the following: - let $j =$ ileftmost and $\theta = -$leftmost[ileftmost] - on position $0, \ldots,$ irblk[1] $- 1$: offdiagonal of factorization of $T_0 + \theta I$ - other positions have to be allocated
- **sol0** (trlib_flt_t, input/output) – pointer to array holding solution, length irblk[1]
- **sol** (trlib_flt_t, input/output) – pointer to array holding solution, length n
- **ones** (trlib_flt_t, input) – array with every value 1.0, length n
- **fwork** (trlib_flt_t, input/output) – floating point workspace, must be allocated memory on input of size *trlib_tri_factor_memory_size()* (call with argument n) and can be discarded on output, memory layout:

start	end (excl)	description
0	n	auxiliary vector
n	2 n	holds diagonal of $T + \lambda I$
2 n	4 n	workspace for iterative refinement

- **refine** (trlib_int_t, input) – set to 1 if iterative refinement should be used on solving linear systems, otherwise to 0
- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_tri_timing_size()*

block	description
0	total duration
1	timing of linear algebra calls
2	timing from *trlib_leftmost_irreducible()*
3	timing from *trlib_eigen_inverse()*

- **obj** (trlib_flt_t, output) – objective function value at solution point
- **iter_newton** (trlib_int_t, output) – number of Newton iterations
- **sub_fail** (trlib_int_t, output) – status code of subroutine if failure occured in subroutines called

Returns

status

- *TRLIB_TTR_CONV_BOUND* success with solution on boundary

- *TRLIB_TTR_CONV_INTERIOR* success with interior solution
- *TRLIB_TTR_HARD* success, but hard case encountered and solution may be approximate
- *TRLIB_TTR_NEWTON_BREAK* most likely success with accurate result; premature end of Newton iteration due to tiny step
- *TRLIB_TTR_HARD_INIT_LAM* hard case encountered without being able to find suitable initial λ for Newton iteration, returned approximate stationary point that maybe suboptimal
- *TRLIB_TTR_ITMAX* iteration limit exceeded
- *TRLIB_TTR_FAIL_FACTOR* failure on matrix factorization
- *TRLIB_TTR_FAIL_LINSOLVE* failure on backsolve
- *TRLIB_TTR_FAIL_EIG* failure on eigenvalue computation. status code of *trlib_eigen_inverse()* in `sub_fail`
- *TRLIB_TTR_FAIL_LM* failure on leftmost eigenvalue computation. status code of *trlib_leftmost_irreducible()* in `sub_fail`

Return type `trlib_int_t`

trlib_int_t **trlib_tri_factor_regularized_umin** (trlib_int_t *n*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_flt_t **neglin*, trlib_flt_t *lam*, trlib_flt_t **sol*, trlib_flt_t **ones*, trlib_flt_t **fwork*, trlib_int_t *refine*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_flt_t **norm_sol*, trlib_int_t **sub_fail*)

Computes minimizer of regularized unconstrained problem

Computes minimizer of

$\min \frac{1}{2}\langle h, (T + \lambda I)h\rangle + \langle g, h\rangle$, where $T \in \mathbb{R}^{n\times n}$ is symmetric tridiagonal and λ such that $T + \lambda I$ is spd.

Parameters

- **n** (`trlib_int_t`, input) – dimension, ensure $n > 0$
- **diag** (`trlib_flt_t`, input) – pointer to array holding diagonal of T, length `n`
- **offdiag** (`trlib_flt_t`, input) – pointer to array holding offdiagonal of T, length `n-1`
- **neglin** (`trlib_flt_t`, input) – pointer to array holding $-g$, length `n`
- **lam** (`trlib_flt_t`, input) – regularization parameter λ
- **sol** (`trlib_flt_t`, input/output) – pointer to array holding solution, length `n`
- **ones** (`trlib_flt_t`, input) – array with every value `1.0`, length `n`
- **fwork** (`trlib_flt_t`, input/output) – floating point workspace, must be allocated memory on input of size *trlib_tri_factor_memory_size()* (call with argument `n`) and can be discarded on output, memory layout:

start	end (excl)	description
0	`n`	holds diagonal of $T + \lambda I$
`n`	2 `n`	holds diagonal of factorization $T + \lambda I$
2 `n`	3 `n`	holds offdiagonal of factorization $T + \lambda I$
3 `n`	5 `n`	workspace for iterative refinement

- **refine** (`trlib_int_t`, input) – set to `1` if iterative refinement should be used on solving linear systems, otherwise to `0`

- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_tri_timing_size()*

block	description
0	total duration
1	timing of linear algebra calls

- **norm_sol** (trlib_flt_t, output) – norm of solution fector
- **sub_fail** (trlib_int_t, output) – status code of subroutine if failure occured in subroutines called

Returns

status

- *TRLIB_TTR_CONV_INTERIOR* success with interior solution
- *TRLIB_TTR_FAIL_FACTOR* failure on matrix factorization
- *TRLIB_TTR_FAIL_LINSOLVE* failure on backsolve

Return type trlib_int_t

trlib_int_t **trlib_tri_factor_get_regularization**(trlib_int_t *n*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_flt_t **neglin*, trlib_flt_t **lam*, trlib_flt_t *sigma*, trlib_flt_t *sigma_l*, trlib_flt_t *sigma_u*, trlib_flt_t **sol*, trlib_flt_t **ones*, trlib_flt_t **fwork*, trlib_int_t *refine*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_flt_t **norm_sol*, trlib_int_t **sub_fail*)

Computes regularization parameter needed in trace

Let $s(\lambda)$ be solution of $(T + \lambda I)s(\lambda) + g = 0$, where $T \in \mathbb{R}^{n \times n}$ is symmetric tridiagonal and λ such that $T + \lambda I$ is spd.

Then find λ with $\sigma_l \leq \frac{\lambda}{s(\lambda)} \leq \sigma_u$.

Parameters

- **n** (trlib_int_t, input) – dimension, ensure $n > 0$
- **diag** (trlib_flt_t, input) – pointer to array holding diagonal of T, length n
- **offdiag** (trlib_flt_t, input) – pointer to array holding offdiagonal of T, length n-1
- **neglin** (trlib_flt_t, input) – pointer to array holding $-g$, length n
- **lam** (trlib_flt_t, input/output) – regularization parameter λ, on input initial guess, on exit desired parameter
- **sigma** (trlib_flt_t, input) – value that is used in root finding, e.g. $\frac{1}{2}(\sigma_l + \sigma_u)$

- **sigma_l** (trlib_flt_t, input) – lower bound
- **sigma_u** (trlib_flt_t, input) – upper bound
- **sol** (trlib_flt_t, input/output) – pointer to array holding solution, length n
- **ones** (trlib_flt_t, input) – array with every value 1.0, length n
- **fwork** (trlib_flt_t, input/output) – floating point workspace, must be allocated memory on input of size *trlib_tri_factor_memory_size()* (call with argument n) and can be discarded on output, memory layout:

start	end (excl)	description
0	n	holds diagonal of $T + \lambda I$
n	2 n	holds diagonal of factorization $T + \lambda I$
2 n	3 n	holds offdiagonal of factorization $T + \lambda I$
3 n	5 n	workspace for iterative refinement
5 n	6 n	auxiliary vector

- **refine** (trlib_int_t, input) – set to 1 if iterative refinement should be used on solving linear systems, otherwise to 0
- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_tri_timing_size()*

block	description
0	total duration
1	timing of linear algebra calls

- **norm_sol** (trlib_flt_t, output) – norm of solution fector
- **sub_fail** (trlib_int_t, output) – status code of subroutine if failure occured in subroutines called

Returns

status

- *TRLIB_TTR_CONV_INTERIOR* success with interior solution
- *TRLIB_TTR_FAIL_FACTOR* failure on matrix factorization
- *TRLIB_TTR_FAIL_LINSOLVE* failure on backsolve

Return type trlib_int_t

trlib_int_t **trlib_tri_factor_regularize_posdef**(trlib_int_t *n*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_flt_t *tol_away*, trlib_flt_t *security_step*, trlib_flt_t **regdiag*)

Compute diagonal regularization to make tridiagonal matrix positive definite

Parameters

- **n** (trlib_int_t, input) – dimension, ensure $n > 0$
- **diag** (trlib_flt_t, input) – pointer to array holding diagonal of T, length n
- **offdiag** (trlib_flt_t, input) – pointer to array holding offdiagonal of T, length n-1
- **tol_away** (trlib_flt_t, input) – tolerance that diagonal entries in factorization should be away from zero, relative to previous entry. Good default 10^{-12}.
- **security_step** (trlib_flt_t, input) – factor greater 1.0 that defines a margin to get away from zero in the step taken. Good default 2.0.
- **regdiag** (trlib_flt_t, input/output) – pointer to array holding regularization term, length n

Returns 0

Return type trlib_int_t

trlib_int_t **trlib_tri_factor_memory_size**(trlib_int_t *n*)
Gives information on memory that has to be allocated for *trlib_tri_factor_min()*

Parameters

- **n** (trlib_int_t, input) – dimension, ensure $n > 0$
- **fwork_size** (trlib_flt_t, output) – size of floating point workspace fwork that has to be allocated for *trlib_tri_factor_min()*

Returns 0

Return type trlib_int_t

trlib_int_t **trlib_tri_timing_size**(void)
size that has to be allocated for timing in *trlib_tri_factor_min()*

1.7.2 Definitions

TRLIB_TTR_CONV_BOUND
0

TRLIB_TTR_CONV_INTERIOR
1

TRLIB_TTR_HARD
2

TRLIB_TTR_NEWTON_BREAK
3

TRLIB_TTR_HARD_INIT_LAM
4

TRLIB_TTR_ITMAX
-1

TRLIB_TTR_FAIL_FACTOR
-2

TRLIB_TTR_FAIL_LINSOLVE
-3

TRLIB_TTR_FAIL_EIG
-4

TRLIB_TTR_FAIL_LM
-5

TRLIB_TTR_FAIL_HARD
-10

1.8 CAPI trlib_leftmost

1.8.1 Functions

trlib_int_t **trlib_leftmost** (trlib_int_t *nirblk*, trlib_int_t **irblk*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_int_t *warm*, trlib_flt_t *leftmost_minor*, trlib_int_t *itmax*, trlib_flt_t *tol_abs*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_int_t **ileftmost*, trlib_flt_t **leftmost*)

Computes smallest eigenvalue of symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, using a iteration based on last-pivot function of Parlett and Reid.

Let $T = \begin{pmatrix} T_1 & & \\ & \ddots & \\ & & T_\ell \end{pmatrix}$ be composed into irreducible blocks T_i.

Calls `trlib_leftmost_irreducible()` on every irreducible block in case of coldstart, in case of warmstart just updates information on T_ℓ.

Parameters

- **nirblk** (`trlib_int_t`, input) – number of irreducible blocks ℓ, ensure $\ell > 0$
- **irblk** (`trlib_int_t`, input) – pointer to indices of irreducible blocks, length `nirblk+1`:
  - `irblk[i]` is the start index of block i in `diag` and `offdiag`
  - `irblk[i+1] - 1` is the stop index of block i
  - `irblk[i+1] - irred[i]` the dimension n_ℓ of block i
  - `irblk[nirred]` the dimension of T
- **diag** (`trlib_flt_t`, input) – pointer to array holding diagonal of T, length `irblk[nirblk]`
- **offdiag** (`trlib_flt_t`, input) – pointer to array holding offdiagonal of T, length `irblk[nirblk]`
- **warm** (`trlib_int_t`, input) – set ≥ 1 if you want to update information about block ℓ, provide values in `leftmost_minor`, `ileftmost`, `leftmost`; else `0`
- **leftmost_minor** (`trlib_flt_t`, input) – smallest eigenvalue of principal $(n_\ell - 1) \times (n_\ell - 1)$ submatrix of T_ℓ
- **itmax** (`trlib_int_t`, input) – maximum number of iterations
- **tol_abs** (`trlib_flt_t`, input) – absolute stopping tolerance in Reid-Parlett zero finding, good default may be $\sqrt{\texttt{macheps}}^{3/4}$ (`TRLIB_EPS_POW_75`)
- **verbose** (`trlib_int_t`, input) – determines the verbosity level of output that is written to `fout`
- **unicode** (`trlib_int_t`, input) – set to `1` if `fout` can handle unicode, otherwise to `0`

- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_leftmost_timing_size()*

block	description
0	total duration

- **ileftmost** (trlib_int_t, input/output) – index of block that corresponds to absolute smallest eigenvalue
- **leftmost** (trlib_flt_t, input/output) – smallest eigenvalue of T, length ℓ
  - on entry: allocated memory
  - on exit: leftmost[i] smallest eigenvalue of T_i

Returns

status

- *TRLIB_LMR_CONV* success
- *TRLIB_LMR_ITMAX* iteration limit exceeded

Return type trlib_int_t

trlib_int_t **trlib_leftmost_irreducible**(trlib_int_t *n*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_int_t *warm*, trlib_flt_t *leftmost_minor*, trlib_int_t *itmax*, trlib_flt_t *tol_abs*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_flt_t **leftmost*, trlib_int_t **iter_pr*)

Computes smallest eigenvalue of irreducible symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, using a iteration based on last-pivot function of Parlett and Reid.

Method is sketched on p. 516 in [Gould1999].

Note that this function most likely will fail in the case of a reducible matrix (offdiag contains 0).

Convergence

Convergence is reported if $\texttt{up} - \texttt{low} \leq \texttt{tol\_abs} * \max\{1, |\texttt{low}|, |\texttt{up}|\}$ or $\texttt{prlp} \leq \texttt{tol\_abs}$, low and up denote bracket values enclosing the leftmost eigenvalue and prlp denotes the last-pivot function value used in root finding.

Parameters

- **n** (trlib_int_t, input) – dimension, ensure $n > 0$
- **diag** (trlib_flt_t, input) – pointer to array holding diagonal of T, length n
- **offdiag** (trlib_flt_t, input) – pointer to array holding offdiagonal of T, length n-1
- **warm** (trlib_int_t, input) – set $\geq$ 1 if you provide a valid value in leftmost_minor, else 0. Exact value determines which model will be used for zero-finding

warm	model
0	linear model for rational function
1	sensible heuristic model choice for lifted rational function
2	asymptotic quadratic model $\theta^2 + b\theta + c$ for lifted rational function
3	taylor quadratic model $a\theta^2 + b\theta + c$ for lifted rational function
4	linear model $b\theta + c$ for lifted rational function

- **leftmost_minor** (trlib_flt_t, input) – smallest eigenvalue of principal $(n-1) \times (n-1)$ submatrix of T
- **itmax** (trlib_int_t, input) – maximum number of iterations
- **tol_abs** (trlib_flt_t, input) – absolute stopping tolerance in Reid-Parlett zero finding, good default may be $\sqrt{\texttt{macheps}}^{3/4}$ (*TRLIB_EPS_POW_75*)
- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_leftmost_timing_size()*

block	description
0	total duration

- **leftmost** (trlib_flt_t, output) – smallest eigenvalue of T
- **iter_pr** (trlib_int_t, output) – number of Parlett-Reid iterations

Returns

status

- *TRLIB_LMR_CONV* success
- *TRLIB_LMR_ITMAX* iteration limit exceeded

Return type trlib_int_t

trlib_int_t **trlib_leftmost_timing_size**(void)

size that has to be allocated for timing in *trlib_leftmost_irreducible()* and *trlib_leftmost()*

1.8.2 Definitions

TRLIB_LMR_CONV

0

TRLIB_LMR_ITMAX

-1

TRLIB_LMR_NEWTON_BREAK

3

1.9 CAPI trlib_eigen_inverse

1.9.1 Functions

trlib_int_t **trlib_eigen_inverse**(trlib_int_t *n*, trlib_flt_t **diag*, trlib_flt_t **offdiag*, trlib_flt_t *lam_init*, trlib_int_t *itmax*, trlib_flt_t *tol_abs*, trlib_flt_t **ones*, trlib_flt_t **diag_fac*, trlib_flt_t **offdiag_fac*, trlib_flt_t **eig*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_int_t **timing*, trlib_flt_t **lam_pert*, trlib_flt_t **pert*, trlib_int_t **iter_inv*)

Computes eigenvector to provided eigenvalue of symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, using inverse iteration.

For a description of the method see https://en.wikipedia.org/wiki/Inverse_iteration.

Convergence

Convergence is reported if $|\frac{1}{\|w_{i+1}\|} - \texttt{pert}| \leq \texttt{tol\_abs}$, where $(T - \lambda I)w_{i+1} = v_i$, v_i the current normalized iterate and `pert` is the perturbation applied to the provided eigenvalue.

Parameters

- **n** (`trlib_int_t`, input) – dimension, ensure $n > 0$
- **diag** (`trlib_flt_t`, input) – pointer to array holding diagonal of T, length `n`
- **offdiag** (`trlib_flt_t`, input) – pointer to array holding offdiagonal of T, length `n-1`
- **lam_init** (`trlib_flt_t`, input) – estimation of eigenvalue corresponding to eigenvector to compute
- **itmax** (`trlib_int_t`, input) – maximum number of iterations
- **tol_abs** (`trlib_flt_t`, input) – absolute stopping tolerance in inverse iteration, good default may be $\sqrt{\texttt{macheps}}$ (*TRLIB_EPS_POW_5*)
- **ones** (`trlib_flt_t`, input) – array with every value `1.0`, length `n`
- **diag_fac** (`trlib_flt_t`, input/output) – pointer to array holding diagonal of Cholesky factorization of $T - \lambda I$, length `n`
  - on entry: allocated memory
  - on exit: factorization corresponding to computed eigenvalue @p lam
- **offdiag_fac** – pointer to array holding offdiagonal of Cholesky factorization of $T - \lambda I$, length `n-1`
  - on entry: allocated memory
  - on exit: factorization corresponding to computed eigenvalue @p lam
- **eig** (`trlib_flt_t`, input/output) – pointer to array holding eigenvector, length `n`
- **verbose** (`trlib_int_t`, input) – determines the verbosity level of output that is written to `fout`
- **unicode** (`trlib_int_t`, input) – set to `1` if `fout` can handle unicode, otherwise to `0`
- **prefix** (*char, input*) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream

- **timing** (trlib_int_t, input/output) – gives timing details, provide allocated zero initialized memory of length *trlib_eigen_timing_size()*

block	description
0	total duration
1	timing of linear algebra calls

- **lam_pert** (trlib_flt_t, output) – eigenvalue corresponding to eigenvector
- **pert** (trlib_flt_t, output) – perturbation applied to provided eigenvalue
- **iter_inv** (trlib_int_t, output) – number of inverse iterations

Returns

status

- *TRLIB_EIR_CONV* success
- *TRLIB_EIR_ITMAX* iteration limit exceeded
- *TRLIB_EIR_FAIL_FACTOR* failure on matrix factorization
- *TRLIB_EIR_FAIL_LINSOLVE* failure on backsolve

Return type trlib_int_t

trlib_int_t **trlib_eigen_timing_size**(void)
size that has to be allocated for timing in *trlib_eigen_inverse()*

1.9.2 Definitions

TRLIB_EIR_CONV
0

TRLIB_EIR_ITMAX
-1

TRLIB_EIR_FAIL_FACTOR
-2

TRLIB_EIR_FAIL_LINSOLVE
-3

TRLIB_EIR_N_STARTVEC
5

1.10 CAPI trlib_quadratic_zero

1.10.1 Functions

trlib_int_t **trlib_quadratic_zero**(trlib_flt_t *c_abs*, trlib_flt_t *c_lin*, trlib_flt_t *tol*, trlib_int_t *verbose*, trlib_int_t *unicode*, char **prefix*, FILE **fout*, trlib_flt_t **t1*, trlib_flt_t **t2*)
Computes real zeros of normalized quadratic polynomial.

Parameters

- **c_abs** (trlib_flt_t, input) – absolute coefficient

- **c_lin** (trlib_flt_t, input) – coefficinet of linear term
- **tol** (trlib_flt_t, input) – tolerance that indicates if ill-conditioning present, good default may be macheps$^{3/4}$ (*TRLIB_EPS_POW_75*)
- **verbose** (trlib_int_t, input) – determines the verbosity level of output that is written to fout
- **unicode** (trlib_int_t, input) – set to 1 if fout can handle unicode, otherwise to 0
- **prefix** (trlib_int_t, input) – string that is printed before iteration output
- **fout** (*FILE, input*) – output stream
- **t1** – first zero, t1 $\leq$ t2
- **t2** (trlib_flt_t, output) – second zero, t1 $\leq$ t2

Returns number of zeros

Return type trlib_int_t

1.10.2 Definitions

1.11 CAPI trlib_types

1.11.1 Functions

1.11.2 Definitions

TRLIB_EPS
2.220446049250313e-16

TRLIB_EPS_POW_4
5.4774205922939014e-07

TRLIB_EPS_POW_5
1.4901161193847656e-08

TRLIB_EPS_POW_75
1.8189894035458565e-12

1.12 References

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[Gould1999] N. Gould, S. Lucidi, M. Roma, P. Toint: *Solving the Trust-Region Subproblem using the Lanczos Method*, SIAM J. Optim., 9(2), 504–525, (1999). http://epubs.siam.org/doi/abs/10.1137/S1052623497322735

[Lenders2016] F. Lenders, C. Kirches, A. Potschka: *trlib: A vector-free implementation of the GLTR method for iterative solution of the trust region problem*, submitted to Optimization, Methods and Software. http://www.optimization-online.org/DB_HTML/2016/11/5724.html https://arxiv.org/abs/1611.04718

[Curtis2016] F. Curtis, D. Robinson, M. Samadi: *A trust region algorithm with a worst-case iteration complexity of :math:‘mathcal O(epsilon^{-3/2})‘ for nonconvex optimization*, Mathematical Programming A, 2016, pp. 1-32. http://dx.doi.org/10.1007/s10107-016-1026-2

Index

T

U