ADPCG

Generated by Doxygen 1.9.4

Chapter 1

Design of an adaptive pre-conditioned CG solver

We make a brief note here on rules implemented in the ADP-CG solver.

1.0.0.0.1 Target problem The solver targets consecutive linear systems $A^k x^{k,j} = b^{k,j}, j = 1, ..., r^k$, where, given a sequence of LHS matrices A^k , we solve a set of linear systems with different RHS. $b^{k,j}$ using pre-conditioned conjugate gradient method. One feature we expect is that spectrums of A^k do not change too aggressively and therefore, it is reasonable to assume that if an expensive but accurate pre-conditioner is computed for A^k , it should also be valid for several future $A^{k+1}, ..., A^{k+t}$. The above scenario is quite common for the normal equation in the interior point method ADA^T and ADP-CG is designed for this purpose.

1.0.0.0.2 Origin The ADP-CG solver is initially written to accelerate the SDP solver HDSDP [1].

1.0.0.0.3 Adaptive pre-conditioning One most critical ingredient of ADP-CG is to find **when to update the pre-conditioner**. If the pre-conditioner is too outdated, it is possible that the spectrum after pre-conditioning gets worse. Therefore, we have to decide when to update the pre-conditioner.

In ADP-CG, we use a series of rules to decide whether to update the pre-conditioner.

First we introduce some definitions of statistics that aid our decision

1. We call solution to linear systems with LHS A^k a *round* indexed by k
2. A *solve* is defined by the solution of $A^k x^{k,j} = b^{k,j}$ for some j and round k contains r^k solves

   A solve is either performed by CG or direct solver
3. A *factorization* is defined by the action to factorize A^k for a Cholesky pre-conditioner
4. The (average) solution time of a solve is defined by the (average) time spent in CG (excluding time building pre-conditioner)
5. The (average) factorization time is defined by the (average) time of a factorization
6. A solve is called SUCCESS if the residual norm reaches below tolerance within maximum iteration number

   A solve is called MAXITER if CG exhibits convergence but residual norm fails to reach tolerance within maximum iteration number

   A solve is called FAILED if CG does not exhibit convergence or there is an irreparable error (to be clarified later)

7. Given a pre-conditioner, its *nused* property refers to the number of rounds it has gone through without update
8. The *latesttime* property refers to the average solution time in the lastest round

Based on the above statistics, we now clarify the rules.

1. Update of diagonal pre-conditioner always happens at the beginning of a round and the update of Cholesky pre-conditioner happens **either** at the beginning a round **or** within the first solve in a round
2. In the first solve of each round, if the Cholesky pre-conditioner is not updated at the beginning of the round, there is a chance to regret

   i.e., if the first CG loop is not SUCCESS due to diagonal pre-conditioner or an outdated Cholesky pre-conditioner, it is allowed to perform a make-up Cholesky factorization step and then update the pre-conditioner. Note that by rule (4), the rest of the solves in this round would by done by direct solver and by rule (5), diagonal pre-conditioner will never be used.
3. At the beginning of each round, pre-conditioner is updated if one of the following criteria, checked in order, is satisfied
   - If the system is classified as ill-conditioned or indefinite by some user-defined criterion
   - If the diagonal pre-conditioner is used
   - **If latesttime $>$ 1.5 average solution time**
   - If ADP-CG is asked to perform direct solve
   - **If average solution time $>$ average factorization time**
   - If the *nused* property of the current pre-conditioner exceeds the user-defined threshold
4. If the Cholesky pre-conditioner is updated in round k , then all the solves in round k after the update are solved by the direct solver
5. If CG switches to Cholesky pre-conditioner, it never returns to diagonal pre-conditioner unless requested by the user
6. The following cases result in a FAILED solve:
   - If pre-conditioner build-up fails
   - If direct solve fails
   - If pre-conditioning step fails
   - If NAN appears in the CG solver

   If FAILED occurs, the current solution is not trustworthy and the whole solution procedure stops due to irreparable error

Here are some extra rules tailored for the deteriorating conditioning of normal equations arising from the interior point method

1. If the number of MAXITER exceeds T in round k , then all the solves thereafter are solved by direct solver

   References$**$

[1] Gao, Wenzhi, Dongdong Ge, and Yinyu Ye. "HDSDP: Software for Semidefinite Programming." *arXiv preprint arXiv:2207.13862* (2022).

Chapter 2

Design of an adaptive pre-conditioned CG solver

We make a brief note here on rules implemented in the ADP-CG solver.

2.0.0.1 Target problem

The solver targets consecutive linear systems $A^kx^{k, j} = b^{k, j}, j = 1,..., r^k$, where, given a sequence of LHS matrices A^k, we solve a set of linear systems with different RHS. $b^{k, j}$ using pre-conditioned conjugate gradient method. One feature we expect is that spectrums of A^k do not change too aggressively and therefore, it is reasonable to assume that if an expensive but accurate pre-conditioner is computed for A^k, it should also be valid for several future $A^{k + 1}, ..., A^{k+\tau}$. The above scenario is quite common for the normal equation in the interior point method ADA^T and ADP-CG is designed for this purpose.

2.0.0.2 Origin

The ADP-CG solver is initially written to accelerate the SDP solver HDSDP [1].

2.0.0.3 Adaptive pre-conditioning

One most critical ingredient of ADP-CG is to find **when to update the pre-conditioner**. If the pre-conditioner is too outdated, it is possible that the spectrum after pre-conditioning gets worse. Therefore, we have to decide when to update the pre-conditioner.

In ADP-CG, we use a series of rules to decide whether to update the pre-conditioner.

First we introduce some definitions of statistics that aid our decision

1. We call solution to linear systems with LHS A^k a *round* indexed by k
2. A *solve* is defined by the solution of $A^k x^{k, j} = b^{k, j}$ for some j and round k contains r^k solves

   A solve is either performed by CG or direct solver
3. A *factorization* is defined by the action to factorize A^k for a Cholesky pre-conditioner
4. The (average) solution time of a solve is defined by the (average) time spent in CG (excluding time building pre-conditioner)

5. The (average) factorization time is defined by the (average) time of a factorization
6. A solve is called SUCCESS if the residual norm reaches below tolerance within maximum iteration number

   A solve is called MAXITER if CG exhibits convergence but residual norm fails to reach tolerance within maximum iteration number

   A solve is called FAILED if CG does not exhibit convergence or there is an irreparable error (to be clarified later)
7. Given a pre-conditioner, its *nused* property refers to the number of rounds it has gone through without update
8. The *latesttime* property refers to the average solution time in the lastest round

Based on the above statistics, we now clarify the rules.

1. Update of diagonal pre-conditioner always happens at the beginning of a round and the update of Cholesky pre-conditioner happens **either** at the beginning a round **or** within the first solve in a round
2. In the first solve of each round, if the Cholesky pre-conditioner is not updated at the beginning of the round, there is a chance to regret

   i.e., if the first CG loop is not SUCCESS due to diagonal pre-conditioner or an outdated Cholesky pre-conditioner, it is allowed to perform a make-up Cholesky factorization step and then update the pre-conditioner. Note that by rule (4), the rest of the solves in this round would by done by direct solver and by rule (5), diagonal pre-conditioner will never be used.
3. At the beginning of each round, pre-conditioner is updated if one of the following criteria, checked in order, is satisfied
   - If the system is classified as ill-conditioned or indefinite by some user-defined criterion
   - If the diagonal pre-conditioner is used
   - **If latesttime $>$ 1.5 average solution time**
   - If ADP-CG is asked to perform direct solve
   - **If average solution time $>$ average factorization time**
   - If the *nused* property of the current pre-conditioner exceeds the user-defined threshold
4. If the Cholesky pre-conditioner is updated in round k, then all the solves in round k after the update are solved by the direct solver
5. If CG switches to Cholesky pre-conditioner, it never returns to diagonal pre-conditioner unless requested by the user
6. The following cases result in a FAILED solve:
   - If pre-conditioner build-up fails
   - If direct solve fails
   - If pre-conditioning step fails
   - If NAN appears in the CG solver

   If FAILED occurs, the current solution is not trustworthy and the whole solution procedure stops due to irreparable error

Here are some extra rules tailored for the deteriorating conditioning of normal equations arising from the interior point method

1. If the number of MAXITER exceeds T in round k, then all the solves thereafter are solved by direct solver

References

[1] Gao, Wenzhi, Dongdong Ge, and Yinyu Ye. "HDSDP: Software for Semidefinite Programming." *arXiv preprint arXiv:2207.13862* (2022).

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 adpcg Struct Reference

Working struct for the adaptive CG solver.

```
#include <adpcg.h>
```

Data Fields

- void ∗ **A**

  LHS data.
- void ∗ **r**

  Residual.
- void ∗ **rnew**

  Workspace array.
- void ∗ **d**

  Workspace array.
- void ∗ **pinvr**

  Workspace array.
- void ∗ **Ad**

  Workspace array.
- void ∗ **x**

  CG solution vector.
- void ∗ **aux**

  CG auxiliary array.
- void ∗ **btmp**

  CG temporary RHS array.
- cgint **ptype**

  Pre-conditioner type.
- void ∗ **chol**

  Cholesky pre-conditioner.
- void ∗ **diag**

  Diagonal pre-conditioner.
- double **tol**

  Relative tolerance of CG.

- double **rnrm**

  Residual norm.
- double **avgsvtime**

  Averate solution time of previous CG solves.
- double **avgfctime**

  Average factorization time of previous CG solves.
- double **currenttime**

  Buffer of solution time in the current round.
- double **latesttime**

  Time for the latest solve.
- cgint **n**

  Dimension of linear system.
- cgint **niter**

  Number of iterations in most recent solve.
- cgint **maxiter**

  Maximum number of iterations.
- cgint **status**

  Solution status.
- cgint **reuse**

  Reuse Cholesky pre-conditioner.
- cgint **nused**

  Number of rounds current Cholesky pre-conditioner is already used.
- cgint **nmaxiter**

  Number of non-successfull solves.
- cgint **restart**

  Restart frequency.
- cgint **nfactors**

  Number of factorizes performed so far.
- cgint **nrounds**

  Number of rounds of solves.
- cgint **nsolverd**

  Number of solves in a round.
- cgint **nsolves**

  Number of linear systems solved.
- void(∗ **v_init**)(void ∗v)

  Initialize vector.
- cgint(∗ **v_alloc**)(void ∗v, cgint n)

  Allocate memory for v.
- void(∗ **v_free**)(void ∗v)

  Free the internal memory of vector.
- void(∗ **v_copy**)(void ∗s, void ∗t)

  Copy s to t.
- void(∗ **v_reset**)(void ∗v)

  Reset vector to 0.
- void(∗ **v_norm**)(void ∗v, double ∗nrm)

  nrm = norm(v)
- void(∗ **v_axpy**)(double a, void ∗x, void ∗y)

  $y = y + a * x$
- void(∗ **v_axpby**)(double a, void ∗x, double b, void ∗y)

  $y = a * x + b * y$
- void(∗ **v_zaxpby**)(void ∗z, double a, void ∗x, double b, void ∗y)

  $z = a * x + b * y$
- void(∗ **v_dot**)(void ∗x, void ∗y, double ∗xTy)

  $xTy = x' * y$
- cgint(∗ **A_chol**)(void ∗A)

  Compute Cholesky of A.
- cgint(∗ **A_cond**)(void ∗A)

  Evaluate conditioning of A.
- void(∗ **A_getdiag**)(void ∗A, void ∗diag)

  Compute diagonal pre-conditioner diag = diag(A)
- cgint(∗ **diagpcd**)(void ∗diag, void ∗v)

  Diagonal Preconditioning operation v = P \ v.
- cgint(∗ **cholpcd**)(void ∗chol, void ∗v, void ∗aux)

  Cholesky Preconditioning operation v = P \ v.
- void(∗ **Av**)(void ∗A, void ∗v, void ∗Av)

  Compute Av = A ∗ v.
- cgint(∗ **Ainv**)(void ∗A, void ∗v, void ∗aux)

  Solve Ainvv = A \ v.

5.1.1 Detailed Description

Working struct for the adaptive CG solver.

To make the CG solver more general, the operations on

1. vector 2. matrix 3. matrix-vector are presented in abstract function pointers and users can define their own implementations for various purposes

More details: the following operations are expected Vector:

1. v_init(v) Initialize a vector struct
2. v_alloc(v) Allocate internal memory for a vector struct
3. v_free(v) Free the internal memory for a vector struct
4. v_copy(s, t) Copy content from vector s to vector t
5. v_reset(v) Set the content of vector 0
6. v_norm(v, &nrm) Compute norm of v
7. v_axpy(a, x, y) Compute y = y + a ∗ x
8. v_zaxpby(z, a, x, b, y) Compute z = a ∗ x + b ∗ y
9. v_dot(x, y, &dot) Compute x' ∗ y

Matrix:

1. A_chol(A) Perform Cholesky decomposition
2. is_illcond = A_cond(A) Get a boolean variable reflecting the conditioning of A

Matrix-vector:

1. diagpcd(diag, v) Perform diagonal pre-conditioning
2. cholpcf(chol, v) Perform Cholesky pre-conditioning

The documentation for this struct was generated from the following file:

- src/adpcg.h

Chapter 6

File Documentation

6.1 src/adpcg.c File Reference

Header for basic types and routine list.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "adpcg.h"
```

Macros

- #define **adpcg_free**(var) do {free((var)); (var) = NULL;} while (0)

Functions

- static double my_clock (void)

    Get time stamp for now.
- static cgint cg_prepare_preconditioner (adpcg ∗cg)

    Prepare pre-conditioner for the CG solver.
- static cgint cg_precondition (adpcg ∗cg, void ∗v)

    Apply pre-conditioning.
- static cgint cg_decision (adpcg ∗cg)

    Decide whether to update pre-conditioner.
- cgint cg_iteration (adpcg ∗cg, void ∗b, cgint warm)

    Implement conjugate gradinet.
- void cg_init (adpcg ∗cg)

    Initialize the conjugate gradient solver.
- cgint cg_alloc (adpcg ∗cg, cgint n, cgint vsize)

    Allocate internal memory for CG solver.
- void cg_register (adpcg ∗cg, void ∗A, void ∗diag, void ∗chol)

    Link pointers to LHS matrix, diagonal, and Cholesky pre-conditioner.
- void cg_free (adpcg ∗cg)

Free the internal memory of CG solver.

- void cg_setparam (adpcg ∗cg, double tol, cgint reuse, cgint maxiter, cgint restart)

  Set parameters for the CG solver.
- void cg_getstats (adpcg ∗cg, cgint ∗status, cgint ∗niter, double ∗rnrm, double ∗avgsvtime, double ∗avgfctime, cgint ∗nused, cgint ∗nmaixter, cgint ∗nfactors, cgint ∗nrounds, cgint ∗nsolverd, cgint ∗nsolves)

  Extract CG statistics after some solve.
- cgint cg_start (adpcg ∗cg)

  Start a round of solves.
- void cg_finish (adpcg ∗cg)

  Finish a round of solve.
- cgint cg_solve (adpcg ∗cg, void ∗b, void ∗x0)

  Solve linear system using adaptive pre-conditioned conjugate gradient with restart.

6.1.1 Detailed Description

Header for basic types and routine list.

Solve a sequence of linear systems by either pre-conditioned conjugate gradient or direct solver, which is chosen heuristically based on problem conditioning. The routine also implements an adaptive pre-conditioning mechanism that updates the pre-conditioner automatically. Diagonal and Cholesky pre-conditioners are implemented

Author

Wenzhi Gao, Shanghai University of Finance and Economics

Date

Aug 29th, 2022

6.1.2 Function Documentation

6.1.2.1 cg_alloc()

```
cgint cg_alloc (
            adpcg * cg,
            cgint n,
            cgint vsize )
```

Allocate internal memory for CG solver.

Parameters

in	*cg*	Adaptive CG solver
in	*n*	Dimension of the linear system
in	*vsize*	Size of vector structure

Returns

CG_OK if memory is successfully allocated

Allocate the internal memory for the CG solver.

6.1.2.2 cg_decision()

```
static cgint cg_decision (
            adpcg * cg )  [static]
```

Decide whether to update pre-conditioner.

Parameters

in	*cg*	CG solver

1.If the system is classified as ill-conditioned or indefinite by some user-defined criterion 2.If the diagonal pre-conditioner is used 3.If latesttime $>$ 1.5 average solution time 4.If ADP-CG is asked to perform direct solve 5.If average solution time $>$ average factorization time 6.If the "nused" property of the current pre-conditioner exceeds the user-defined threshold

6.1.2.3 cg_finish()

```
void cg_finish (
            adpcg * cg )
```

Finish a round of solve.

Parameters

in	*cg*	CG solver

At the end of each round,

1. nused increases by 1
2. currenttime overwrites latesttime
3. nsolverd is reset

6.1.2.4 cg_free()

```
void cg_free (
            adpcg * cg )
```

Free the internal memory of CG solver.

Parameters

in	*cg*	CG solver

Free all the internal memory allocated by adaptive CG solver. The solver pointer itself has to be freed by user.

6.1.2.5 cg_getstats()

```
void cg_getstats (
            adpcg * cg,
            cgint * status,
            cgint * niter,
            double * rnrm,
            double * avgsvtime,
            double * avgfctime,
            cgint * nused,
            cgint * nmaixter,
            cgint * nfactors,
            cgint * nrounds,
            cgint * nsolverd,
            cgint * nsolves )
```

Extract CG statistics after some solve.

Parameters

in	*cg*	CG solver
out	*status*	CG solution status
out	*niter*	Number of CG iterations
out	*rnrm*	Residual norm
out	*avgsvtime*	Current avarage solution time
out	*avgfctime*	Current average factorization time
out	*nused*	Current number of iterations the pre-conditioner is used
out	*nmaixter*	Number of iterations CG fails to solve the system
out	*nfactors*	Number of factorizations
out	*nrounds*	Number of rounds
out	*nsolverd*	Number of solves in the current round
out	*nsolves*	Number of solves

Collect different CG statistics. If some statistic is not needed, just let it be NULL.

6.1.2.6 cg_init()

```
void cg_init (
            adpcg * cg )
```

Initialize the conjugate gradient solver.

Parameters

in	*cg*	Adaptive CG solver Initlaize all the pointers to NULL and values to 0

6.1.2.7 cg_iteration()

```
cgint cg_iteration (
            adpcg * cg,
            void * b,
            cgint warm )
```

Implement conjugate gradinet.

Parameters

in	*cg*	CG solver
in	*b*	RHS vector
in	*warm*	If there is warm start?

Implement pre-conditioned CG with restart

6.1.2.8 cg_precondition()

```
static cgint cg_precondition (
            adpcg * cg,
            void * v ) [static]
```

Apply pre-conditioning.

Parameters

in	*cg*	CG solver
in	*v*	Overwritten by P \ v

Returns

CG_OK if pre-conditioning is done successfully

6.1.2.9 cg_prepare_preconditioner()

```
static cgint cg_prepare_preconditioner (
            adpcg * cg ) [static]
```

Prepare pre-conditioner for the CG solver.

Parameters

in	*cg*	CG solver

Returns

CG_OK if the pre-conditioner is successfully collected

The method invokes internal preparation routine of pre-conditioner. The time computing pre-conditioner is counted into avgsvtime

6.1.2.10 cg_register()

```
void cg_register (
            adpcg * cg,
            void * A,
            void * diag,
            void * chol )
```

Link pointers to LHS matrix, diagonal, and Cholesky pre-conditioner.

Parameters

in	*cg*	CG Solver
in	*A*	Left hand side matrix
in	*diag*	Diagonal matrix
in	*chol*	Cholesky factor

Register pointers for coefficient data, diagonal matrix and Cholesky factor

6.1.2.11 cg_setparam()

```
void cg_setparam (
            adpcg * cg,
            double tol,
            cgint reuse,
            cgint maxiter,
            cgint restart )
```

Set parameters for the CG solver.

Parameters

in	*cg*	CG solver
in	*tol*	Relative solution tolerance
in	*reuse*	The maximum reuse number
in	*maxiter*	Maximum of iteration
in	*restart*	Restart frequency of CG. -1 if automatically decided

Set CG parameters

6.1.2.12 cg_solve()

```
cgint cg_solve (
```

```
            adpcg * cg,
            void * b,
            void * x0 )
```

Solve linear system using adaptive pre-conditioned conjugate gradient with restart.

Parameters

in	*cg*	CG Solver
in	*b*	RHS vector. Overwritten when solved
in	*x0*	Initial point

Solve the linear system by adaptive pre-conditioning and restart.

6.1.2.13 cg_start()

```
cgint cg_start (
            adpcg * cg )
```

Start a round of solves.

Parameters

in	*cg*	CG solver

Several rules decide whether to update the current pre-conditioner

6.1.2.14 my_clock()

```
static double my_clock (
            void ) [static]
```

Get time stamp for now.

Returns

Current time stamp

6.2 src/adpcg.h File Reference

Header for basic types and routine list.

```
#include <stddef.h>
```

Data Structures

- struct adpcg

    Working struct for the adaptive CG solver.

Macros

- #define **id** "%d"
- #define **cgerr**(x) printf(x);
- #define **CG_OK** (0)
- #define **CG_ERR** (1)
- #define **CG_TRUE** (1)
- #define **CG_FALSE** (0)
- #define **CG_PRECOND_DIAG** (10)

  Use diagonal as the pre-conditioner.
- #define **CG_PRECOND_CHOL** (11)

  Use Cholesky factor as the pre-conditioner.
- #define **CG_NO_PRECOND** (12)

  Use direct solver.
- #define **CG_STATUS_SOLVED** (100)

  System is solved to desired accuracy within maximum iteration.
- #define **CG_STATUS_MAXITER** (101)

  System reaches maximum iteration.
- #define **CG_STATUS_FAILED** (102)

  CG fails to converge.
- #define **CG_STATUS_UNKNOWN** (104)

  CG status is not known. Solution not yet started.
- #define **CG_STATUS_DIRECT** (105)

  CG serves as a wrapper for direct solver.
- #define **MIN**(a, b) (((a)<(b))?(a):(b))
- #define **MAX**(a, b) (((a)>(b))?(a):(b))

Typedefs

- typedef int32_t **cgint**

6.2.1 Detailed Description

Header for basic types and routine list.

Given a set of positive definite linear systems $A^k x = b^k$, adpcg solves them adaptively with pre-conditioning conjugate gradient method.

The routine is employed in HDSDP.

Author

Wenzhi Gao, Shanghai University of Finance and Economics

Date

Aug 29th, 2022

6.3 adpcg.h

Go to the documentation of this file.

```

#ifndef adpcg_h
#define adpcg_h


#include <stddef.h>

#ifdef ADPCG_64
typedef int64_t cgint;
#define id "%lld"
#else
typedef int32_t cgint;
#define id "%d"
#endif

#define cgerr(x) printf(x);

/* Return code */
#define CG_OK                 (0)
#define CG_ERR                (1)

/* Boolean */
#define CG_TRUE               (1)
#define CG_FALSE              (0)

/* Pre-conditioner */
#define CG_PRECOND_DIAG      (10)
#define CG_PRECOND_CHOL      (11)
#define CG_NO_PRECOND        (12)

/* Solution status */
#define CG_STATUS_SOLVED     (100)
#define CG_STATUS_MAXITER    (101)
#define CG_STATUS_FAILED     (102)
#define CG_STATUS_UNKNOWN    (104)
#define CG_STATUS_DIRECT     (105)

/* Auxiliary macros */
#ifndef MIN
#define MIN(a, b)  (((a)<(b))?(a):(b))
#endif /* MIN */
#ifndef MAX
#define MAX(a, b)  (((a)>(b))?(a):(b))
#endif  /* MAX */

typedef struct {

    void *A;
    void *r;
    void *rnew;
    void *d;
    void *pinvr;
    void *Ad;
    void *x;
    void *aux;
    void *btmp;

    cgint ptype;
    void *chol;
    void *diag;

    double tol;
    double rnrm;
    double avgsvtime;
    double avgfctime;
    double currenttime;
    double latesttime;

    cgint  n;
    cgint  niter;
    cgint  maxiter;
    cgint  status;
    cgint  reuse;
    cgint  nused;
    cgint  nmaxiter;
    cgint  restart;
    cgint  nfactors;
    cgint  nrounds;
    cgint  nsolverd;
    cgint  nsolves;

    /* Vector operations */
```

```
    void  (*v_init)   (void *v);
    cgint (*v_alloc)  (void *v, cgint n);
    void  (*v_free)   (void *v);
    void  (*v_copy)   (void *s, void *t);
    void  (*v_reset)  (void *v);
    void  (*v_norm)   (void *v, double *nrm);
    void  (*v_axpy)   (double a, void *x, void *y);
    void  (*v_axpby)  (double a, void *x, double b, void *y);
    void  (*v_zaxpby) (void *z, double a, void *x, double b, void *y);
    void  (*v_dot)    (void *x, void *y, double *xTy);

    /* Matrix operations */
    cgint (*A_chol) (void *A);
    cgint (*A_cond) (void *A);
    void  (*A_getdiag) (void *A, void *diag);

    /* Matrix-vector operations */
    cgint (*diagpcd)(void *diag, void *v);
    cgint (*cholpcd)(void *chol, void *v, void *aux);
    void  (*Av) (void *A, void *v, void *Av);
    cgint (*Ainv) (void *A, void *v, void *aux);

} adpcg;


#endif /* adpcg_h */
```

Index