Help

```
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else
/*****************************************************************************/
/*                                  eigenval.c                               */
/*****************************************************************************/
/*                                                                           */
/* estimation of extremal EIGENVALues                                        */
/*                                                                           */
/* Copyright (C) 1992-1995 Tomas Skalicky. All rights reserved.              */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*        ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF THE TERMS           */
/*              OF THE COPYRIGHT NOTICE (SEE FILE COPYRGHT.H)                */
/*                                                                           */
/*****************************************************************************/

#include <stddef.h>
#include <
href../../../common/math/cdo/cdo_math_h_src.pdfmath.h>

#include "
href../../../common/math/highdim_solver/laspack/eigenval_h_src.pdflaspack/eigenv
#include "
href../../../common/math/highdim_solver/laspack/elcmp_h_src.pdflaspack/elcmp.h"
#include "
href../../../common/math/highdim_solver/laspack/errhandl_h_src.pdflaspack/errhan
#include "
href../../../common/math/highdim_solver/laspack/operats_h_src.pdflaspack/operats
#include "
href../../../common/math/highdim_solver/laspack/rtc_h_src.pdflaspack/rtc.h"
#include "
href../../../common/math/highdim_solver/laspack/copyrght_h_src.pdflaspack/copyrg

typedef struct
{
  double MinEigenval;
  double MaxEigenval;
  PrecondProcType PrecondProcUsed;
```

```
  double OmegaPrecondUsed;
} EigenvalInfoType;

/* accuracy for the estimation of extremal eigenvalues */
static double EigenvalEps = 1e-4;

static void EstimEigenvals(QMatrix *A, PrecondProcType PrecondProc, double Omega
static void SearchEigenval(size_t n, double *Alpha, double *Beta, size_t k,
                           double BoundMin, double BoundMax, Boolean *Found, dou
static size_t NoSmallerEigenvals(size_t n, double *Alpha, double *Beta, double L

#define max(x, y) ((x) > (y) ? (x) : (y))
#define min(x, y) ((x) < (y) ? (x) : (y))

void SetEigenvalAccuracy(double Eps)
/* set accuracy for the estimation of extremal eigenvalues */
{
  EigenvalEps = Eps;
}

double GetMinEigenval(QMatrix *A, PrecondProcType PrecondProc, double OmegaPreco
/* returns estimate for minimum eigenvalue of the matrix A */
{
  double MinEigenval;

  EigenvalInfoType *EigenvalInfo;

  Q_Lock(A);

  if (LASResult() == LASOK)
    {
      EigenvalInfo = (EigenvalInfoType *) * (Q_EigenvalInfo(A));
      /* if eigenvalues not estimated yet, ... */
      if (EigenvalInfo == NULL)
        {
          EigenvalInfo = (EigenvalInfoType *)malloc(sizeof(EigenvalInfoType));
          if (EigenvalInfo != NULL)
            {
              *(Q_EigenvalInfo(A)) = (void *)EigenvalInfo;
              EstimEigenvals(A, PrecondProc, OmegaPrecond);
            }
```

```
          else
            {
              LASError(LASMemAllocErr, "GetMinEigenval", Q_GetName(A), NULL, NUL
            }
        }

      /* if eigenvalues estimated with an other preconditioner, ... */
      if (EigenvalInfo->PrecondProcUsed != PrecondProc
          || EigenvalInfo->OmegaPrecondUsed != OmegaPrecond)
        {
          EstimEigenvals(A, PrecondProc, OmegaPrecond);
        }

      if (LASResult() == LASOK)
        MinEigenval = EigenvalInfo->MinEigenval;
      else
        MinEigenval = 1.0;
    }
  else
    {
      MinEigenval = 1.0;
    }

  return (MinEigenval);
}

double GetMaxEigenval(QMatrix *A, PrecondProcType PrecondProc, double OmegaPreco
/* returns estimate for maximum eigenvalue of the matrix A */
{
  double MaxEigenval;

  EigenvalInfoType *EigenvalInfo;

  Q_Lock(A);

  if (LASResult() == LASOK)
    {
      EigenvalInfo = (EigenvalInfoType *) * (Q_EigenvalInfo(A));
      /* if eigenvalues not estimated yet, ... */
      if (EigenvalInfo == NULL)
        {
```

```
          EigenvalInfo = (EigenvalInfoType *)malloc(sizeof(EigenvalInfoType));
          if (EigenvalInfo != NULL)
            {
              *(Q_EigenvalInfo(A)) = (void *)EigenvalInfo;
              EstimEigenvals(A, PrecondProc, OmegaPrecond);
            }
          else
            {
              LASError(LASMemAllocErr, "GetMaxEigenval", Q_GetName(A), NULL, NUL
            }
        }

      /* if eigenvalues estimated with an other preconditioner, ... */
      if (EigenvalInfo->PrecondProcUsed != PrecondProc
          || EigenvalInfo->OmegaPrecondUsed != OmegaPrecond)
        {
          EstimEigenvals(A, PrecondProc, OmegaPrecond);
        }

      if (LASResult() == LASOK)
        MaxEigenval = EigenvalInfo->MaxEigenval;
      else
        MaxEigenval = 1.0;
    }
  else
    {
      MaxEigenval = 1.0;
    }

  return (MaxEigenval);
}

static void EstimEigenvals(QMatrix *A, PrecondProcType PrecondProc, double Omega
/* estimates extremal eigenvalues of the matrix A by means of the Lanczos method
{
  /*
   *  for details to the Lanczos algorithm see
   *
   *  G. H. Golub, Ch. F. van Loan:
   *  Matrix Computations;
   *  North Oxford Academic, Oxford, 1986
```

```
 *
 *  (for modification for preconditioned matrices compare with sec. 10.3)
 *
 */

double LambdaMin = 0.0, LambdaMax = 0.0;
double LambdaMinOld, LambdaMaxOld;
double GershBoundMin = 0.0, GershBoundMax = 0.0;
double *Alpha, *Beta;
size_t Dim, j;
Boolean Found;
Vector q, qOld, h, p;

Q_Lock(A);

Dim = Q_GetDim(A);
V_Constr(&q, "q", Dim, Normal, True);
V_Constr(&qOld, "qOld", Dim, Normal, True);
V_Constr(&h, "h", Dim, Normal, True);
if (PrecondProc != NULL)
  V_Constr(&p, "p", Dim, Normal, True);

if (LASResult() == LASOK)
  {
    Alpha = (double *)malloc((Dim + 1) * sizeof(double));
    Beta = (double *)malloc((Dim + 1) * sizeof(double));
    if (Alpha != NULL && Beta != NULL)
      {
        j = 0;

        V_SetAllCmp(&qOld, 0.0);
        V_SetRndCmp(&q);
        if (Q_KerDefined(A))
          OrthoRightKer_VQ(&q, A);
        if (Q_GetSymmetry(A) && PrecondProc != NULL)
          {
            (*PrecondProc)(A, &p, &q, OmegaPrecond);
            MulAsgn_VS(&q, 1.0 / sqrt(Mul_VV(&q, &p)));
          }
        else
          {
```

```
    MulAsgn_VS(&q, 1.0 / l2Norm_V(&q));
  }

Beta[0] = 1.0;
do
  {
    j++;
    if (Q_GetSymmetry(A) && PrecondProc != NULL)
      {
        /* p = M^(-1) q */
        (*PrecondProc)(A, &p, &q, OmegaPrecond);
        /* h = A p */
        Asgn_VV(&h, Mul_QV(A, &p));
        if (Q_KerDefined(A))
          OrthoRightKer_VQ(&h, A);
        /* Alpha = p . h */
        Alpha[j] = Mul_VV(&p, &h);
        /* r = h - Alpha q - Beta qOld */
        SubAsgn_VV(&h, Add_VV(Mul_SV(Alpha[j], &q), Mul_SV(Beta[j - 1]
        /* z = M^(-1) r */
        (*PrecondProc)(A, &p, &h, OmegaPrecond);
        /* Beta = sqrt(r . z) */
        Beta[j] = sqrt(Mul_VV(&h, &p));
        Asgn_VV(&qOld, &q);
        /* q = r / Beta */
        Asgn_VV(&q, Mul_SV(1.0 / Beta[j], &h));
      }
    else
      {
        /* h = A p */
        if (Q_GetSymmetry(A))
          {
            Asgn_VV(&h, Mul_QV(A, &q));
          }
        else
          {
            if (PrecondProc != NULL)
              {
                (*PrecondProc)(A, &h, Mul_QV(A, &q), OmegaPrecond);
                (*PrecondProc)(Transp_Q(A), &h, &h, OmegaPrecond);
                Asgn_VV(&h, Mul_QV(Transp_Q(A), &h));
```

```
          }
        else
          {
            Asgn_VV(&h, Mul_QV(Transp_Q(A), Mul_QV(A, &q)));
          }
      }
    if (Q_KerDefined(A))
      OrthoRightKer_VQ(&h, A);
    /* Alpha = q . h */
    Alpha[j] = Mul_VV(&q, &h);
    /* r = h - Alpha q - Beta qOld */
    SubAsgn_VV(&h, Add_VV(Mul_SV(Alpha[j], &q), Mul_SV(Beta[j - 1]
    /* Beta = || r || */
    Beta[j] = l2Norm_V(&h);
    Asgn_VV(&qOld, &q);
    /* q = r / Beta */
    Asgn_VV(&q, Mul_SV(1.0 / Beta[j], &h));
  }

LambdaMaxOld = LambdaMax;
LambdaMinOld = LambdaMin;

/* determination of extremal eigenvalues of the tridiagonal matrix
   (Beta[i-1] Alpha[i] Beta[i]) (where 1 <= i <= j)
by means of the method of bisection; bounds for eigenvalues
are determined after Gershgorin circle theorem */
if (j == 1)
  {
    GershBoundMin = Alpha[1] - fabs(Beta[1]);
    GershBoundMax = Alpha[1] + fabs(Beta[1]);

    LambdaMin = Alpha[1];
    LambdaMax = Alpha[1];
  }
else
  {
    GershBoundMin = min(Alpha[j] - fabs(Beta[j]) - fabs(Beta[j - 1
                        GershBoundMin);
    GershBoundMax = max(Alpha[j] + fabs(Beta[j]) + fabs(Beta[j - 1
                        GershBoundMax);
```

```
                SearchEigenval(j, Alpha, Beta, 1, GershBoundMin, LambdaMin,
                               &Found, &LambdaMin);
                if (!Found)
                  SearchEigenval(j, Alpha, Beta, 1, GershBoundMin, GershBoundM
                                 &Found, &LambdaMin);

                SearchEigenval(j, Alpha, Beta, j, LambdaMax, GershBoundMax,
                               &Found, &LambdaMax);
                if (!Found)
                  SearchEigenval(j, Alpha, Beta, j, GershBoundMin, GershBoundM
                                 &Found, &LambdaMax);
              }
          }
        while (!IsZero(Beta[j]) && j < Dim
               && (fabs(LambdaMin - LambdaMinOld) > EigenvalEps * LambdaMin
                   || fabs(LambdaMax - LambdaMaxOld) > EigenvalEps * LambdaMax
               && LASResult() == LASOK);

        if (Q_GetSymmetry(A))
          {
            LambdaMin = (1.0 - j * EigenvalEps) * LambdaMin;
          }
        else
          {
            LambdaMin = (1.0 - sqrt(j) * EigenvalEps) * sqrt(LambdaMin);
          }
        if (Alpha != NULL)
          free(Alpha);
        if (Beta != NULL)
          free(Beta);
      }
    else
      {
        LASError(LASMemAllocErr, "EstimEigenvals", Q_GetName(A), NULL, NULL);
      }

  }

V_Destr(&q);
V_Destr(&qOld);
V_Destr(&h);
```

```
  if (PrecondProc != NULL)
    V_Destr(&p);

  if (LASResult() == LASOK)
    {
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->MinEigenval = LambdaMin;
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->MaxEigenval = LambdaMax;
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->PrecondProcUsed = PrecondPro
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->OmegaPrecondUsed = OmegaPrec
    }
  else
    {
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->MinEigenval = 1.0;
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->MaxEigenval = 1.0;
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->PrecondProcUsed = NULL;
      ((EigenvalInfoType *) * (Q_EigenvalInfo(A)))->OmegaPrecondUsed = 1.0;
    }

  Q_Unlock(A);
}

static void SearchEigenval(size_t n, double *Alpha, double *Beta, size_t k,
                           double BoundMin, double BoundMax, Boolean *Found, dou
/* search the k-th eigenvalue of the tridiagonal matrix
   (Beta[i-1] Alpha[i] Beta[i]) (where 1 <= i <= n)
   by means of the method of bisection */
{
  /*
   *  for details to the method of bisection see
   *
   *  G. H. Golub, Ch. F. van Loan:
   *  Matrix Computations;
   *  North Oxford Academic, Oxford, 1986
   *
   */

  if (NoSmallerEigenvals(n, Alpha, Beta, BoundMin) < k
      && NoSmallerEigenvals(n, Alpha, Beta, BoundMax) >= k)
    {
      while (fabs(BoundMax - BoundMin) > 0.01 * EigenvalEps
             * (fabs(BoundMin) + fabs(BoundMax)))
```

```
        {
          *Lambda = 0.5 * (BoundMin + BoundMax);
          if (NoSmallerEigenvals(n, Alpha, Beta, *Lambda) >= k)
            BoundMax = *Lambda;
          else
            BoundMin = *Lambda;
        }
      *Lambda = BoundMax;

      *Found = True;
    }
  else
    {
      *Found = False;
    }
}

static size_t NoSmallerEigenvals(size_t n, double *Alpha, double *Beta, double L
/* returns number of eigenvalues of the tridiagonal matrix
   (Beta[i-1] Alpha[i] Beta[i]) (where 1 <= i <= n)
   which are less then Lambda */
{
  size_t No;

  double p, pNew, pOld, Sign;
  size_t i;

  No = 0;

  pOld = 1.0;
  p = (Alpha[1] - Lambda) / fabs(Beta[1]);
  /* check for change of sign */
  if (IsZero(p) || p * pOld < 0)
    No++;

  for (i = 2; i <= n; i++)
    {
      Sign = Beta[i - 1] / fabs(Beta[i - 1]);
      pNew = ((Alpha[i] - Lambda) * p - Beta[i - 1] * Sign * pOld) / fabs(Beta[i
      pOld = p;
      p = pNew;
```

```
      /* check for change of sign */
      if (p * pOld < 0 || (IsZero(p) && !IsZero(pOld)))
        No++;
    }

  return (No);
}

#endif //PremiaCurrentVersion
```