

Help

```

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else

/*****
/*                                qmatrix.c                                */
/*****
/*                                */
/* type QMATRIX                                */
/*                                */
/* 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 COPYRIGHT.H)                      */
/*                                */
/*****

#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "laspack/qmatrix.h"
#include "laspack/errhandl.h"
#include "laspack/copyright.h"

static ElType ZeroEl = { 0, 0.0 };

static int ElCompar(const void *El1, const void *El2);

void Q_Constr(QMatrix *Q, char *Name, size_t Dim, Boolean Symmetry,
              ElOrderType ElOrder, InstanceType Instance, Boolean OwnData)
/* constructor of the type QMatrix */
{
    size_t RoC;

    Q->Name = (char *)malloc((strlen(Name) + 1) * sizeof(char));
    if (Q->Name != NULL)

```

```

    strcpy(Q->Name, Name);
else
    LASError(LASMemAllocErr, "Q_Constr", Name, NULL, NULL);
Q->Dim = Dim;
Q->Symmetry = Symmetry;
Q->ElOrder = ElOrder;
Q->Instance = Instance;
Q->LockLevel = 0;
Q->MultiplD = 1.0;
Q->MultiplU = 1.0;
Q->MultiplL = 1.0;
Q->OwnData = OwnData;
if (OwnData)
{
    if (LASResult() == LASOK)
    {
        Q->Len = (size_t *)malloc((Dim + 1) * sizeof(size_t));
        Q->El = (ElType **)malloc((Dim + 1) * sizeof(ElType *));
        Q->ElSorted = (Boolean *)malloc(sizeof(Boolean));
        Q->DiagElAlloc = (Boolean *)malloc(sizeof(Boolean));
        Q->DiagEl = (ElType **)malloc((Dim + 1) * sizeof(ElType *));
        Q->ZeroInDiag = (Boolean *)malloc(sizeof(Boolean));
        Q->InvDiagEl = (double *)malloc((Dim + 1) * sizeof(double));
        Q->ILUExists = (Boolean *)malloc(sizeof(Boolean));
        Q->ILU = (QMatrix *)malloc(sizeof(QMatrix));
        if (Q->Len != NULL && Q->El != NULL && Q->ElSorted != NULL
            && Q->DiagElAlloc != NULL && Q->DiagEl != NULL && Q->ZeroInDiag !=
            && Q->InvDiagEl != NULL && Q->ILUExists != NULL && Q->ILU != NULL)
        {
            for (RoC = 1; RoC <= Dim; RoC++)
            {
                Q->Len[RoC] = 0;
                Q->El[RoC] = NULL;
                Q->DiagEl[RoC] = NULL;
                Q->InvDiagEl[RoC] = 0.0;
            }
            *Q->ElSorted = False;
            *Q->DiagElAlloc = False;
            *Q->ZeroInDiag = True;
            *Q->ILUExists = False;
        }
    }
}

```

```

        else
        {
            LASError(LASMemAllocErr, "Q_Constr", Name, NULL, NULL);
        }
    }
else
{
    Q->Len = NULL;
    Q->E1 = NULL;
    Q->E1Sorted = NULL;
    Q->DiagE1Alloc = NULL;
    Q->DiagE1 = NULL;
    Q->ZeroInDiag = NULL;
    Q->InvDiagE1 = NULL;
    Q->ILUExists = NULL;
    Q->ILU = NULL;
}

}

Q->UnitRightKer = False;
Q->RightKerCmp = NULL;
Q->UnitLeftKer = False;
Q->LeftKerCmp = NULL;
Q->EigenvalInfo = NULL;
}

void Q_Destr(QMatrix *Q)
/* destructor of the type QMatrix */
{
    size_t Dim, RoC;

    if (Q->Name != NULL)
        free(Q->Name);
    Dim = Q->Dim;
    if (Q->OwnData)
    {
        if (Q->Len != NULL && Q->E1 != NULL)
        {
            for (RoC = 1; RoC <= Dim; RoC++)
            {
                if (Q->Len[RoC] > 0)
                {

```

```

        if (Q->El[RoC] != NULL)
            free(Q->El[RoC]);
    }
}
}
if (Q->Len != NULL)
{
    free(Q->Len);
    Q->Len = NULL;
}
if (Q->El != NULL)
{
    free(Q->El);
    Q->El = NULL;
}
if (Q->ElSorted != NULL)
{
    free(Q->ElSorted);
    Q->ElSorted = NULL;
}
if (Q->DiagElAlloc != NULL)
{
    free(Q->DiagElAlloc);
    Q->DiagElAlloc = NULL;
}
if (Q->DiagEl != NULL)
{
    free(Q->DiagEl);
    Q->DiagEl = NULL;
}
if (Q->ZeroInDiag != NULL)
{
    free(Q->ZeroInDiag);
    Q->ZeroInDiag = NULL;
}
if (Q->InvDiagEl != NULL)
{
    free(Q->InvDiagEl);
    Q->InvDiagEl = NULL;
}
if (Q->ILUExists != NULL && Q->ILU != NULL)

```

```

        {
            if (*Q->ILUExists)
                Q_Destr(Q->ILU);
        }
    if (Q->ILUExists != NULL)
    {
        free(Q->ILUExists);
        Q->ILUExists = NULL;
    }
    if (Q->ILU != NULL)
    {
        free(Q->ILU);
        Q->ILU = NULL;
    }
}

if (Q->RightKerCmp != NULL)
{
    free(Q->RightKerCmp);
    Q->RightKerCmp = NULL;
}

if (Q->LeftKerCmp != NULL)
{
    free(Q->LeftKerCmp);
    Q->LeftKerCmp = NULL;
}

if (Q->EigenvalInfo != NULL)
{
    free(Q->EigenvalInfo);
    Q->EigenvalInfo = NULL;
}

}

void Q_SetName(QMatrix *Q, char *Name)
/* (re)set name of the matrix Q */
{
    if (LASResult() == LASOK)
    {
        free(Q->Name);
        Q->Name = (char *)malloc((strlen(Name) + 1) * sizeof(char));
        if (Q->Name != NULL)
            strcpy(Q->Name, Name);
    }
}

```

```
        else
            LASError(LASMemAllocErr, "Q_SetName", Name, NULL, NULL);
    }
}

char *Q_GetName(QMatrix *Q)
/* returns the name of the matrix Q */
{
    if (LASResult() == LASOK)
        return (Q->Name);
    else
        return ("");
}

size_t Q_GetDim(QMatrix *Q)
/* returns the dimension of the matrix Q */
{
    size_t Dim;

    if (LASResult() == LASOK)
        Dim = Q->Dim;
    else
        Dim = 0;
    return (Dim);
}

Boolean Q_GetSymmetry(QMatrix *Q)
/* returns True if Q is symmetric otherwise False */
{
    Boolean Symmetry;

    if (LASResult() == LASOK)
    {
        Symmetry = Q->Symmetry;
    }
    else
    {
        Symmetry = (Boolean)0;
    }
    return (Symmetry);
}
```

```

ElOrderType Q_GetElOrder(QMatrix *Q)
/* returns element order of the matrix Q */
{
    ElOrderType ElOrder;

    if (LASResult() == LASOK)
    {
        ElOrder = Q->ElOrder;
    }
    else
    {
        ElOrder = (ElOrderType)0;
    }
    return (ElOrder);
}

void Q_SetLen(QMatrix *Q, size_t RoC, size_t Len)
/* set the length of a row or column of the matrix Q */
{
    size_t ElCount;
    ElType *PtrEl;

    if (LASResult() == LASOK)
    {
        if (Q->Instance == Normal && RoC > 0 && RoC <= Q->Dim)
        {
            Q->Len[RoC] = Len;

            PtrEl = Q->El[RoC];

            if (PtrEl != NULL)
            {
                free(PtrEl);
                PtrEl = NULL;
            }

            if (Len > 0)
            {
                PtrEl = (ElType *)malloc(Len * sizeof(ElType));
                Q->El[RoC] = PtrEl;
            }
        }
    }
}

```

```

        if (PtrEl != NULL)
        {
            for (ElCount = Len; ElCount > 0; ElCount--)
            {
                *PtrEl = ZeroEl;
                PtrEl++;
            }
        }
        else
        {
            LASError(LASMemAllocErr, "Q_SetLen", Q->Name, NULL, NULL);
        }
    }
    else
    {
        {
            Q->El[RoC] = NULL;
        }
    }
}
else
{
    if (Q->Instance != Normal)
        LASError(LASLValErr, "Q_SetLen", Q->Name, NULL, NULL);
    else
        LASError(LASRangeErr, "Q_SetLen", Q->Name, NULL, NULL);
}
}

size_t Q_GetLen(QMatrix *Q, size_t RoC)
/* returns the lenght of a row or column of the matrix Q */
{
    size_t Len;

    if (LASResult() == LASOK)
    {
        if (RoC > 0 && RoC <= Q->Dim)
        {
            Len = Q->Len[RoC];
        }
        else

```



```

        {
            LASError(LASRangeErr, "Q_GetLen", Q->Name, NULL, NULL);
            Len = 0;
        }
    }
else
    {
        Len = 0;
    }
return (Len);
}

void Q_SetEntry(QMatrix *Q, size_t RoC, size_t Entry, size_t Pos, double Val)
/* set a new matrix entry */
{
    if (LASResult() == LASOK)
    {
        if ((RoC > 0 && RoC <= Q->Dim && Pos > 0 && Pos <= Q->Dim) &&
            (Entry < Q->Len[RoC]))
        {
            Q->El[RoC][Entry].Pos = Pos;
            Q->El[RoC][Entry].Val = Val;
        }
        else
        {
            LASError(LASRangeErr, "Q_SetEntry", Q->Name, NULL, NULL);
        }
    }
}

size_t Q_GetPos(QMatrix *Q, size_t RoC, size_t Entry)
/* returns the position of a matrix element */
{
    size_t Pos;

    if (LASResult() == LASOK)
        if (RoC > 0 && RoC <= Q->Dim && Entry < Q->Len[RoC])
        {
            Pos = Q->El[RoC][Entry].Pos;
        }
    else

```

```

        {
            LASError(LASRangeErr, "Q_GetPos", Q->Name, NULL, NULL);
            Pos = 0;
        }
    else
        Pos = 0;
    return (Pos);
}

double Q_GetVal(QMatrix *Q, size_t RoC, size_t Entry)
/* returns the value of a matrix element */
{
    double Val;

    if (LASResult() == LASOK)
        if (RoC > 0 && RoC <= Q->Dim && Entry < Q->Len[RoC])
        {
            Val = Q->El[RoC][Entry].Val;
        }
    else
    {
        LASError(LASRangeErr, "Q_GetVal", Q->Name, NULL, NULL);
        Val = 0.0;
    }
    else
        Val = 0.0;
    return (Val);
}

void Q_AddVal(QMatrix *Q, size_t RoC, size_t Entry, double Val)
/* add a value to a matrix entry */
{
    if (LASResult() == LASOK)
    {
        if ((RoC > 0 && RoC <= Q->Dim) && (Entry < Q->Len[RoC]))
            Q->El[RoC][Entry].Val += Val;
        else
            LASError(LASRangeErr, "Q_AddVal", Q->Name, NULL, NULL);
    }
}

```

```

double Q_GetEl(QMatrix *Q, size_t Row, size_t Clm)
/* returns the value of a matrix element (all matrix elements are considered) */
{
    double Val;

    size_t Len, ElCount;
    ElType *PtrEl;

    if (LASResult() == LASOK)
    {
        if (Row > 0 && Row <= Q->Dim && Clm > 0 && Clm <= Q->Dim)
        {
            Val = 0.0;
            if (Q->Symmetry && Q->ElOrder == Rows)
            {
                if (Clm >= Row)
                {
                    Len = Q->Len[Row];
                    PtrEl = Q->El[Row];
                    for (ElCount = Len; ElCount > 0; ElCount--)
                    {
                        if ((*PtrEl).Pos == Clm)
                            Val = (*PtrEl).Val;
                        PtrEl++;
                    }
                }
                else
                {
                    Len = Q->Len[Clm];
                    PtrEl = Q->El[Clm];
                    for (ElCount = Len; ElCount > 0; ElCount--)
                    {
                        if ((*PtrEl).Pos == Row)
                            Val = (*PtrEl).Val;
                        PtrEl++;
                    }
                }
            }
            else if (Q->Symmetry && Q->ElOrder == Clmws)
            {
                if (Clm >= Row)

```

```

    {
        Len = Q->Len[C1m];
        PtrEl = Q->El[C1m];
        for (ElCount = Len; ElCount > 0; ElCount--)
        {
            if ((*PtrEl).Pos == Row)
                Val = (*PtrEl).Val;
            PtrEl++;
        }
    }
else
    {
        Len = Q->Len[Row];
        PtrEl = Q->El[Row];
        for (ElCount = Len; ElCount > 0; ElCount--)
        {
            if ((*PtrEl).Pos == C1m)
                Val = (*PtrEl).Val;
            PtrEl++;
        }
    }
}
else if (!Q->Symmetry && Q->ElOrder == Rowws)
{
    Len = Q->Len[Row];
    PtrEl = Q->El[Row];
    for (ElCount = Len; ElCount > 0; ElCount--)
    {
        if ((*PtrEl).Pos == C1m)
            Val = (*PtrEl).Val;
        PtrEl++;
    }
}
else if (!Q->Symmetry && Q->ElOrder == Clmws)
{
    Len = Q->Len[C1m];
    PtrEl = Q->El[C1m];
    for (ElCount = Len; ElCount > 0; ElCount--)
    {
        if ((*PtrEl).Pos == Row)
            Val = (*PtrEl).Val;
    }
}

```

```

        PtrEl++;
    }
}

    if (Row == Clmws)
        Val *= Q->MultiplD;
    if (Row > Clm)
        Val *= Q->MultiplU;
    if (Row > Clm)
        Val *= Q->MultiplL;
}
else
{
    LASError(LASRangeErr, "Q_GetEl", Q->Name, NULL, NULL);
    Val = 0.0;
}
}
else
{
    Val = 0.0;
}
return (Val);
}

void Q_SortEl(QMatrix *Q)
/* sorts elements of a row or column in ascended order */
{
    size_t Dim, RoC;
    Boolean UpperOnly;

    if (LASResult() == LASOK && !(*Q->ElSorted))
    {
        Dim = Q->Dim;
        UpperOnly = True;
        for (RoC = 1; RoC <= Dim; RoC++)
        {
            /* sort of elements by the quick sort algorithms */
            qsort((void *)Q->El[RoC], Q->Len[RoC], sizeof(ElType), ElCompar);

            /* test whether elements contained in upper triangular part
               (incl. diagonal) of the matrix only */

```

```

        if (Q->ElOrder == Rowws)
        {
            if (Q->El[RoC][0].Pos < RoC)
                UpperOnly = False;
        }
        if (Q->ElOrder == Clmws)
        {
            if (Q->El[RoC][Q->Len[RoC] - 1].Pos > RoC)
                UpperOnly = False;
        }
    }

    *Q->ElSorted = True;
    *Q->DiagElAlloc = False;
    *Q->ZeroInDiag = True;

    if (Q->Symmetry)
    {
        if (!UpperOnly)
            LASError(LASSymStorErr, "Q_SortEl", Q->Name, NULL, NULL);
    }
}

void Q_AllocInvDiagEl(QMatrix *Q)
/* allocate pointers and compute inverse for diagonal elements of the matrix Q */
{
    size_t Dim, RoC, Len, ElCount;
    Boolean Found;
    ElType *PtrEl;

    if (LASResult() == LASOK && !(*Q->DiagElAlloc))
    {
        Dim = Q->Dim;
        *Q->ZeroInDiag = False;
        if (Q->Symmetry && Q->ElOrder == Rowws)
        {
            for (RoC = 1; RoC <= Dim; RoC++)
            {
                if (Q->El[RoC][0].Pos == RoC)
                {

```

```

        Q->DiagEl[RoC] = Q->El[RoC];
    }
    else
    {
        *Q->ZeroInDiag = True;
        Q->DiagEl[RoC] = &ZeroEl;
    }
}

}
if (Q->Symmetry && Q->ElOrder == Clmws)
{
    for (RoC = 1; RoC <= Dim; RoC++)
    {
        Len = Q->Len[RoC];
        if (Q->El[RoC][Len - 1].Pos == RoC)
        {
            Q->DiagEl[RoC] = Q->El[RoC] + Len - 1;
        }
        else
        {
            *Q->ZeroInDiag = True;
            Q->DiagEl[RoC] = &ZeroEl;
        }
    }
}
if (!Q->Symmetry)
{
    for (RoC = 1; RoC <= Dim; RoC++)
    {
        Found = False;
        Len = Q->Len[RoC];
        PtrEl = Q->El[RoC] + Len - 1;
        for (ElCount = Len; ElCount > 0; ElCount--)
        {
            if ((*PtrEl).Pos == RoC)
            {
                Found = True;
                Q->DiagEl[RoC] = PtrEl;
            }
            PtrEl--;
        }
    }
}

```

```

        if (!Found)
        {
            *Q->ZeroInDiag = True;
            Q->DiagEl[RoC] = &ZeroEl;
        }
    }
    *Q->DiagElAlloc = True;

    if (!(*Q->ZeroInDiag))
    {
        for (RoC = 1; RoC <= Dim; RoC++)
            Q->InvDiagEl[RoC] = 1.0 / (*Q->DiagEl[RoC]).Val;
    }
}

static int ElCompar(const void *El1, const void *El2)
/* compares positions of two matrix elements */
{
    int Compar;

    Compar = 0;
    if (((ElType *)El1)->Pos < ((ElType *)El2)->Pos)
        Compar = -1;
    if (((ElType *)El1)->Pos > ((ElType *)El2)->Pos)
        Compar = +1;

    return (Compar);
}

void Q_SetKer(QMatrix *Q, Vector *RightKer, Vector *LeftKer)
/* defines the null space in the case of a singular matrix */
{
    double Sum, Mean, Cmp, Norm;
    size_t Dim, Ind;
    double *KerCmp;

    V_Lock(RightKer);
    V_Lock(LeftKer);

```



```

if (LASResult() == LASOK)
{
    if (Q->Dim == RightKer->Dim && (Q->Symmetry || Q->Dim == LeftKer->Dim))
    {
        Dim = Q->Dim;

        /* release array for old null space components when it exists */
        if (Q->RightKerCmp != NULL)
        {
            free(Q->RightKerCmp);
            Q->RightKerCmp = NULL;
        }
        if (Q->LeftKerCmp != NULL)
        {
            free(Q->LeftKerCmp);
            Q->LeftKerCmp = NULL;
        }
        Q->UnitRightKer = False;
        Q->UnitLeftKer = False;

        /* right null space */
        KerCmp = RightKer->Cmp;
        /* test whether the matrix Q has a unit right null space */
        Sum = 0.0;
        for (Ind = 1; Ind <= Dim; Ind++)
            Sum += KerCmp[Ind];
        Mean = Sum / (double)Dim;
        Q->UnitRightKer = True;
        if (!IsZero(Mean))
        {
            for (Ind = 1; Ind <= Dim; Ind++)
                if (!IsOne(KerCmp[Ind] / Mean))
                    Q->UnitRightKer = False;
        }
    }
    else
    {
        Q->UnitRightKer = False;
    }
    if (!Q->UnitRightKer)
    {
        Sum = 0.0;
    }
}

```

```

for (Ind = 1; Ind <= Dim; Ind++)
{
    Cmp = KerCmp[Ind];
    Sum += Cmp * Cmp;
}
Norm = sqrt(Sum);
if (!IsZero(Norm))
{
    Q->RightKerCmp = (double *)malloc((Dim + 1) * sizeof(double));
    if (Q->RightKerCmp != NULL)
    {
        for (Ind = 1; Ind <= Dim; Ind++)
            Q->RightKerCmp[Ind] = KerCmp[Ind] / Norm;
    }
    else
    {
        LASError(LASMemAllocErr, "Q_SetKer", Q->Name, RightKer->Name,
                LeftKer->Name);
    }
}
}

if (!Q->Symmetry)
{
    /* left null space */
    KerCmp = LeftKer->Cmp;
    /* test whether the matrix Q has a unit left null space */
    Sum = 0.0;
    for (Ind = 1; Ind <= Dim; Ind++)
        Sum += KerCmp[Ind];
    Mean = Sum / (double)Dim;
    Q->UnitLeftKer = True;
    if (!IsZero(Mean))
    {
        for (Ind = 1; Ind <= Dim; Ind++)
            if (!IsOne(KerCmp[Ind] / Mean))
                Q->UnitLeftKer = False;
    }
    else
    {
        Q->UnitLeftKer = False;
    }
}

```

```

    }
    if (!Q->UnitLeftKer)
    {
        Sum = 0.0;
        for (Ind = 1; Ind <= Dim; Ind++)
        {
            Cmp = KerCmp[Ind];
            Sum += Cmp * Cmp;
        }
        Norm = sqrt(Sum);
        if (!IsZero(Norm))
        {
            Q->LeftKerCmp = (double *)malloc((Dim + 1) * sizeof(double));
            if (Q->LeftKerCmp != NULL)
            {
                for (Ind = 1; Ind <= Dim; Ind++)
                    Q->LeftKerCmp[Ind] = KerCmp[Ind] / Norm;
            }
            else
            {
                LASError(LASMemAllocErr, "Q_SetKer", Q->Name,
                        RightKer->Name, LeftKer->Name);
            }
        }
    }
}
else
{
}
}
else
{
    LASError(LASDimErr, "Q_SetKer", Q->Name, RightKer->Name, LeftKer->Name);
}
}

V_Unlock(RightKer);
V_Unlock(LeftKer);
}

Boolean Q_KerDefined(QMatrix *Q)

```

```

/* returns True if Q is singular and the null space has been defined
   otherwise False */
{
    Boolean KerDefined;

    if (LASResult() == LASOK)
    {
        if ((Q->UnitRightKer || Q->RightKerCmp != NULL) && !IsZero(Q->MultiplD)
            && IsOne(Q->MultiplU / Q->MultiplD) && IsOne(Q->MultiplL / Q->MultiplD))
            KerDefined = True;
        else
            KerDefined = False;
    }
    else
    {
        KerDefined = (Boolean)0;
    }
    return (KerDefined);
}

void **Q_EigenvalInfo(QMatrix *Q)
/* return address of the infos for eigenvalues */
{
    return (&(Q->EigenvalInfo));
}

void Q_Lock(QMatrix *Q)
/* lock the matrix Q */
{
    if (Q != NULL)
        Q->LockLevel++;
}

void Q_Unlock(QMatrix *Q)
/* unlock the matrix Q */
{
    if (Q != NULL)
    {
        Q->LockLevel--;
        if (Q->Instance == Tempor && Q->LockLevel <= 0)
        {

```

```
        Q_Destr(Q);
        free(Q);
    }
}

#endif //PremiaCurrentVersion
```