Help

```
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2008+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
#else
/*********************************************************
    *****************/
/*                                  matrix.c
                    */
/*********************************************************
    *****************/
/*
                    */
/* type MATRIX
                    */
/*
                    */
/* Copyright (C) 1992-1995 Tomas Skalicky. All rights res
    erved.            */
/*
                    */
/*********************************************************
    *****************/
/*
                    */
/*         ANY USE OF THIS CODE CONSTITUTES ACCEPTANCE OF TH
    E TERMS           */
/*               OF THE COPYRIGHT NOTICE (SEE FILE copyrght.h
    )                */
/*
                    */
/*********************************************************
    *****************/

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

#include "laspack/highdim_matrix.h"
#include "laspack/errhandl.h"
```

```
#include "laspack/copyrght.h"

static ElType ZeroEl = { 0, 0.0 };

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

void M_Constr(Matrix *M, char *Name, size_t RowDim, size_t
    ClmDim,
              ElOrderType ElOrder, InstanceType Instance,
    Boolean OwnData)
/* constructor of the type Matrix */
{
    size_t Dim, RoC;

    M->Name = (char *)malloc((strlen(Name) + 1) * sizeof(
    char));
    if (M->Name != NULL)
        strcpy(M->Name, Name);
    else
        LASError(LASMemAllocErr, "M_Constr", Name, NULL,
    NULL);
    M->RowDim = RowDim;
    M->ClmDim = ClmDim;
    M->ElOrder = ElOrder;
    M->Instance = Instance;
    M->LockLevel = 0;
    M->Multipl = 1.0;
    M->OwnData = OwnData;
    if (OwnData) {
        if (LASResult() == LASOK) {
            if (ElOrder == Rowws)
                Dim = RowDim;
            else
                Dim = ClmDim;
      M->Len = (size_t *)malloc((Dim + 1) * sizeof(size_t)
    );
      M->El = (ElType **)malloc((Dim + 1) * sizeof(ElType
    *));
      M->ElSorted = (Boolean *)malloc(sizeof(Boolean));
      if (M->Len != NULL && M->El != NULL) {
                for (RoC = 1; RoC <= Dim; RoC++) {
                    M->Len[RoC] = 0;
                    M->El[RoC] = NULL;
                }
                *M->ElSorted = False;
            } else {
          LASError(LASMemAllocErr, "M_Constr", Name, NULL,
     NULL);
            }
        } else {
      M->Len = NULL;
      M->El = NULL;
      M->ElSorted = NULL;
        }
    }
}

void M_Destr(Matrix *M)
/* destructor of the type Matrix */
{
    size_t Dim, RoC;

    if (M->Name != NULL)
        free(M->Name);
    if (M->ElOrder == Rowws)
        Dim = M->RowDim;
    else
        Dim = M->ClmDim;
    if (M->OwnData) {
  if (M->Len != NULL && M->El != NULL) {
            for (RoC = 1; RoC <= Dim; RoC++) {
                if (M->Len[RoC] > 0) {
                    if (M->El[RoC] != NULL)
                        free(M->El[RoC]);
                }
            }
        }
        if (M->Len != NULL) {
            free(M->Len);
            M->Len = NULL;
        }
        if (M->El != NULL) {
            free(M->El);
            M->El = NULL;
        }
        if (M->ElSorted != NULL) {
            free(M->ElSorted);
            M->ElSorted = NULL;
        }
    }
}

void M_SetName(Matrix *M, char *Name)
/* (re)set name of the matrix M */
{
    if (LASResult() == LASOK) {
        free(M->Name);
        M->Name = (char *)malloc((strlen(Name) + 1) * size
    of(char));
        if (M->Name != NULL)
            strcpy(M->Name, Name);
        else
            LASError(LASMemAllocErr, "M_SetName", Name, NUL
    L, NULL);
    }
}

char *M_GetName(Matrix *M)
/* returns the name of the matrix M */
{
    if (LASResult() == LASOK)
        return(M->Name);
    else
        return("");
}

size_t M_GetRowDim(Matrix *M)
/* returns the row dimension of the matrix M */
{
    size_t Dim;

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

size_t M_GetClmDim(Matrix *M)
/* returns the column dimension of the matrix M */
{
    size_t Dim;

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

ElOrderType M_GetElOrder(Matrix *M)
/* returns the element order */
{
    ElOrderType ElOrder;

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

void M_SetLen(Matrix *M, size_t RoC, size_t Len)
/* set the length of a row or column of the matrix M */
{
    size_t ElCount;
    ElType *PtrEl;

    if (LASResult() == LASOK) {
        if (M->Instance == Normal
            && ((M->ElOrder == Rowws && RoC > 0 && RoC <=
    M->RowDim)
            || (M->ElOrder == Clmws && RoC > 0 && RoC <= M-
    >ClmDim))) {
            M->Len[RoC] = Len;

            PtrEl = M->El[RoC];

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

            if (Len > 0) {
                PtrEl = (ElType *)malloc(Len * sizeof(ElTyp
    e));
                M->El[RoC] = PtrEl;

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

size_t M_GetLen(Matrix *M, size_t RoC)
/* returns the length of a row or column of the matrix M */
```

```
{
    size_t Len;

    if (LASResult() == LASOK) {
        if ((M->ElOrder == Rowws && RoC > 0 && RoC <= M->Ro
    wDim) ||
            (M->ElOrder == Clmws && RoC > 0 && RoC <= M->
    ClmDim)) {
            Len = M->Len[RoC];
        } else {
            LASError(LASRangeErr, "M_GetLen", M->Name, NUL
    L, NULL);
            Len = 0;
        }
    } else {
        Len = 0;
    }
    return(Len);
}

void M_SetEntry(Matrix *M, size_t RoC, size_t Entry, size_
    t Pos, double Val)
/* set a new matrix entry */
{
    if (LASResult() == LASOK) {
        if ((M->ElOrder == Rowws && RoC > 0 && RoC <= M->Ro
    wDim && Pos > 0 && Pos <= M->ClmDim) ||
            ((M->ElOrder == Clmws && RoC > 0 && RoC <= M->
    ClmDim && Pos > 0 && Pos <= M->RowDim) &&
            (Entry < M->Len[RoC]))) {
            M->El[RoC][Entry].Val = Val;
            M->El[RoC][Entry].Pos = Pos;
        } else {
            LASError(LASRangeErr, "M_SetEntry", M->Name,
    NULL, NULL);
        }
    }
}

size_t M_GetPos(Matrix *M, size_t RoC, size_t Entry)
/* returns the position of a matrix entry */
```

```
{
    size_t Pos;

    if (LASResult() == LASOK)
        if ((M->ElOrder == Rowws && RoC > 0 && RoC <= M->Ro
    wDim) ||
            ((M->ElOrder == Clmws && RoC > 0 && RoC <= M->
    ClmDim) &&
            (Entry < M->Len[RoC])) {
            Pos = M->El[RoC][Entry].Pos;
        } else {
            LASError(LASRangeErr, "M_GetPos", M->Name, NUL
    L, NULL);
      Pos = 0;
        }
    else
        Pos = 0;
    return(Pos);
}

double M_GetVal(Matrix *M, size_t RoC, size_t Entry)
/* returns the value of a matrix entry */
{
    double Val;

    if (LASResult() == LASOK)
        if ((M->ElOrder == Rowws && RoC > 0 && RoC <= M->Ro
    wDim) ||
            ((M->ElOrder == Clmws && RoC > 0 && RoC <= M->
    ClmDim) &&
            (Entry < M->Len[RoC])) {
            Val = M->El[RoC][Entry].Val;
        } else {
            LASError(LASRangeErr, "M_GetVal", M->Name, NUL
    L, NULL);
      Val = 0.0;
  }
    else
        Val = 0.0;
    return(Val);
}
```

```
void M_AddVal(Matrix *M, size_t RoC, size_t Entry, double
    Val)
/* add a value to a matrix entry */
{
    if (LASResult() == LASOK) {
        if ((M->ElOrder == Rowws && RoC > 0 && RoC <= M->Ro
    wDim) ||
            ((M->ElOrder == Clmws && RoC > 0 && RoC <= M->
    ClmDim) &&
            (Entry < M->Len[RoC]))
            M->El[RoC][Entry].Val += Val;
        else
            LASError(LASRangeErr, "M_AddVal", M->Name, NUL
    L, NULL);
    }
}

double M_GetEl(Matrix *M, size_t Row, size_t Clm)
/* returns the value of a matrix element (all matrix elemen
    ts are considered) */
{
    double Val;

    size_t Len, ElCount;
    ElType *PtrEl;

    if (LASResult() == LASOK) {
        if (Row > 0 && Row <= M->RowDim && Clm > 0 && Clm <
    = M->ClmDim) {
            Val = 0.0;
            if (M->ElOrder == Rowws) {
                Len = M->Len[Row];
                PtrEl = M->El[Row];
                for (ElCount = Len; ElCount > 0; ElCount--)
     {
                    if ((*PtrEl).Pos == Clm)
                        Val = (*PtrEl).Val;
                    PtrEl++;
                }
            } else if (M->ElOrder == Clmws) {
                Len = M->Len[Clm];
                PtrEl = M->El[Clm];
                for (ElCount = Len; ElCount > 0; ElCount--)
     {
                    if ((*PtrEl).Pos == Row)
                        Val = (*PtrEl).Val;
                    PtrEl++;
                }
            }
        } else {
            LASError(LASRangeErr, "M_GetEl", M->Name, NULL,
     NULL);
            Val = 0.0;
        }
    } else {
        Val = 0.0;
    }
    return(Val);
}

void M_SortEl(Matrix *M)
/* sorts elements of a row or column in ascended order */
{
    size_t Dim = 0, RoC;

    if (LASResult() == LASOK && !(*M->ElSorted)) {
        if (M->ElOrder == Rowws)
      Dim = M->ClmDim;
        if (M->ElOrder == Clmws)
      Dim = M->ClmDim;
        for (RoC = 1; RoC <= Dim; RoC++) {
            /* sort of elements by the quick sort algorith
    ms */
            qsort((void *)M->El[RoC], M->Len[RoC], sizeof(
    ElType), ElCompar);
        }

        *M->ElSorted = True;
    }
}
```

```
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 M_Lock(Matrix *M)
/* lock the matrix M */
{
    if (M != NULL)
        M->LockLevel++;
}

void M_Unlock(Matrix *M)
/* unlock the matrix M */
{
    if (M != NULL) {
        M->LockLevel--;
        if (M->Instance == Tempor && M->LockLevel <= 0) {
            M_Destr(M);
      free(M);
  }
    }
}

#endif //PremiaCurrentVersion
```

References