Help

```
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else
/*****************************************************************************/
/*                                  operats.c                                */
/*****************************************************************************/
/*                                                                           */
/* basic OPERATionS for the types vector, matrix and 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 COPYRGHT.H)                */
/*                                                                           */
/*****************************************************************************/

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

#include "
href../../../common/math/highdim_solver/laspack/operats_h_src.pdflaspack/operats
#include "
href../../../common/math/highdim_solver/laspack/errhandl_h_src.pdflaspack/errhan
#include "
href../../../common/math/highdim_solver/laspack/copyrght_h_src.pdflaspack/copyrg

/* the following macro allows to optimize vector operations
 for different computer architecture */
#if !defined(__hppa) && !defined(sparc)
#define for_AllCmp for(Ind = 1; Ind <= Dim; Ind++)
#else
#define for_AllCmp for(Ind = Dim; Ind > 0; Ind--)
#endif

Vector *Asgn_VV(Vector *V1, Vector *V2)
/* VRes = V1 = V2 */
```

```
{
  Vector *VRes;

  double Multipl;
  size_t Dim, Ind;
  double *V1Cmp, *V2Cmp;

  V_Lock(V1);
  V_Lock(V2);

  if (LASResult() == LASOK)
    {
      if (V1->Instance == Normal && V1->Dim == V2->Dim)
        {
          V1Cmp = V1->Cmp;
          V2Cmp = V2->Cmp;
          if (V2->Instance == Normal)
            {
              Dim = V1->Dim;
              for_AllCmp
              V1Cmp[Ind] = V2Cmp[Ind];
            }
          else
            {
              if (V2->OwnData)
                {
                  V1->Cmp = V2Cmp;
                  V2->Cmp = V1Cmp;
                }
              else
                {
                  if (IsOne(V2->Multipl))
                    {
                      Dim = V1->Dim;
                      for_AllCmp
                      V1Cmp[Ind] = V2Cmp[Ind];
                    }
                  else
                    {
                      Dim = V1->Dim;
                      Multipl = V2->Multipl;
```

```
                    for_AllCmp
                    V1Cmp[Ind] = Multipl * V2Cmp[Ind];
                  }
              }
            }
          VRes = V1;
        }
      else
        {
          if (V1->Instance != Normal)
            LASError(LASLValErr, "Asgn_VV", V_GetName(V1), V_GetName(V2), NULL);
          if (V1->Dim != V2->Dim)
            LASError(LASDimErr, "Asgn_VV", V_GetName(V1), V_GetName(V2), NULL);
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (VRes);
}

Vector *AddAsgn_VV(Vector *V1, Vector *V2)
/* VRes = V1 += V2 */
{
  Vector *VRes;

  double Multipl;
  size_t Dim, Ind;
  double *V1Cmp, *V2Cmp;

  V_Lock(V1);
  V_Lock(V2);

  if (LASResult() == LASOK)
    {
```

```
      if (V1->Instance == Normal && V1->Dim == V2->Dim)
        {
          Dim = V1->Dim;
          V1Cmp = V1->Cmp;
          V2Cmp = V2->Cmp;
          if (IsOne(V2->Multipl))
            {
              for_AllCmp
              V1Cmp[Ind] += V2Cmp[Ind];
            }
          else
            {
              Multipl = V2->Multipl;
              for_AllCmp
              V1Cmp[Ind] += Multipl * V2Cmp[Ind];
            }
          VRes = V1;
        }
      else
        {
          if (V1->Instance != Normal)
            LASError(LASLValErr, "AddAsgn_VV", V_GetName(V1), V_GetName(V2), NUL
          if (V1->Dim != V2->Dim)
            LASError(LASDimErr, "AddAsgn_VV", V_GetName(V1), V_GetName(V2), NULL
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (VRes);
}

Vector *SubAsgn_VV(Vector *V1, Vector *V2)
/* VRes = V1 -= V2 */
{
```

```
Vector *VRes;

double Multipl;
size_t Dim, Ind;
double *V1Cmp, *V2Cmp;

V_Lock(V1);
V_Lock(V2);

if (LASResult() == LASOK)
  {
    if (V1->Instance == Normal && V1->Dim == V2->Dim)
      {
        Dim = V1->Dim;
        V1Cmp = V1->Cmp;
        V2Cmp = V2->Cmp;
        if (IsOne(V2->Multipl))
          {
            for_AllCmp
            V1Cmp[Ind] -= V2Cmp[Ind];
          }
        else
          {
            Multipl = V2->Multipl;
            for_AllCmp
            V1Cmp[Ind] -= Multipl * V2Cmp[Ind];
          }
        VRes = V1;
      }
    else
      {
        if (V1->Instance != Normal)
          LASError(LASLValErr, "SubAsgn_VV", V_GetName(V1), V_GetName(V2), NUL
        if (V1->Dim != V2->Dim)
          LASError(LASDimErr, "SubAsgn_VV", V_GetName(V1), V_GetName(V2), NULL
        VRes = NULL;
      }
  }
else
  {
    VRes = NULL;
```

```
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (VRes);
}

Vector *MulAsgn_VS(Vector *V, double S)
/* VRes = V *= S */
{
  Vector *VRes;

  size_t Dim, Ind;
  double *VCmp;

  V_Lock(V);

  if (LASResult() == LASOK)
    {
      if (V->Instance == Normal)
        {
          Dim = V->Dim;
          VCmp = V->Cmp;
          for_AllCmp
          VCmp[Ind] *= S;
          VRes = V;
        }
      else
        {
          LASError(LASLValErr, "MulAsgn_VS", V_GetName(V), NULL, NULL);
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V);
```

```
  return (VRes);
}

Vector *Add_VV(Vector *V1, Vector *V2)
/* VRes = V1 + V2 */
{
  Vector *VRes;

  char *VResName;
  double Multipl1, Multipl2;
  size_t Dim, Ind;
  double *V1Cmp, *V2Cmp, *VResCmp;

  V_Lock(V1);
  V_Lock(V2);

  if (LASResult() == LASOK)
    {
      if (V1->Dim == V2->Dim)
        {
          Dim = V1->Dim;
          VRes = (Vector *)malloc(sizeof(Vector));
          VResName = (char *)malloc((strlen(V_GetName(V1)) + strlen(V_GetName(V2
                                    * sizeof(char));
          if (VRes != NULL && VResName != NULL)
            {
              sprintf(VResName, "%s + %s", V_GetName(V1), V_GetName(V2));
              V_Constr(VRes, VResName, Dim, Tempor, True);
              if (LASResult() == LASOK)
                {
                  V1Cmp = V1->Cmp;
                  V2Cmp = V2->Cmp;
                  VResCmp = VRes->Cmp;
                  if (IsOne(V1->Multipl))
                    {
                      if (IsOne(V2->Multipl))
                        {
                          for_AllCmp
                          VResCmp[Ind] = V1Cmp[Ind] + V2Cmp[Ind];
                        }
                      else
```

```
                    {
                      Multipl2 = V2->Multipl;
                      for_AllCmp
                      VResCmp[Ind] = V1Cmp[Ind] + Multipl2 * V2Cmp[Ind];
                    }
                }
              else
                {
                  Multipl1 = V1->Multipl;
                  if (IsOne(V2->Multipl))
                    {
                      for_AllCmp
                      VResCmp[Ind] = Multipl1 * V1Cmp[Ind] + V2Cmp[Ind];
                    }
                  else
                    {
                      Multipl2 = V2->Multipl;
                      for_AllCmp
                      VResCmp[Ind] = Multipl1 * V1Cmp[Ind] + Multipl2 * V2Cm
                    }
                }
            }
          else
            {
              LASError(LASMemAllocErr, "Add_VV", V_GetName(V1), V_GetName(V2), N
              if (VRes != NULL)
                free(VRes);
            }

          if (VResName != NULL)
            free(VResName);
        }
      else
        {
          LASError(LASDimErr, "Add_VV", V_GetName(V1), V_GetName(V2), NULL);
          VRes = NULL;
        }
    }
  else
    {
```

```
      VRes = NULL;
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (VRes);
}

QMatrix *Add_QQ(QMatrix *Q1, QMatrix *Q2)
/* QRes = Q1 + Q2, this operation is limited to matrices, which are
 derived from the same matrix */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q1);
  Q_Lock(Q2);

  if (LASResult() == LASOK)
    {
      if (Q1->Dim == Q2->Dim)
        {
          if (Q1->Symmetry == Q2->Symmetry &&
              Q1->ElOrder == Q2->ElOrder &&
              Q1->Len == Q2->Len &&
              Q1->El == Q2->El)
            {
              QRes = (QMatrix *)malloc(sizeof(QMatrix));
              QResName = (char *)malloc((strlen(Q_GetName(Q1)) + strlen(Q_GetNam
                                         * sizeof(char));
              if (QRes != NULL && QResName != NULL)
                {
                  sprintf(QResName, "%s + %s", Q_GetName(Q1), Q_GetName(Q2));
                  Q_Constr(QRes, QResName, Q1->Dim, Q1->Symmetry, Q1->ElOrder, T
                  if (LASResult() == LASOK)
                    {
                      QRes->MultiplD = Q1->MultiplD + Q2->MultiplD;
                      QRes->MultiplU = Q1->MultiplU + Q2->MultiplU;
                      QRes->MultiplL = Q1->MultiplL + Q2->MultiplL;
```

```
                    QRes->Len = Q1->Len;
                    QRes->El = Q1->El;
                    QRes->ElSorted = Q1->ElSorted;
                    QRes->DiagElAlloc = Q1->DiagElAlloc;
                    QRes->DiagEl = Q1->DiagEl;
                    QRes->ZeroInDiag = Q1->ZeroInDiag;
                    QRes->InvDiagEl = Q1->InvDiagEl;
                    QRes->UnitRightKer = Q1->UnitRightKer;
                    QRes->RightKerCmp = Q1->RightKerCmp;
                    QRes->UnitLeftKer = Q1->UnitLeftKer;
                    QRes->LeftKerCmp = Q1->LeftKerCmp;
                    QRes->ILUExists = Q1->ILUExists;
                    QRes->ILU = Q1->ILU;
                  }
              }
            else
              {
                LASError(LASMemAllocErr, "Add_QQ", Q_GetName(Q1), Q_GetName(Q2
                if (QRes != NULL)
                  free(QRes);
              }

            if (QResName != NULL)
              free(QResName);
          }
        else
          {
            LASError(LASMatrCombErr, "Add_QQ", Q_GetName(Q1), Q_GetName(Q2), N
            QRes = NULL;
          }
      }
    else
      {
        LASError(LASDimErr, "Add_QQ", Q_GetName(Q1), Q_GetName(Q2), NULL);
        QRes = NULL;
      }
  }
else
  {
    QRes = NULL;
  }
```

```
  Q_Unlock(Q1);
  Q_Unlock(Q2);

  return (QRes);
}

Vector *Sub_VV(Vector *V1, Vector *V2)
/* VRes = V1 - V2 */
{
  Vector *VRes;

  char *VResName;
  double Multipl1, Multipl2;
  size_t Dim, Ind;
  double *V1Cmp, *V2Cmp, *VResCmp;

  V_Lock(V1);
  V_Lock(V2);

  if (LASResult() == LASOK)
    {
      if (V1->Dim == V2->Dim)
        {
          Dim = V1->Dim;
          VRes = (Vector *)malloc(sizeof(Vector));
          VResName = (char *)malloc((strlen(V_GetName(V1)) + strlen(V_GetName(V2
                                    * sizeof(char));
          if (VRes != NULL && VResName != NULL)
            {
              sprintf(VResName, "%s - %s", V_GetName(V1), V_GetName(V2));
              V_Constr(VRes, VResName, Dim, Tempor, True);
              if (LASResult() == LASOK)
                {
                  V1Cmp = V1->Cmp;
                  V2Cmp = V2->Cmp;
                  VResCmp = VRes->Cmp;
                  if (IsOne(V1->Multipl))
                    {
                      if (IsOne(V2->Multipl))
                        {
```

```
                    for_AllCmp
                    VResCmp[Ind] = V1Cmp[Ind] - V2Cmp[Ind];
                  }
                else
                  {
                    Multipl2 = V2->Multipl;
                    for_AllCmp
                    VResCmp[Ind] = V1Cmp[Ind] - Multipl2 * V2Cmp[Ind];
                  }
              }
            else
              {
                Multipl1 = V1->Multipl;
                if (IsOne(V2->Multipl))
                  {
                    for_AllCmp
                    VResCmp[Ind] = Multipl1 * V1Cmp[Ind] - V2Cmp[Ind];
                  }
                else
                  {
                    Multipl2 = V2->Multipl;
                    for_AllCmp
                    VResCmp[Ind] = Multipl1 * V1Cmp[Ind] - Multipl2 * V2Cm
                  }
              }
          }
        else
          {
            LASError(LASMemAllocErr, "Sub_VV", V_GetName(V1), V_GetName(V2), N
            if (VRes != NULL)
              free(VRes);
          }

        if (VResName != NULL)
          free(VResName);
      }
    else
      {
        LASError(LASDimErr, "Sub_VV", V_GetName(V1), V_GetName(V2), NULL);
        VRes = NULL;
```

```
        }
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (VRes);
}

QMatrix *Sub_QQ(QMatrix *Q1, QMatrix *Q2)
/* QRes = Q1 - Q2, this operation ist limited to matricies, which are
 derived from the same matrix */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q1);
  Q_Lock(Q2);

  if (LASResult() == LASOK)
    {
      if (Q1->Dim == Q2->Dim)
        {
          if (Q1->Symmetry == Q2->Symmetry &&
              Q1->ElOrder == Q2->ElOrder &&
              Q1->Len == Q2->Len &&
              Q1->El == Q2->El)
            {
              QRes = (QMatrix *)malloc(sizeof(QMatrix));
              QResName = (char *)malloc((strlen(Q_GetName(Q1)) + strlen(Q_GetNam
                                         * sizeof(char));
              if (QRes != NULL && QResName != NULL)
                {
                  sprintf(QResName, "%s - %s", Q_GetName(Q1), Q_GetName(Q2));
                  Q_Constr(QRes, QResName, Q1->Dim, Q1->Symmetry, Q1->ElOrder, T
                  if (LASResult() == LASOK)
```

```
                {
                  QRes->MultiplD = Q1->MultiplD - Q2->MultiplD;
                  QRes->MultiplU = Q1->MultiplU - Q2->MultiplU;
                  QRes->MultiplL = Q1->MultiplL - Q2->MultiplL;
                  QRes->Len = Q1->Len;
                  QRes->El = Q1->El;
                  QRes->ElSorted = Q1->ElSorted;
                  QRes->DiagElAlloc = Q1->DiagElAlloc;
                  QRes->DiagEl = Q1->DiagEl;
                  QRes->ZeroInDiag = Q1->ZeroInDiag;
                  QRes->InvDiagEl = Q1->InvDiagEl;
                  QRes->UnitRightKer = Q1->UnitRightKer;
                  QRes->RightKerCmp = Q1->RightKerCmp;
                  QRes->UnitLeftKer = Q1->UnitLeftKer;
                  QRes->LeftKerCmp = Q1->LeftKerCmp;
                  QRes->ILUExists = Q1->ILUExists;
                  QRes->ILU = Q1->ILU;
                }
            }
          else
            {
              LASError(LASMemAllocErr, "Sub_QQ", Q_GetName(Q1), Q_GetName(Q2
              if (QRes != NULL)
                free(QRes);
            }

          if (QResName != NULL)
            free(QResName);
        }
      else
        {
          LASError(LASMatrCombErr, "Sub_QQ", Q_GetName(Q1), Q_GetName(Q2), N
          QRes = NULL;
        }
    }
  else
    {
      LASError(LASDimErr, "Sub_QQ", Q_GetName(Q1), Q_GetName(Q2), NULL);
      QRes = NULL;
    }
}
```

```
  else
    {
      QRes = NULL;
    }

  Q_Unlock(Q1);
  Q_Unlock(Q2);

  return (QRes);
}

Vector *Mul_SV(double S, Vector *V)
/* VRes = S * V */
{
  Vector *VRes;

  char *VResName;

  V_Lock(V);

  if (LASResult() == LASOK)
    {
      VRes = (Vector *)malloc(sizeof(Vector));
      VResName = (char *)malloc((strlen(V_GetName(V)) + 18) * sizeof(char));
      if (VRes != NULL && VResName != NULL)
        {
          sprintf(VResName, "%12.5e * (%s)", S, V_GetName(V));
          V_Constr(VRes, VResName, V->Dim, Tempor, False);
          if (LASResult() == LASOK)
            {
              VRes->Multipl = S * V->Multipl;
              if (V->Instance == Tempor && V->OwnData)
                {
                  V->OwnData = False;
                  VRes->OwnData = True;
                }
              VRes->Cmp = V->Cmp;
            }
        }
      else
        {
```

```
          LASError(LASMemAllocErr, "Mul_SV", V_GetName(V), NULL, NULL);
          if (VRes != NULL)
            free(VRes);
        }

      if (VResName != NULL)
        free(VResName);
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V);

  return (VRes);
}

Matrix *Mul_SM(double S, Matrix *M)
/* MRes = S * M */
{
  Matrix *MRes;

  char *MResName;

  M_Lock(M);

  if (LASResult() == LASOK)
    {
      MRes = (Matrix *)malloc(sizeof(Matrix));
      MResName = (char *)malloc((strlen(M_GetName(M)) + 20) * sizeof(char));
      if (MRes != NULL && MResName != NULL)
        {
          sprintf(MResName, "%12.5e * (%s)", S, M_GetName(M));
          M_Constr(MRes, MResName,  M->RowDim, M->ClmDim, M->ElOrder, Tempor, Fa
          if (LASResult() == LASOK)
            {
              MRes->Multipl = S * M->Multipl;
              MRes->Len = M->Len;
              MRes->El = M->El;
              MRes->ElSorted = M->ElSorted;
```

```
            }
        }
      else
        {
          LASError(LASMemAllocErr, "Mul_SM", M_GetName(M), NULL, NULL);
          if (MRes != NULL)
            free(MRes);
        }

      if (MResName != NULL)
        free(MResName);
    }
  else
    {
      MRes = NULL;
    }

  M_Unlock(M);

  return (MRes);
}

QMatrix *Mul_SQ(double S, QMatrix *Q)
/* QRes = S * Q */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q);

  if (LASResult() == LASOK)
    {
      QRes = (QMatrix *)malloc(sizeof(QMatrix));
      QResName = (char *)malloc((strlen(Q_GetName(Q)) + 20) * sizeof(char));
      if (QRes != NULL && QResName != NULL)
        {
          sprintf(QResName, "%12.5e * (%s)", S, Q_GetName(Q));
          Q_Constr(QRes, QResName, Q->Dim, Q->Symmetry, Q->ElOrder, Tempor, Fals
          if (LASResult() == LASOK)
            {
```

```
            if (Q->Instance == Tempor && Q->OwnData)
              {
                Q->OwnData = False;
                QRes->OwnData = True;
              }
            QRes->MultiplD = S * Q->MultiplD;
            QRes->MultiplU = S * Q->MultiplU;
            QRes->MultiplL = S * Q->MultiplL;
            QRes->Len = Q->Len;
            QRes->El = Q->El;
            QRes->ElSorted = Q->ElSorted;
            QRes->DiagElAlloc = Q->DiagElAlloc;
            QRes->DiagEl = Q->DiagEl;
            QRes->ZeroInDiag = Q->ZeroInDiag;
            QRes->InvDiagEl = Q->InvDiagEl;
            QRes->UnitRightKer = Q->UnitRightKer;
            QRes->RightKerCmp = Q->RightKerCmp;
            QRes->UnitLeftKer = Q->UnitLeftKer;
            QRes->LeftKerCmp = Q->LeftKerCmp;
            QRes->ILUExists = Q->ILUExists;
            QRes->ILU = Q->ILU;
          }
      }
    else
      {
        LASError(LASMemAllocErr, "Mul_SQ", Q_GetName(Q), NULL, NULL);
        if (QRes != NULL)
          free(QRes);
      }

    if (QResName != NULL)
      free(QResName);
  }
else
  {
    QRes = NULL;
  }

Q_Unlock(Q);

return (QRes);
```

```
}

double Mul_VV(Vector *V1, Vector *V2)
/* S = V1 * V2 */
{
  double SRes;

  size_t Dim, Ind;
  double *V1Cmp, *V2Cmp;

  V_Lock(V1);
  V_Lock(V2);

  if (LASResult() == LASOK)
    {
      if (V1->Dim == V2->Dim)
        {
          Dim = V1->Dim;
          V1Cmp = V1->Cmp;
          V2Cmp = V2->Cmp;
          SRes = 0.0;
          for_AllCmp
          SRes += V1Cmp[Ind] * V2Cmp[Ind];
          SRes *= V1->Multipl * V2->Multipl;
        }
      else
        {
          LASError(LASDimErr, "Mul_VV", V_GetName(V1),  V_GetName(V2), NULL);
          SRes = 1.0;
        }
    }
  else
    {
      SRes = 1.0;
    }

  V_Unlock(V1);
  V_Unlock(V2);

  return (SRes);
}
```

```
Vector *Mul_MV(Matrix *M, Vector *V)
/* VRes = M * V */
{
  Vector *VRes;

  char *VResName;
  double MultiplMV;
  double Sum, Cmp;
  size_t RowDim, ClmDim, Row, Clm, Len, ElCount;
  size_t *MLen;
  Boolean MultiplMVIsOne;
  ElType **MEl, *PtrEl;
  double *VCmp, *VResCmp;

  M_Lock(M);
  V_Lock(V);

  if (LASResult() == LASOK)
    {
      if (M->ClmDim == V->Dim)
        {
          RowDim = M->RowDim;
          ClmDim = M->ClmDim;
          VRes = (Vector *)malloc(sizeof(Vector));
          VResName = (char *)malloc((strlen(M_GetName(M)) + strlen(V_GetName(V))
                                     * sizeof(char));
          if (VRes != NULL && VResName != NULL)
            {
              sprintf(VResName, "(%s) * (%s)", M_GetName(M), V_GetName(V));
              V_Constr(VRes, VResName, RowDim, Tempor, True);
              if (LASResult() == LASOK)
                {
                  /* assignment of auxiliary lokal variables */
                  MLen = M->Len;
                  MEl = M->El;
                  VCmp = V->Cmp;
                  VResCmp = VRes->Cmp;

                  /* initialisation of the vector VRes */
                  if (M->ElOrder == Clmws)
```

```
  V_SetAllCmp(VRes, 0.0);

/* analysis of multipliers of the matrix M and the vector V */
MultiplMV = M->Multipl * V->Multipl;
if (IsOne(MultiplMV))
  {
    MultiplMVIsOne = True;
  }
else
  {
    MultiplMVIsOne = False;
  }

/* multiplication of matrix elements by vector components */
if (M->ElOrder == Rowws)
  {
    for (Row = 1; Row <= RowDim; Row++)
      {
        Len = MLen[Row];
        PtrEl = MEl[Row];
        Sum = 0.0;
        for (ElCount = Len; ElCount > 0; ElCount--)
          {
            Sum += (*PtrEl).Val * VCmp[(*PtrEl).Pos];
            PtrEl++;
          }
        if (MultiplMVIsOne)
          VResCmp[Row] = Sum;
        else
          VResCmp[Row] = MultiplMV * Sum;
      }
  }
if (M->ElOrder == Clmws)
  {
    for (Clm = 1; Clm <= ClmDim; Clm++)
      {
        Len = MLen[Clm];
        PtrEl = MEl[Clm];
        if (MultiplMVIsOne)
          Cmp = VCmp[Clm];
        else
```

```
                        Cmp = MultiplMV * VCmp[Clm];
                      for (ElCount = Len; ElCount > 0; ElCount--)
                        {
                          VResCmp[(*PtrEl).Pos] += (*PtrEl).Val * Cmp;
                          PtrEl++;
                        }
                    }
                }
            }
          else
            {
              LASError(LASMemAllocErr, "Mul_MV", M_GetName(M), V_GetName(V), NUL
              if (VRes != NULL)
                free(VRes);
            }

          if (VResName != NULL)
            free(VResName);
        }
      else
        {
          LASError(LASDimErr, "Mul_MV", M_GetName(M), V_GetName(V), NULL);
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  M_Unlock(M);
  V_Unlock(V);

  return (VRes);
}

Vector *Mul_QV(QMatrix *Q, Vector *V)
/* VRes = Q * V */
{
  Vector *VRes;
```

```
char *VResName;
double MultiplDV, MultiplUV, MultiplLV;
double Sum, PartSum, Cmp, PartCmp;
size_t Dim, Row, Clm, RoC, Len, ElCount;
size_t *QLen;
Boolean MultiplDVIsZero, MultiplUVIsZero, MultiplLVIsZero;
Boolean MultiplDVIsOne, MultiplUVIsOne, MultiplLVIsOne;
Boolean MultiplDULVEquals;
ElType **QEl, **QDiagEl, *PtrEl;
double *VCmp, *VResCmp;

Q_Lock(Q);
V_Lock(V);

if (LASResult() == LASOK)
  {
    if (Q->Dim == V->Dim)
      {
        Dim = V->Dim;
        VRes = (Vector *)malloc(sizeof(Vector));
        VResName = (char *)malloc((strlen(Q_GetName(Q)) + strlen(V_GetName(V))
                                   * sizeof(char));
        if (VRes != NULL && VResName != NULL)
          {
            sprintf(VResName, "(%s) * (%s)", Q_GetName(Q), V_GetName(V));
            V_Constr(VRes, VResName, Dim, Tempor, True);

            /* sort of elements and allocation of diagonal elements
             of the matrix Q */
            Q_SortEl(Q);
            Q_AllocInvDiagEl(Q);

            if (LASResult() == LASOK && Q->ElSorted)
              {
                /* assignment of auxiliary lokal variables */
                QLen = Q->Len;
                QEl = Q->El;
                QDiagEl = Q->DiagEl;
                VCmp = V->Cmp;
                VResCmp = VRes->Cmp;
```

```
/* initialisation of the vector VRes */
if (Q->Symmetry || Q->ElOrder == Clmws)
  V_SetAllCmp(VRes, 0.0);

/* analysis of multipliers of the lower, diagonal and upper pa
 of the matrix Q and of the vector V */
MultiplDV = Q->MultiplD * V->Multipl;
MultiplUV = Q->MultiplU * V->Multipl;
MultiplLV = Q->MultiplL * V->Multipl;
MultiplDVIsZero = IsZero(MultiplDV);
MultiplUVIsZero = IsZero(MultiplUV);
MultiplLVIsZero = IsZero(MultiplLV);
MultiplDVIsOne = IsOne(MultiplDV);
MultiplUVIsOne = IsOne(MultiplUV);
MultiplLVIsOne = IsOne(MultiplLV);
if (!IsZero(MultiplDV) && IsOne(MultiplUV / MultiplDV)
    && IsOne(MultiplLV / MultiplDV))
  {
    MultiplDULVEquals = True;
  }
else
  {
    MultiplDULVEquals = False;
  }

/* multiplication of the lower, diagonal and upper part
 of the matrix Q by the vector V */
if (Q->Symmetry)
  {
    if (MultiplDULVEquals)
      {
        for (RoC = 1; RoC <= Dim; RoC++)
          {
            Len = QLen[RoC];
            PtrEl = QEl[RoC];
            Sum = 0.0;
            if (!MultiplDVIsOne)
              Cmp = MultiplDV * VCmp[RoC];
            else
              Cmp = VCmp[RoC];
```

```
            for (ElCount = Len; ElCount > 0; ElCount--)
              {
                if ((*PtrEl).Pos != RoC)
                  {
                    VResCmp[(*PtrEl).Pos] += (*PtrEl).Val * Cm
                    Sum += (*PtrEl).Val * VCmp[(*PtrEl).Pos];
                  }
                PtrEl++;
              }
            Sum += (*QDiagEl[RoC]).Val * VCmp[RoC];
            if (MultiplDVIsOne)
              VResCmp[RoC] += Sum;
            else
              VResCmp[RoC] += MultiplDV * Sum;
          }
      }
    else
      {
        for (RoC = 1; RoC <= Dim; RoC++)
          {
            Len = QLen[RoC];
            Sum = 0.0;
            if ((!MultiplUVIsZero && Q->ElOrder == Rowws)
                || (!MultiplLVIsZero && Q->ElOrder == Clmws))
              {
                PtrEl = QEl[RoC];
                for (ElCount = Len; ElCount > 0; ElCount--)
                  {
                    if ((*PtrEl).Pos != RoC)
                      {
                        Sum += (*PtrEl).Val * VCmp[(*PtrEl).Po
                      }
                    PtrEl++;
                  }
                if (!MultiplUVIsZero && !MultiplUVIsOne)
                  Sum = MultiplUV * Sum;
                if (!MultiplLVIsZero && !MultiplLVIsOne)
                  Sum = MultiplLV * Sum;
              }
            if (!MultiplDVIsZero)
              {
```

```
                    if (MultiplDVIsOne)
                      Sum += (*QDiagEl[RoC]).Val * VCmp[RoC];
                    else
                      Sum += MultiplDV * (*QDiagEl[RoC]).Val * VCm
                  }
                VResCmp[RoC] += Sum;
                if ((!MultiplUVIsZero && Q->ElOrder == Clmws)
                    || (!MultiplLVIsZero && Q->ElOrder == Rowws))
                  {
                    Cmp = VCmp[RoC];
                    if (!MultiplUVIsZero && !MultiplUVIsOne)
                      Cmp *= MultiplUV;
                    if (!MultiplLVIsZero && !MultiplLVIsOne)
                      Cmp *= MultiplLV;
                    PtrEl = QEl[RoC];
                    for (ElCount = Len; ElCount > 0; ElCount--)
                      {
                        if ((*PtrEl).Pos != RoC)
                          VResCmp[(*PtrEl).Pos] += (*PtrEl).Val *
                        PtrEl++;
                      }
                  }
              }
          }
      }
    if (!Q->Symmetry && Q->ElOrder == Rowws)
      {
        if (MultiplDULVEquals)
          {
            for (Row = 1; Row <= Dim; Row++)
              {
                Len = QLen[Row];
                PtrEl = QEl[Row];
                Sum = 0.0;
                for (ElCount = Len; ElCount > 0; ElCount--)
                  {
                    Sum += (*PtrEl).Val * VCmp[(*PtrEl).Pos];
                    PtrEl++;
                  }
                if (MultiplDVIsOne)
                  VResCmp[Row] = Sum;
```

```
        else
          VResCmp[Row] = MultiplDV * Sum;
      }
  }
else
  {
    for (Row = 1; Row <= Dim; Row++)
      {
        Len = QLen[Row];
        Sum = 0.0;
        if (!MultiplLVIsZero)
          {
            PtrEl = QEl[Row];
            PartSum = 0.0;
            for (ElCount = Len; ElCount > 0 && (*PtrEl).Po
                 ElCount--)
              {
                PartSum += (*PtrEl).Val * VCmp[(*PtrEl).Po
                PtrEl++;
              }
            if (MultiplLVIsOne)
              Sum += PartSum;
            else
              Sum += MultiplLV * PartSum;
          }
        if (!MultiplDVIsZero)
          {
            if (MultiplDVIsOne)
              Sum += (*QDiagEl[Row]).Val * VCmp[Row];
            else
              Sum += MultiplDV * (*QDiagEl[Row]).Val * VCm
          }
        if (!MultiplUVIsZero)
          {
            PtrEl = QEl[Row] + Len - 1;
            PartSum = 0.0;
            for (ElCount = Len; ElCount > 0 && (*PtrEl).Po
                 ElCount--)
              {
                PartSum += (*PtrEl).Val * VCmp[(*PtrEl).Po
                PtrEl--;
```

```
                }
              if (MultiplUVIsOne)
                Sum += PartSum;
              else
                Sum += MultiplUV * PartSum;
            }
          VResCmp[Row] = Sum;
        }
    }
  }
if (!Q->Symmetry && Q->ElOrder == Clmws)
  {
    if (MultiplDULVEquals)
      {
        for (Clm = 1; Clm <= Dim; Clm++)
          {
            Len = QLen[Clm];
            PtrEl = QEl[Clm];
            if (MultiplDVIsOne)
              Cmp = VCmp[Clm];
            else
              Cmp = MultiplDV * VCmp[Clm];
            for (ElCount = Len; ElCount > 0; ElCount--)
              {
                VResCmp[(*PtrEl).Pos] += (*PtrEl).Val * Cmp;
                PtrEl++;
              }
          }
      }
    else
      {
        for (Clm = 1; Clm <= Dim; Clm++)
          {
            Len = QLen[Clm];
            Cmp = VCmp[Clm];
            if (!MultiplUVIsZero)
              {
                PtrEl = QEl[Clm];
                if (MultiplUVIsOne)
                  PartCmp = Cmp;
                else
```

```
                    PartCmp = MultiplUV * Cmp;
                  for (ElCount = Len; ElCount > 0 && (*PtrEl).Po
                       ElCount--)
                    {
                      VResCmp[(*PtrEl).Pos] += (*PtrEl).Val * Pa
                      PtrEl++;
                    }
                }
              if (!MultiplDVIsZero)
                {
                  if (MultiplDVIsOne)
                    VResCmp[Clm] += (*QDiagEl[Clm]).Val * Cmp;
                  else
                    VResCmp[Clm] += MultiplDV * (*QDiagEl[Clm]).
                }
              if (!MultiplLVIsZero)
                {
                  PtrEl = QEl[Clm] + Len - 1;
                  if (MultiplLVIsOne)
                    PartCmp = Cmp;
                  else
                    PartCmp = MultiplLV * Cmp;
                  for (ElCount = Len; ElCount > 0 && (*PtrEl).Po
                       ElCount--)
                    {
                      VResCmp[(*PtrEl).Pos] += (*PtrEl).Val * Cm
                      PtrEl--;
                    }
                }
            }
        }
    }
  else
    {
      if (LASResult() == LASOK && !(*Q->ElSorted))
        LASError(LASElNotSortedErr, "Mul_QV", Q_GetName(Q), V_GetNam
    }
}
else
  {
```

```
              LASError(LASMemAllocErr, "Mul_QV", Q_GetName(Q), V_GetName(V), NUL
              if (VRes != NULL)
                free(VRes);
            }

          if (VResName != NULL)
            free(VResName);
        }
      else
        {
          LASError(LASDimErr, "Mul_QV", Q_GetName(Q), V_GetName(V), NULL);
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  Q_Unlock(Q);
  V_Unlock(V);

  return (VRes);
}

Vector *MulInv_QV(QMatrix *Q, Vector *V)
/* VRes = Q^(-1) * V, this operation is limited to diagonal or triangular
 matrices */
{
  Vector *VRes;

  char *VResName;
  double MultiplD, MultiplU, MultiplL, MultiplV, MultiplDV;
  double Sum, Cmp;
  size_t Dim, Row, Clm, Ind, Len, ElCount;
  size_t *QLen;
  Boolean MultiplDIsZero, MultiplUIsZero, MultiplLIsZero;
  Boolean MultiplDIsOne, MultiplUIsOne, MultiplLIsOne, MultiplVIsOne,
          MultiplDVIsOne;
  ElType **QEl, *PtrEl;
  double *QInvDiagEl;
```

```
double *VCmp, *VResCmp;

Q_Lock(Q);
V_Lock(V);

if (LASResult() == LASOK)
  {
    if (Q->Dim == V->Dim)
      {
        Dim = V->Dim;
        VRes = (Vector *)malloc(sizeof(Vector));
        VResName = (char *)malloc((strlen(Q_GetName(Q)) + strlen(V_GetName(V))
                                   * sizeof(char));
        if (VRes != NULL && VResName != NULL)
          {
            sprintf(VResName, "(%s)^(-1) * (%s)", Q_GetName(Q), V_GetName(V));
            V_Constr(VRes, VResName, Dim, Tempor, True);

            /* sort of elements and allocation of the inverse of diagonal elem
             of the matrix Q */
            Q_SortEl(Q);
            Q_AllocInvDiagEl(Q);

            if (LASResult() == LASOK && *Q->ElSorted && !(*Q->ZeroInDiag)
                && !IsZero(Q->MultiplD))
              {
                /* assignment of auxiliary lokal variables */
                QLen = Q->Len;
                QEl = Q->El;
                QInvDiagEl = Q->InvDiagEl;
                VCmp = V->Cmp;
                VResCmp = VRes->Cmp;

                /* initialisation of the vector VRes */
                if (Q->Symmetry || Q->ElOrder == Clmws)
                  V_SetAllCmp(VRes, 0.0);

                /* analysis of multipliers of the lower, diagonal and upper pa
                 of the matrix Q and of the vector V */
                MultiplD = 1.0 / Q->MultiplD;  /* attention here !!! */
                MultiplU = Q->MultiplU;
```

```
MultiplL = Q->MultiplL;
MultiplV = V->Multipl;
MultiplDV = V->Multipl / Q->MultiplD;  /* attention here !!! *
MultiplDIsZero = IsZero(MultiplD);
MultiplUIsZero = IsZero(MultiplU);
MultiplLIsZero = IsZero(MultiplL);
MultiplDIsOne = IsOne(MultiplD);
MultiplUIsOne = IsOne(MultiplU);
MultiplLIsOne = IsOne(MultiplL);
MultiplVIsOne = IsOne(MultiplV);
MultiplDVIsOne = IsOne(MultiplDV);

/* multiplication of the vector V by the inverse matrix
 of the diagonal of M */
if (!MultiplDIsZero && MultiplUIsZero && MultiplLIsZero)
  {
    if (MultiplDVIsOne)
      {
        for_AllCmp
        VResCmp[Ind] = VCmp[Ind] * QInvDiagEl[Ind];
      }
    else
      {
        for_AllCmp
        VResCmp[Ind] = MultiplDV * VCmp[Ind] * QInvDiagEl[Ind]
      }
  }

/* multiplication of the vector V by the inverse matrix
 of the upper triangular part of M */
if (!MultiplUIsZero && MultiplLIsZero)
  {
    if ((!Q->Symmetry && Q->ElOrder == Rowws)
        || (Q->Symmetry && Q->ElOrder == Rowws))
      {
        for (Row = Dim; Row >= 1; Row--)
          {
            Len = QLen[Row];
            PtrEl = QEl[Row] + Len - 1;
            Sum = 0.0;
            for (ElCount = Len; ElCount > 0 && (*PtrEl).Pos >
```

```
            ElCount--)
          {
            Sum -= (*PtrEl).Val * VResCmp[(*PtrEl).Pos];
            PtrEl--;
          }
        if (!MultiplUIsOne)
          Sum *= MultiplU;
        if (MultiplVIsOne)
          Sum += VCmp[Row];
        else
          Sum += MultiplV * VCmp[Row];
        if (MultiplDIsOne)
          {
            VResCmp[Row] = Sum * QInvDiagEl[Row];
          }
        else
          {
            VResCmp[Row] = Sum * MultiplD * QInvDiagEl[Row
          }
      }
  }
if ((!Q->Symmetry && Q->ElOrder == Clmws)
    || (Q->Symmetry && Q->ElOrder == Clmws))
  {
    for (Clm = Dim; Clm >= 1; Clm--)
      {
        Sum = VResCmp[Clm];
        if (!MultiplUIsOne)
          Sum *= MultiplU;
        if (MultiplVIsOne)
          Sum += VCmp[Clm];
        else
          Sum += MultiplV * VCmp[Clm];
        if (MultiplDIsOne)
          {
            Cmp = Sum * QInvDiagEl[Clm];
          }
        else
          {
            Cmp = Sum * MultiplD * QInvDiagEl[Clm];
          }
```

```
                VResCmp[Clm] = Cmp;

                Len = QLen[Clm];
                PtrEl = QEl[Clm];
                for (ElCount = Len; ElCount > 0 && (*PtrEl).Pos <
                     ElCount--)
                  {
                    VResCmp[(*PtrEl).Pos] -= (*PtrEl).Val * Cmp;
                    PtrEl++;
                  }
              }
          }
      }

/* multiplication of the vector V by the inverse matrix
 of the lower triangular part of M */
if (MultiplUIsZero && !MultiplLIsZero)
  {
    if ((!Q->Symmetry && Q->ElOrder == Rowws)
        || (Q->Symmetry && Q->ElOrder == Clmws))
      {
        for (Row = 1; Row <= Dim; Row++)
          {
            Len = QLen[Row];
            PtrEl = QEl[Row];
            Sum = 0.0;
            for (ElCount = Len; ElCount > 0 && (*PtrEl).Pos <
                 ElCount--)
              {
                Sum -= (*PtrEl).Val * VResCmp[(*PtrEl).Pos];
                PtrEl++;
              }
            if (!MultiplLIsOne)
              Sum *= MultiplL;
            if (MultiplVIsOne)
              Sum += VCmp[Row];
            else
              Sum += MultiplV * VCmp[Row];
            if (MultiplDIsOne)
              {
                VResCmp[Row] = Sum * QInvDiagEl[Row];
```

```
                }
              else
                {
                  VResCmp[Row] = Sum * MultiplD * QInvDiagEl[Row
                }
            }
        }
      if ((!Q->Symmetry && Q->ElOrder == Clmws)
          || (Q->Symmetry && Q->ElOrder == Rowws))
        {
          for (Clm = 1; Clm <= Dim; Clm++)
            {
              Sum = VResCmp[Clm];
              if (!MultiplLIsOne)
                Sum *= MultiplL;
              if (MultiplVIsOne)
                Sum += VCmp[Clm];
              else
                Sum += MultiplV * VCmp[Clm];
              if (MultiplDIsOne)
                {
                  Cmp = Sum * QInvDiagEl[Clm];
                }
              else
                {
                  Cmp = Sum * MultiplD * QInvDiagEl[Clm];
                }
              VResCmp[Clm] = Cmp;

              Len = QLen[Clm];
              PtrEl = QEl[Clm] + Len - 1;
              for (ElCount = Len; ElCount > 0 && (*PtrEl).Pos >
                   ElCount--)
                {
                  VResCmp[(*PtrEl).Pos] -= (*PtrEl).Val * Cmp;
                  PtrEl--;
                }
            }
        }
    }
  if (!MultiplUIsZero && !MultiplLIsZero)
```

```
                    {
                      LASError(LASMulInvErr, "MulInv_QV", Q_GetName(Q), V_GetNam
                    }
                }
              else
                {
                  if (LASResult() == LASOK && !(*Q->ElSorted))
                    LASError(LASElNotSortedErr, "MulInv_QV", Q_GetName(Q), V_Get
                  if (LASResult() == LASOK && (*Q->ZeroInDiag || IsZero(Q->Multi
                    LASError(LASZeroInDiagErr, "MulInv_QV", Q_GetName(Q), V_GetN
                }
            }
          else
            {
              LASError(LASMemAllocErr, "MulInv_QV", Q_GetName(Q), V_GetName(V),
              if (VRes != NULL)
                free(VRes);
            }

          if (VResName != NULL)
            free(VResName);
        }
      else
        {
          LASError(LASDimErr, "MulInv_QV", Q_GetName(Q), V_GetName(V), NULL);
          VRes = NULL;
        }
    }
  else
    {
      VRes = NULL;
    }

  Q_Unlock(Q);
  V_Unlock(V);

  return (VRes);
}

Matrix *Transp_M(Matrix *M)
/* MRes = M^T, returns transposed matrix M */
```

```
{
  Matrix *MRes;

  char *MResName;

  M_Lock(M);

  if (LASResult() == LASOK)
    {
      MRes = (Matrix *)malloc(sizeof(Matrix));
      MResName = (char *)malloc((strlen(M_GetName(M)) + 5) * sizeof(char));
      if (MRes != NULL && MResName != NULL)
        {
          sprintf(MResName, "(%s)^T", M_GetName(M));
          M_Constr(MRes, MResName,  M->ClmDim, M->RowDim, M->ElOrder, Tempor, Fa
          if (LASResult() == LASOK)
            {
              if (M->ElOrder == Rowws)
                MRes->ElOrder = Clmws;
              if (M->ElOrder == Clmws)
                MRes->ElOrder = Rowws;
              MRes->Multipl = M->Multipl;
              if (M->Instance == Tempor && M->OwnData)
                {
                  M->OwnData = False;
                  MRes->OwnData = True;
                }
              MRes->Len = M->Len;
              MRes->El = M->El;
              MRes->ElSorted = M->ElSorted;
            }
        }
      else
        {
          LASError(LASMemAllocErr, "Transp_M", M_GetName(M), NULL, NULL);
          if (MRes != NULL)
            free(MRes);
        }

      if (MResName != NULL)
        free(MResName);
```

```
    }
  else
    {
      MRes = NULL;
    }

  M_Unlock(M);

  return (MRes);
}

QMatrix *Transp_Q(QMatrix *Q)
/* QRes = Q^T, returns transposed matrix Q */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q);

  if (LASResult() == LASOK)
    {
      QRes = (QMatrix *)malloc(sizeof(QMatrix));
      QResName = (char *)malloc((strlen(Q_GetName(Q)) + 5) * sizeof(char));
      if (QRes != NULL && QResName != NULL)
        {
          sprintf(QResName, "(%s)^T", Q_GetName(Q));
          Q_Constr(QRes, QResName, Q->Dim, Q->Symmetry, Q->ElOrder, Tempor, Fals
          if (LASResult() == LASOK)
            {
              if (Q->Instance == Tempor && Q->OwnData)
                {
                  Q->OwnData = False;
                  QRes->OwnData = True;
                }
              if (!Q->Symmetry)
                {
                  if (Q->ElOrder == Rowws)
                    QRes->ElOrder = Clmws;
                  if (Q->ElOrder == Clmws)
                    QRes->ElOrder = Rowws;
```

```
            }
          QRes->MultiplD = Q->MultiplD;
          QRes->MultiplU = Q->MultiplL;
          QRes->MultiplL = Q->MultiplU;
          QRes->Len = Q->Len;
          QRes->El = Q->El;
          QRes->ElSorted = Q->ElSorted;
          QRes->DiagElAlloc = Q->DiagElAlloc;
          QRes->DiagEl = Q->DiagEl;
          QRes->ZeroInDiag = Q->ZeroInDiag;
          QRes->InvDiagEl = Q->InvDiagEl;
          if (!Q->Symmetry)
            {
              QRes->UnitRightKer = Q->UnitLeftKer;
              QRes->RightKerCmp = Q->LeftKerCmp;
              QRes->UnitLeftKer = Q->UnitRightKer;
              QRes->LeftKerCmp = Q->RightKerCmp;
            }
          else
            {
              QRes->UnitRightKer = Q->UnitRightKer;
              QRes->RightKerCmp = Q->RightKerCmp;
              QRes->UnitLeftKer = Q->UnitLeftKer;
              QRes->LeftKerCmp = Q->LeftKerCmp;
            }
          QRes->ILUExists = Q->ILUExists;
          QRes->ILU = Q->ILU;
        }
    }
  else
    {
      LASError(LASMemAllocErr, "Transp_Q", Q_GetName(Q), NULL, NULL);
      if (QRes != NULL)
        free(QRes);
    }

  if (QResName != NULL)
    free(QResName);
}
else
{
```

```
      QRes = NULL;
    }

  Q_Unlock(Q);

  return (QRes);
}

QMatrix *Diag_Q(QMatrix *Q)
/* QRes = Diag(Q), returns the diagonal of the matrix Q */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q);

  if (LASResult() == LASOK)
    {
      QRes = (QMatrix *)malloc(sizeof(QMatrix));
      QResName = (char *)malloc((strlen(Q_GetName(Q)) + 7) * sizeof(char));
      if (QRes != NULL && QResName != NULL)
        {
          sprintf(QResName, "Diag(%s)", Q_GetName(Q));
          Q_Constr(QRes, QResName, Q->Dim, Q->Symmetry, Q->ElOrder, Tempor, Fals
          if (LASResult() == LASOK)
            {
              if (Q->Instance == Tempor && Q->OwnData)
                {
                  Q->OwnData = False;
                  QRes->OwnData = True;
                }
              QRes->MultiplD = Q->MultiplD;
              QRes->MultiplU = 0.0;
              QRes->MultiplL = 0.0;
              QRes->Len = Q->Len;
              QRes->El = Q->El;
              QRes->ElSorted = Q->ElSorted;
              QRes->DiagElAlloc = Q->DiagElAlloc;
              QRes->DiagEl = Q->DiagEl;
              QRes->ZeroInDiag = Q->ZeroInDiag;
```

```
              QRes->InvDiagEl = Q->InvDiagEl;
              QRes->UnitRightKer = Q->UnitRightKer;
              QRes->RightKerCmp = Q->RightKerCmp;
              QRes->UnitLeftKer = Q->UnitLeftKer;
              QRes->LeftKerCmp = Q->LeftKerCmp;
              QRes->ILUExists = Q->ILUExists;
              QRes->ILU = Q->ILU;
            }
        }
      else
        {
          LASError(LASMemAllocErr, "Diag_Q", Q_GetName(Q), NULL, NULL);
          if (QRes != NULL)
            free(QRes);
        }

      if (QResName != NULL)
        free(QResName);
    }
  else
    {
      QRes = NULL;
    }

  Q_Unlock(Q);

  return (QRes);
}

QMatrix *Upper_Q(QMatrix *Q)
/* QRes = Upper(Q), returns the upper triagonal part of the matrix Q */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q);

  if (LASResult() == LASOK)
    {
      QRes = (QMatrix *)malloc(sizeof(QMatrix));
```

```
    QResName = (char *)malloc((strlen(Q_GetName(Q)) + 8) * sizeof(char));
    if (QRes != NULL && QResName != NULL)
      {
        sprintf(QResName, "Upper(%s)", Q_GetName(Q));
        Q_Constr(QRes, QResName, Q->Dim, Q->Symmetry, Q->ElOrder, Tempor, Fals
        if (LASResult() == LASOK)
          {
            if (Q->Instance == Tempor && Q->OwnData)
              {
                Q->OwnData = False;
                QRes->OwnData = True;
              }
            QRes->MultiplD = 0.0;
            QRes->MultiplU = Q->MultiplU;
            QRes->MultiplL = 0.0;
            QRes->Len = Q->Len;
            QRes->El = Q->El;
            QRes->ElSorted = Q->ElSorted;
            QRes->DiagElAlloc = Q->DiagElAlloc;
            QRes->DiagEl = Q->DiagEl;
            QRes->ZeroInDiag = Q->ZeroInDiag;
            QRes->InvDiagEl = Q->InvDiagEl;
            QRes->UnitRightKer = Q->UnitRightKer;
            QRes->RightKerCmp = Q->RightKerCmp;
            QRes->UnitLeftKer = Q->UnitLeftKer;
            QRes->LeftKerCmp = Q->LeftKerCmp;
            QRes->ILUExists = Q->ILUExists;
            QRes->ILU = Q->ILU;
          }
      }
    else
      {
        LASError(LASMemAllocErr, "Upper_Q", Q_GetName(Q), NULL, NULL);
        if (QRes != NULL)
          free(QRes);
      }

    if (QResName != NULL)
      free(QResName);
  }
else
```

```
    {
      QRes = NULL;
    }

  Q_Unlock(Q);

  return (QRes);
}

QMatrix *Lower_Q(QMatrix *Q)
/* QRes = Lower(Q), returns the lower triagonal part of the matrix Q */
{
  QMatrix *QRes;

  char *QResName;

  Q_Lock(Q);

  if (LASResult() == LASOK)
    {
      QRes = (QMatrix *)malloc(sizeof(QMatrix));
      QResName = (char *)malloc((strlen(Q_GetName(Q)) + 8) * sizeof(char));
      if (QRes != NULL && QResName != NULL)
        {
          sprintf(QResName, "Lower(%s)", Q_GetName(Q));
          Q_Constr(QRes, QResName, Q->Dim, Q->Symmetry, Q->ElOrder, Tempor, Fals
          if (LASResult() == LASOK)
            {
              if (Q->Instance == Tempor && Q->OwnData)
                {
                  Q->OwnData = False;
                  QRes->OwnData = True;
                }
              QRes->MultiplD = 0.0;
              QRes->MultiplU = 0.0;
              QRes->MultiplL = Q->MultiplL;
              QRes->Len = Q->Len;
              QRes->El = Q->El;
              QRes->ElSorted = Q->ElSorted;
              QRes->DiagElAlloc = Q->DiagElAlloc;
              QRes->DiagEl = Q->DiagEl;
```

```
              QRes->ZeroInDiag = Q->ZeroInDiag;
              QRes->InvDiagEl = Q->InvDiagEl;
              QRes->UnitRightKer = Q->UnitRightKer;
              QRes->RightKerCmp = Q->RightKerCmp;
              QRes->UnitLeftKer = Q->UnitLeftKer;
              QRes->LeftKerCmp = Q->LeftKerCmp;
              QRes->ILUExists = Q->ILUExists;
              QRes->ILU = Q->ILU;
            }
        }
      else
        {
          LASError(LASMemAllocErr, "Lower_Q", Q_GetName(Q), NULL, NULL);
          if (QRes != NULL)
            free(QRes);
        }

      if (QResName != NULL)
        free(QResName);
    }
  else
    {
      QRes = NULL;
    }

  Q_Unlock(Q);

  return (QRes);
}

double l1Norm_V(Vector *V)
/* SRes = l1-Norm of the vector V */
{
  double SRes;

  double Sum;
  size_t Dim, Ind;
  double *VCmp;

  V_Lock(V);
```

```
  if (LASResult() == LASOK)
    {
      Dim = V->Dim;
      VCmp = V->Cmp;
      Sum = 0.0;
      for_AllCmp
      Sum += fabs(VCmp[Ind]);
      Sum *= V->Multipl;
      SRes = Sum;
    }
  else
    {
      SRes = 1.0;
    }

  V_Unlock(V);

  return (SRes);
}

double l2Norm_V(Vector *V)
/* SRes = l2-Norm of the vector V */
{
  double SRes;

  double Sum, Cmp;
  size_t Dim, Ind;
  double *VCmp;

  V_Lock(V);

  if (LASResult() == LASOK)
    {
      Dim = V->Dim;
      VCmp = V->Cmp;
      Sum = 0.0;
      for_AllCmp
      {
        Cmp = VCmp[Ind];
        Sum += Cmp * Cmp;
      }
```

```
      Sum *= V->Multipl * V->Multipl;
      SRes = sqrt(Sum);
    }
  else
    {
      SRes = 1.0;
    }

  V_Unlock(V);

  return (SRes);
}

double MaxNorm_V(Vector *V)
/* SRes = max-Norm of the vector V */
{
  double SRes;

  double MaxCmp, Cmp;
  size_t Dim, Ind;
  double *VCmp;

  V_Lock(V);

  if (LASResult() == LASOK)
    {
      Dim = V->Dim;
      VCmp = V->Cmp;
      MaxCmp = 0.0;
      for_AllCmp
      {
        Cmp = fabs(VCmp[Ind]);
        if (Cmp > MaxCmp)
          MaxCmp = Cmp;
      }
      MaxCmp *= V->Multipl;
      SRes = MaxCmp;
    }
  else
    {
      SRes = 1.0;
```

```
    }

  V_Unlock(V);

  return (SRes);
}

Vector *OrthoRightKer_VQ(Vector *V, QMatrix *Q)
/* orthogonalize vector V to the "right" null space of matrix Q */
{
  Vector *VRes;

  double Sum, Mean;
  size_t Dim, Ind;
  double *VCmp, *KerCmp;

  V_Lock(V);

  if (LASResult() == LASOK)
    {
      if (Q->UnitRightKer || Q->RightKerCmp != NULL)
        {
          if (V->Instance == Normal && V->Dim == Q->Dim)
            {
              Dim = V->Dim;
              VCmp = V->Cmp;
              KerCmp = Q->RightKerCmp;

              if (Q->UnitRightKer)
                {
                  Sum = 0.0;
                  for_AllCmp
                  Sum += VCmp[Ind];
                  Mean = Sum / (double)Dim;
                  for_AllCmp
                  VCmp[Ind] -= Mean;
                }
              else
                {
                  Sum = 0.0;
                  for_AllCmp
```

```
                Sum += VCmp[Ind] * KerCmp[Ind];
                for_AllCmp
                VCmp[Ind] -= Sum * KerCmp[Ind];
              }

            VRes = V;
          }
        else
          {
            if (V->Instance != Normal)
              LASError(LASLValErr, "OrthoRightKer_VQ", V_GetName(V), Q_GetName
            if (V->Dim != Q->Dim)
              LASError(LASDimErr, "OrthoRightKer_VQ", V_GetName(V), Q_GetName(
            VRes = NULL;
          }
      }
    else
      {
        VRes = V;
      }
  }
else
  {
    VRes = NULL;
  }

V_Unlock(V);

return (VRes);
}

Vector *OrthoLeftKer_VQ(Vector *V, QMatrix *Q)
/* orthogonalize vector V to the "left" null space of matrix Q */
{
  Vector *VRes;

  double Sum, Mean;
  size_t Dim, Ind;
  double *VCmp, *KerCmp;

  V_Lock(V);
```

```
if (LASResult() == LASOK)
  {
    if (Q->UnitRightKer || Q->RightKerCmp != NULL)
      {
        if (V->Instance == Normal && V->Dim == Q->Dim)
          {
            Dim = V->Dim;
            VCmp = V->Cmp;
            if (Q->Symmetry)
              KerCmp = Q->RightKerCmp;
            else
              KerCmp = Q->LeftKerCmp;

            if ((Q->Symmetry && Q->UnitRightKer) || Q->UnitLeftKer)
              {
                Sum = 0.0;
                for_AllCmp
                Sum += VCmp[Ind];
                Mean = Sum / (double)Dim;
                for_AllCmp
                VCmp[Ind] -= Mean;
              }
            else
              {
                Sum = 0.0;
                for_AllCmp
                Sum += VCmp[Ind] * KerCmp[Ind];
                for_AllCmp
                VCmp[Ind] -= Sum * KerCmp[Ind];
              }

            VRes = V;
          }
        else
          {
            if (V->Instance != Normal)
              LASError(LASLValErr, "OrthoLeftKer_VQ", V_GetName(V), Q_GetName(
            if (V->Dim != Q->Dim)
              LASError(LASDimErr, "OrthoLeftKer_VQ", V_GetName(V), Q_GetName(Q
            VRes = NULL;
```

```
            }
        }
      else
        {
          VRes = V;
        }
    }
  else
    {
      VRes = NULL;
    }

  V_Unlock(V);

  return (VRes);
}

#endif //PremiaCurrentVersion
```