Module 9 solutions

CodeJudge exercises

BLAS level 1: `dscal`

```
#include <stdlib.h>
#include "array2d.h"

/* DSCAL (scale array)                                   */
void dscal_(
    const int * n,          /* length of array      */
    const double * a,       /* scalar a             */
    double * x,             /* array x              */
    const int * incx        /* array x, stride      */
);

/* Scale the k'th column of a two-dimensional array */
int scale_column(double alpha, array2d_t *A, size_t k) {

    if ( A==NULL || k>=A->shape[1] ) return 1;

    if (A->order == RowMajor)
        dscal_(&(int){A->shape[0]},&alpha,A->val+k,&(int){A->shape[1]});
    else
        dscal_(&(int){A->shape[0]},&alpha,A->val+k*A->shape[0],&(int){1});

    return 0;
}

/* Scale the k'th row of a two-dimensional array */
int scale_row(double alpha, array2d_t *A, size_t k) {

    if ( A==NULL || k>=A->shape[0] ) return 1;

    if (A->order == RowMajor)
        dscal_(&(int){A->shape[1]},&alpha,A->val+k*A->shape[1],&(int){1});
    else
        dscal_(&(int){A->shape[1]},&alpha,A->val+k,&(int){A->shape[0]});

    return 0;
}

/* Scale the diagonal elements of a square two-dimensional array */
int scale_diag(double alpha, array2d_t *A) {
    if (!A || A->shape[0] != A->shape[1]) return 1;
    dscal_(&(int){A->shape[0]},&alpha,A->val,&(int){A->shape[0]+1});
    return 0;
```

```
}
```

BLAS level 1: `daxpy`

```
#include <stdlib.h>
#include <math.h>
#include "array2d.h"

/* DAXPY (double a x plus y)                          */
void daxpy_(
    const int * n,          /* length of arrays x and y */
    const double * a,       /* scalar a                 */
    const double * x,       /* array x                  */
    const int * incx,       /* array x, stride          */
    double * y,             /* array y                  */
    const int * incy        /* array y, stride          */
);

/* Adds alpha times column i to column j */
int add_column(double alpha, array2d_t *A, size_t i, size_t j) {
    if (!A || i >= A->shape[1] || j >= A->shape[1] || i == j) return 1;
    if (A->order == RowMajor) {
        daxpy_(&(int){A->shape[0]}, &alpha, A->val+i,
            &(int){A->shape[1]}, A->val+j, &(int){A->shape[1]});
    }
    else {
        daxpy_(&(int){A->shape[0]}, &alpha, A->val+i*A->shape[0],
            &(int){1}, A->val+j*A->shape[0], &(int){1});
    }
    return 0;
}

/* Adds alpha times row i to row j */
int add_row(double alpha, array2d_t *A, size_t i, size_t j) {

    if (!A || i >= A->shape[0] || j >= A->shape[0] || i == j) return 1;
    if (A->order == RowMajor) {
        daxpy_(&(int){A->shape[1]}, &alpha, A->val+i*A->shape[1],
            &(int){1}, A->val+j*A->shape[1], &(int){1});
    }
    else {
        daxpy_(&(int){A->shape[1]}, &alpha, A->val+i,
            &(int){A->shape[0]}, A->val+j, &(int){A->shape[0]});
    }
    return 0;
}
```

BLAS level 2: `dtrsv`

```c
#include <stdlib.h>
#include <math.h>
#include "array2d.h"

/** DTRSV
 *  BLAS level 2 routine for forward/back substitution
 *  Documentation: http://www.netlib.org/blas/#_level_2
 */
void dtrsv_(
  const char * uplo,  /* upper 'U' or lower 'L'               */
  const char * trans, /* not trans. 'N' or trans. 'T'         */
  const char * diag,  /* not unit diag. 'N' or unit diag. 'U' */
  const int * n,      /* dimension                            */
  const double * A,   /* column-major matrix of order n       */
  const int * lda,    /* leading dimension of A               */
  double * x,         /* right-hand side                      */
  const int * incx    /* stride for array x                   */
);

/** Solves system of equations L*U*x = b where
 *  L is unit lower triangular and U is upper triangular.
 *  The matrices L and U must be stored in a single array M
 *  of size n-by-n. On exit, the array b is overwritten by
 *  the solution x.
 *
 *  If successful, the function returns zero, and in case
 *  of an error, the return value is 1.
 *
 *  Inputs:
 *    M    dynamically allocated two-dimensional array of size n-by-n
 *    b    one-dimensional array of length n
 */
int lu_solve(array2d_t * M, double * b) {

  int incx=1;
  char uplo, trans, diag;

  /* Check inputs */
  if (!M || M->shape[0] != M->shape[1]) return 1;
  size_t n = M->shape[0];

  /* Check for singularity */
  for (size_t i=0;i<n;i++) {
      // Minimal check; room for improvements
      if (!isnormal(M->val[i+i*n])) return 1;
  }
```

```
  if (M->order == RowMajor) {
    /*
      Solve L*z = b
      If we interpret M as column-major storage of M',
      L' is stored in the upper triangular part of M'.
    */
    uplo = 'U'; trans = 'T'; diag = 'U';
    dtrsv_(&uplo,&trans,&diag,&(int){n},M->val,&(int){n},b,&incx);
    /*
      Solve U*x = z
      If we interpret M as column-major storage of M',
      U' is stored in the lower triangular part of M'.
    */
    uplo = 'L'; trans = 'T'; diag = 'N';
    dtrsv_(&uplo,&trans,&diag,&(int){n},M->val,&(int){n},b,&incx);
  }
  else {
    /* Solve L*z = b */
    uplo = 'L'; trans = 'N'; diag = 'U';
    dtrsv_(&uplo,&trans,&diag,&(int){n},M->val,&(int){n},b,&incx);
    /* Solve U*x = z */
    uplo = 'U'; trans = 'N'; diag = 'N';
    dtrsv_(&uplo,&trans,&diag,&(int){n},M->val,&(int){n},b,&incx);
  }

  return 0;
}
```