

```

/*
 * Dirichlet_basepoint.c
 *
 * The Dirichlet domain code is divided among several files. The header
 * file Dirichlet.h explains the organization of the three files, and
 * provides the common definitions and declarations.
 *
 * This file provides the function
 *
 * void maximize_the_injectivity_radius(
 *         MatrixPairList      *gen_list,
 *         Boolean              *basepoint_was_moved,
 *         DirichletInteractivity interactivity;
 *
 * Throughout this file, let "image height" refer to the height (i.e.
 * zeroth coordinate) of the image of the basepoint (1, 0, 0, 0) under
 * the action of a given matrix. The image height is the hyperbolic cosine
 * of the distance from the origin to its image. Note that the two matrices
 * of a MatrixPair have the same image height.
 *
 * maximize_the_injectivity_radius() assumes the MatrixPairs are given
 * in order of increasing image height. It moves the basepoint (i.e. it
 * conjugates the matrices) so that the smallest image height (excluding
 * the identity) is maximized subject to the constraint that no other image
 * height be less than it. This is a nonlinear programming problem. To
 * solve it, we replace it with its linear approximation, which is a linear
 * programming problem in the tangent space to  $H^3$  at (1, 0, 0, 0), and
 * then iterate the linear problem as many times as necessary to converge
 * to the true solution.
 *
 * Comments on saddle points.
 *
 * In three dimensions a saddle point may have
 *
 * (A) a 2-dimensionsal set of uphill directions and
 *     a 1-dimensional set of downhill directions, or
 *
 * (B) a 1-dimensionsal set of uphill directions and
 *     a 2-dimensional set of downhill directions.
 *
 * If the basepoint happens to be at a type (A) saddle point, this
 * code will notice and ask the user whether s/he would like to see
 * the Dirichlet domain based at that point or move on to a local
 * maximum of the injectivity radius.
 *
 * At present the code does not recognize type (B) saddle points;
 * it simply moves on. To write code to recognize type (B) saddle
 * points, one would need to check whether the gradients of all
 * the "h" functions are coplanar.
 *
 * Note: The check for a type (A) saddle point -- seeing whether
 * a constraint takes the form "objective function == 0" -- sometimes
 * flags a local maximum as a saddle. (Because even though the full
 * set of constraints define a local maximum, a subset defines a
 * saddle.) This is no big deal, just so you're aware it happens.
 *
 * The injectivity radius isn't differentiable (only piecewise
 * differentiable), but it's continuous, which is all that matters.
 *
 * End of comment on saddle points.
 *
 * Digression on choice of variables.
 *
 * One could work in terms of the actual distance from the basepoint
 * to its images instead of the image height. The latter is the
 * hyperbolic cosine of the former. In an abstract mathematical sense,
 * both approaches are of course equivalent. The questions are
 *
 * (1) which approach yields simpler code, and
 *
 * (2) which approach gives faster convergence.
 *
 * The image height approach is slightly simpler algebraically, and
 * therefore yields slightly simpler code. But the difference is

```

```

*      small, so if using the actual distances turned out to be faster,
*      that would be the best approach.  At the moment, I can't decide
*      which is likely to converge faster.  So I'll go with the simpler
*      image height approach, and perhaps later will do some experimental
*      tests to see which converges faster.
*
* End of digression.
*
*
* A translation through a distance dx in the x-direction is given
* exactly by the matrix
*
*      cosh dx   sinh dx   0   0
*      sinh dx   cosh dx   0   0
*      0         0         1   0
*      0         0         0   1
*
* For small dx, this matrix may be replaced with its linear approximation
*
*      1         dx       0   0
*      dx        1        0   0
*      0         0        1   0
*      0         0        0   1
*
* More generally, a translation (dx, dy, dz) has linear approximation
*
*      1         dx       dy       dz
*      dx        1        0        0
*      dy        0        1        0
*      dz        0        0        1
*
* Such matrices behave nicely under multiplication:  the dx, dy and dz
* terms simply add, as you would hope.
*
* Lemma.  If T is a translation matrix such as the one shown immediately
* above, then conjugating all matrices M in a covering transformation
* group G by T (i.e. replacing each M by (T^-1)(M)(T)), has the effect of
* moving the basepoint to (1, dx, dy, dz).  (The vector (1, dx, dy, dz) is
* expressed in the old coordinate system.  In the new coordinate system
* the basepoint will of course be (1, 0, 0, 0).)
*
* Proof.  Hmmm . . . visually this seems plausible, but at the moment
* I'm having trouble making my thoughts more precise.
*
*
* Conjugating an arbitrary group element with matrix M by the above
* linear approximation T to a translation matrix gives a matrix
* (T^-1)(M)(T) =
*
*      1  -dx -dy -dz   m00 m01 m02 m03   1  +dx +dy +dz
*      -dx  1   0   0   m10 m11 m12 m13   +dx  1   0   0
*      -dy  0   1   0   m20 m21 m22 m23   +dy  0   1   0
*      -dz  0   0   1   m30 m31 m32 m33   +dz  0   0   1
*
* whose image height, which is just the (0,0)th entry, is
*
*      h = m00 + (m01 - m10)dx + (m02 - m20)dy + (m03 - m30)dz.
*
* We don't care about the remaining fifteen entries.
*
* From the above expression for h, it's easy to read off the partial
* derivatives
*
*      dh/dx = m01 - m10
*      dh/dy = m02 - m20
*      dh/dz = m03 - m30
*/
#include "kernel.h"
#include "Dirichlet.h"
#include <stdlib.h>      /* needed for qsort() */

/*
* If an iteration of the linear programming algorithm moves the basepoint

```

```
* a distance less than BASEPOINT_EPSILON, we assume we've converged to
* a solution.
*/
#define BASEPOINT_EPSILON          (1e4 * DBL_EPSILON)

/*
 * In low precision, non general position situations we may never
 * get a solution accurate to within BASEPOINT_EPSILON. So we make the
 * convention that if we move a distance less than BIG_BASEPOINT_EPSILON
 * twice in a row, we assume we have converged.
 */
#define BIG_BASEPOINT_EPSILON      1e-5

/*
 * MAX_TOTAL_DISTANCE is the greatest cumulative distance we're willing
 * to move the basepoint before recomputing the Dirichlet domain to get
 * a fresh set of generators.
 */
#define MAX_TOTAL_DISTANCE         0.25

/*
 * The derivative of each image height is typically nonzero. If it's ever
 * less than DERIVATIVE_EPSILON, ask the user how to proceed.
 */
#define DERIVATIVE_EPSILON         1e-6

/*
 * MAX_STEP_SIZE defines the largest step size the linear programming
 * algorithm will take. If it's too large, the algorithm will go
 * beyond the region in which the linear approximation is meaningful.
 * If it's too small, the algorithm will converge unnecessarily slowly.
 */
#define MAX_STEP_SIZE              0.1

/*
 * The identity MatrixPair is recognized by the fact that its height
 * is less than 1.0 + IDENTITY_EPSILON. (Since  $\cosh(dx) \sim 1 + (1/2)dx^2$ ,
 * a MatrixPair which translates the origin a distance dx will have
 * height  $(1/2)dx^2$ .)
 */
#define IDENTITY_EPSILON            1e-6

/*
 * A Constraint is considered to be satisfied at a point iff it evaluates
 * to at most CONSTRAINT_EPSILON. If CONSTRAINT_EPSILON is too small,
 * bad things are likely to happen at points where more than three
 * constraint planes intersect. If CONSTRAINT_EPSILON is too large, we
 * could lose accuracy in delicate situations. The latter possibility
 * seems much less likely than the former (indeed we know the former occurs
 * for many of the most beautiful Dirichlet domains), so let's go with a
 * fairly large value of CONSTRAINT_EPSILON.
 */
#define CONSTRAINT_EPSILON          1e-6

/*
 * Two vectors u and v are considered parallel (resp. antiparallel) iff
 * the cosine of the angle between them is greater than 1.0 - SADDLE_EPSILON
 * (resp. less than -1.0 + SADDLE_EPSILON).
 */
#define SADDLE_EPSILON              1e-6

/*
 * A constraint is considered to have zero derivative iff the norm of
 * its gradient vector is less than ZERO_DERIV_EPSILON.
 */
#define ZERO_DERIV_EPSILON          1e-6

/*
 * A set of three linear equations in three variables is considered
 * degenerate iff some pivot has absolute value less than MIN_PIVOT.
 */
#define MIN_PIVOT                   (1e5 * DBL_EPSILON)
```

```

#define ROOT3OVER2                0.86602540378443864676

/*
 * We want to evaluate Constraints quickly, without the overhead of a
 * function call, but we don't want a lot of messy code. So let's define
 * a macro for testing the value of an equation at a given point.
 * This macro will also test the value of an ObjectiveFunction at a point.
 */
#define EVALUATE_EQN(eqn, pt) \
    (eqn[0]*pt[0] + eqn[1]*pt[1] + eqn[2]*pt[2] + eqn[3])

/*
 * An ObjectiveFunction is a 4-element vector (a, b, c, k).
 * The linear_programming() function tries to maximize
 * a*dx + b*dy + c*dz + k subject to the given constraints.
 */
typedef double ObjectiveFunction[4];

/*
 * A constraint is a 4-element vector (a, b, c, k)
 * interpreted as the inequality a*dx + b*dy + c*dz + k <= 0.
 */
typedef double Constraint[4];

/*
 * A solution is a vector (dx, dy, dz) which maximizes the
 * objective function subject to the constraints.
 */
typedef double Solution[3];

static int          count_matrix_pairs(MatrixPairList *gen_list);
static void          verify_gen_list(MatrixPairList *gen_list, int num_matrix_pairs);
static FuncResult    set_objective_function(ObjectiveFunction objective_function, MatrixPairList *matrix_pair);
static void          step_size_constraints(Constraint *constraints, ObjectiveFunction objective_function);
static void          regular_constraints(Constraint *constraints, MatrixPairList *gen_list, ObjectiveFunction objective_function, Boolean *may_be_saddle_point);
static void          linear_programming(ObjectiveFunction objective_function, int num_constraints, Constraint *constraints, Solution solution);
static Boolean       apex_is_higher(double height1, double height2, Solution apex1, Solution apex2);
static FuncResult    solve_three_equations(Constraint *equations[3], Solution solution);
static void          initialize_t2(Solution solution, O3Matrix t2);
static void          sort_gen_list(MatrixPairList *gen_list, int num_matrix_pairs);
static int CDECL     compare_image_height(const void *ptr1, const void *ptr2);
static double        length3(double v[3]);
static double        inner3(double u[3], double v[3]);
static void          copy3(Solution dest, const Solution source);

void maximize_the_injectivity_radius(
    MatrixPairList *gen_list,
    Boolean *basepoint_moved,
    DirichletInteractivity interactivity)
{
    int          num_matrix_pairs;
    double       distance_moved,
                prev_distance_moved,
                total_distance_moved;
    Boolean      keep_going;
    ObjectiveFunction objective_function;
    int          num_constraints;
    Constraint    *constraints;
    Solution      solution;
    Boolean      may_be_saddle_point,
                saddle_query_given;
    int          choice;

    static const Solution zero_solution = {0.0, 0.0, 0.0},
                        small_displacement = {0.001734, 0.002035, 0.000721};

    const static char    *saddle_message = "The basepoint may be at a saddle point of

```

```

the injectivity radius function.";
const static int      num_saddle_responses = 2;
const static char      *saddle_responses[2] = {
    "Continue On",
    "Stop Here and See Dirichlet Domain"};
const static int      saddle_default = 1;

const static char      *zero_deriv_message = "The derivative of the distance to the
closest translate of the basepoint is zero.";
const static char      num_zero_deriv_responses = 2;
const static char      *zero_deriv_responses[2] = {
    "Displace Basepoint and Continue On",
    "Stop Here and See Dirichlet Domain"};
const static int      zero_deriv_default = 1;

/*
 * Count the number of MatrixPairs.
 */
num_matrix_pairs = count_matrix_pairs(gen_list);

/*
 * Make sure that
 *
 * (1) the identity and at least two other MatrixPairs are present,
 *
 * (2) the MatrixPairs are in order of increasing height.
 *
 * Technical notes: We don't really need to have the gen_list
 * completely sorted -- it would be enough to have the identity come
 * first and the element of lowest image height come immediately after.
 * But I think the algorithm will run a tad faster if all elements are
 * in order of increasing image height. That way we get to the
 * meaningful constraints first.
 */
verify_gen_list(gen_list, num_matrix_pairs);

/*
 * Initialize *basepoint_moved to FALSE.
 * If we later move the basepoint, we'll set *basepoint_moved to TRUE.
 */
*basepoint_moved = FALSE;

/*
 * Keep track of the total distance we've moved the basepoint.
 * We don't want to go too far without recomputing the Dirichlet
 * domain to get a fresh set of group elements.
 */
total_distance_moved = 0.0;

/*
 * Some ad hoc code for handling low precision situations
 * needs to keep track of the prev_distance_moved.
 */
prev_distance_moved = DBL_MAX;

/*
 * We don't want to bother the user with the saddle query
 * more than once. We initialize saddle_query_given to FALSE,
 * and then set it to TRUE if and when the query takes place.
 */
saddle_query_given = FALSE;

/*
 * We want to move the basepoint to a local maximum of the injectivity
 * radius function. Solve the linear approximation to this problem,
 * and repeat until a solution is found.
 */
do
{
    /*
     * Set the objective function using the first nonidentity matrix
     * on the list. If the derivative of the objective function is
     * nonzero, proceed normally. Otherwise ask the user how s/he

```

```

    * would like to proceed.
    */

    if (set_objective_function(objective_function, gen_list->begin.next->next) ==
func_OK)
    {
        /*
        * Allocate space for the Constraints.
        * There'll be num_matrix_pairs - 2 regular constraints
        * (one for each MatrixPair, excluding the identity and the
        * MatrixPair used to define the objective function),
        * preceded by three constraints which limit the step size
        * to MAX_STEP_SIZE.
        */
        num_constraints = (num_matrix_pairs - 2) + 3;
        constraints = NEW_ARRAY(num_constraints, Constraint);

        /*
        * Set up the three step size constraints.
        */
        step_size_constraints(constraints, objective_function);

        /*
        * Set up the regular constraints.
        */
        regular_constraints(constraints, gen_list, objective_function, &
may_be_saddle_point);

        /*
        * If we're not near an apparent saddle point,
        * do the linear programming.
        */
        if (may_be_saddle_point == FALSE)
            linear_programming(objective_function, num_constraints, constraints,
solution);
        /*
        * Otherwise ask the user whether s/he would like
        * to continue on normally or stop here.
        */
        else
        {
            switch (interactivity)
            {
                case Dirichlet_interactive:
                    if (saddle_query_given == FALSE)
                    {
                        choice = uQuery(    saddle_message,
                                           num_saddle_responses,
                                           saddle_responses,
                                           saddle_default);

                        saddle_query_given = TRUE;
                    }
                    else
                        choice = 0; /* continue on */
                    break;

                case Dirichlet_stop_here:
                    choice = 1; /* stop here */
                    break;

                case Dirichlet_keep_going:
                    choice = 0; /* continue on */
                    break;
            }

            switch (choice)
            {
                case 0:
                    /*
                    * Continue on normally.
                    */
                    linear_programming(objective_function, num_constraints, constraints
, solution);
                    break;

```

```

        case 1:
            /*
             * Stop here, set *basepoint_moved to FALSE
             * to force an exit from the loop, and look at
             * the Dirichlet domain.
             */
            copy3(solution, zero_solution);
            *basepoint_moved = FALSE;
            break;
    }
}

/*
 * Free the Constraint array.
 */
my_free(constraints);
}

else
{
    /*
     * The derivative of the objective function is zero.
     *
     * Ask the user whether to use this basepoint, or move on in
     * search of a local maximum of the injectivity radius.
     */
    switch (interactivity)
    {
        case Dirichlet_interactive:
            choice = uQuery(    zero_deriv_message,
                               num_zero_deriv_responses,
                               zero_deriv_responses,
                               zero_deriv_default);

            break;

        case Dirichlet_stop_here:
            choice = 1; /* stop here */
            break;

        case Dirichlet_keep_going:
            choice = 0; /* continue on */
            break;
    }

    switch (choice)
    {
        case 0:
            /*
             * Displace the basepoint and continue on.
             */
            copy3(solution, small_displacement);
            break;

        case 1:
            /*
             * We want to stay at this point, so set the solution
             * to (0, 0, 0), and set *basepoint_moved to FALSE
             * to force an exit from the loop.
             */
            copy3(solution, zero_solution);
            *basepoint_moved = FALSE;
            break;
    }
}

/*
 * Use the solution to conjugate the MatrixPairs.
 */
conjugate_matrices(gen_list, solution);

/*
 * Resort the gen_list according to increasing image height.
 */

```

```

    sort_gen_list(gen_list, num_matrix_pairs);

    /*
     * How far was the basepoint moved this time?
     */
    distance_moved = length3(solution);

    /*
     * What is the total distance we've moved the basepoint?
     */
    total_distance_moved += distance_moved;

    /*
     * If the basepoint moved any meaningful distance,
     * set *basepoint_moved to TRUE.
     */
    if (distance_moved > BASEPOINT_EPSILON)
    {
        *basepoint_moved = TRUE;
        /*
         * If we move too far from the original basepoint, we should
         * recompute the Dirichlet domain to get a fresh set of
         * group elements. Otherwise we keep going.
         */
        keep_going = (total_distance_moved < MAX_TOTAL_DISTANCE);
    }
    else
        keep_going = FALSE;

    /*
     * The preceding code works great when the constraints are
     * either in general position, or are given to moderately high
     * precision. But for low precision, non general position
     * constraints (e.g. for an 8-component circular chain with no
     * twist), the algorithm can knock around forever making
     * changes on the order of 1e-9. The following code lets the
     * algorithm terminate in those cases.
     */
    if (prev_distance_moved < BIG_BASEPOINT_EPSILON
        && distance_moved < BIG_BASEPOINT_EPSILON)
    {
        /*
         * For sure we don't want to keep going after making
         * two fairly small changes in a row.
         */
        keep_going = FALSE;
        /*
         * If the total_distance_moved is less than BIG_BASEPOINT_EPSILON,
         * then we want to set *basepoint_moved to FALSE to prevent
         * recomputation of the Dirichlet domain. Note that we still
         * allow the possibility that *basepoint_moved is TRUE even when
         * the total_distance_moved is greater than BIG_BASEPOINT_EPSILON,
         * as could happen if preceding code artificially set *basepoint_moved
         * to FALSE to force an exit from the loop.
         */
        if (total_distance_moved < BIG_BASEPOINT_EPSILON)
            *basepoint_moved = FALSE;
    }
    prev_distance_moved = distance_moved;
} while (keep_going == TRUE);
}

static int count_matrix_pairs(
    MatrixPairList *gen_list)
{
    int num_matrix_pairs;
    MatrixPair *matrix_pair;

    num_matrix_pairs = 0;

    for ( matrix_pair = gen_list->begin.next;
          matrix_pair != &gen_list->end;

```



```

        matrix_pair = matrix_pair->next)

    num_matrix_pairs++;

    return num_matrix_pairs;
}

static void verify_gen_list(
    MatrixPairList *gen_list,
    int num_matrix_pairs)
{
    MatrixPair *matrix_pair;

    /*
     * Does the list have at least two elements beyond the identity?
     * Algebraically, we'll need an objective function
     * and at least one constraint.
     * Geometrically, the Dirichlet domain which provided these
     * generators must have at least two pairs of faces.
     */

    if (num_matrix_pairs < 2)
        uFatalError("verify_gen_list", "Dirichlet_basepoint");

    /*
     * The first MatrixPair on gen_list should be the identity.
     */

    if (gen_list->begin.next->height > 1.0 + IDENTITY_EPSILON)
        uFatalError("verify_gen_list", "Dirichlet_basepoint");

    /*
     * We want the MatrixPairs to be in order of increasing image height.
     * (Note that this loop starts at the second MatrixPair on the list.)
     */

    for (    matrix_pair = gen_list->begin.next->next;
           matrix_pair != &gen_list->end;
           matrix_pair = matrix_pair->next)

        if (matrix_pair->height < matrix_pair->prev->height)

            uFatalError("verify_gen_list", "Dirichlet_basepoint");
}

static FuncResult set_objective_function(
    ObjectiveFunction objective_function,
    MatrixPair *matrix_pair)
{
    int i;

    /*
     * Read the objective function from the first matrix on the list.
     * Recall from above that we want to maximize
     *
     * 
$$h = m_{00} + (m_{01} - m_{10})dx + (m_{02} - m_{20})dy + (m_{03} - m_{30})dz.$$

     *
     * Note that the object function is the same for matrix_pair[0] and
     * matrix_pair[1], because they are inverses. (This follows from the
     * rule for computing inverses in O(3,1) (see o3l_invert() in
     * o3l_matrices.c), as well as from the geometrical fact that an
     * isometry and its inverse must translate the basepoint equal amounts.)
     *
     * Return
     * func_OK if the objective function's derivative is nonzero, or
     * func_failed if it isn't.
     */

    /*
     * Compute the objective function.
     */

```

```

    for (i = 0; i < 3; i++)
        objective_function[i] = matrix_pair->m[0][0][i+1] - matrix_pair->m[0][i+1][0];

    objective_function[3] = matrix_pair->m[0][0][0];

    /*
     * Check whether the derivative is nonzero.
     */

    if (length3(objective_function) > DERIVATIVE_EPSILON)
        return func_OK;
    else
        return func_failed;
}

static void step_size_constraints(
    Constraint *constraints,
    ObjectiveFunction objective_function)
{
    int i,
        j,
        i0;
    double v[3][3],
           w[3][3],
           max_abs,
           length;

    /*
     * The three step size constraints will be faces of a cube,
     * with the vector for the objective_function pointing in the
     * direction of the cube's corner where the three faces intersect.
     */

    /*
     * First find an orthonormal basis {v[0], v[1], v[2]} with
     * v[0] equal to the vector part of the objective function.
     */

    /*
     * Let v[0] be the vector part of the objective function,
     * normalized to have length one. (Elsewhere we checked that
     * its length is at least DERIVATIVE_EPSILON.)
     */
    length = length3(objective_function);
    for (i = 0; i < 3; i++)
        v[0][i] = objective_function[i] / length;

    /*
     * Let v[1] be a unit vector orthogonal to v[0].
     */

    /*
     * Let i0 be the index of the component of v[0] which has the greatest
     * absolute value. (In particular, its absolute value is sure to be
     * nonzero.)
     */
    max_abs = 0.0;
    for (i = 0; i < 3; i++)
        if (fabs(v[0][i]) > max_abs)
        {
            i0 = i;
            max_abs = fabs(v[0][i]);
        }

    /*
     * Write down a nonzero v[1] orthogonal to v[0] . . .
     */
    v[1][i0] = -v[0][(i0+1)%3] / v[0][i0];
    v[1][(i0+1)%3] = 1.0;
    v[1][(i0+2)%3] = 0.0;

    /*

```

```

    *   . . . and normalize its length to 1.0.
    */
    length = length3(v[1]);
    for (i = 0; i < 3; i++)
        v[1][i] /= length;

    /*
    *   Let v[2] = v[0] x v[1].
    */
    for (i = 0; i < 3; i++)
        v[2][i] = v[0][(i+1)%3] * v[1][(i+2)%3] - v[0][(i+2)%3] * v[1][(i+1)%3];

    /*
    *   Use the orthonormal basis {v[0], v[1], v[2]} to find another basis
    *   {w[0], w[1], w[2]} whose elements symmetrically surround v[0].
    *
    *       w[0] = v[0] + v[1]
    *       w[1] = v[0] + ( -1/2 v[1] + sqrt(3)/2 v[2] )
    *       w[2] = v[0] + ( -1/2 v[1] - sqrt(3)/2 v[2] )
    */
    for (j = 0; j < 3; j++)
    {
        w[0][j] = v[0][j] + v[1][j];
        w[1][j] = v[0][j] + (-0.5*v[1][j] + ROOT3OVER2*v[2][j]);
        w[2][j] = v[0][j] + (-0.5*v[1][j] - ROOT3OVER2*v[2][j]);
    }

    /*
    *   Use the basis {w[0], w[1], w[2]} to write down
    *   the three step size constraints.
    *
    *   Technical note: If you move in the direction of the objective
    *   function vector v[0], the three constraints will be exactly
    *   satisfied at a distance MAX_STEP_SIZE from the origin. That is, the
    *   inner product of (MAX_STEP_SIZE, 0, 0) with each of (1, 1, 0),
    *   (1, -1/2, sqrt(3)/2) and (1, -1/2, -sqrt(3)/2) is exactly
    *   MAX_STEP_SIZE. However, if you move to the side (orthogonally to
    *   v[0]) it's possible to move a distance 2*MAX_STEP_SIZE. E.g. the
    *   inner product of (0, -2*MAX_STEP_SIZE, 0) with each of
    *   (1, 1, 0), (1, -1/2, sqrt(3)/2) and (1, -1/2, -sqrt(3)/2) is
    *   -2*MAX_STEP_SIZE, MAX_STEP_SIZE and MAX_STEP_SIZE, respectively, so
    *   it satisfies all three constraints. But typically we won't be
    *   moving to the side, and in any case all we really care about anyhow
    *   is the order of magnitude of MAX_STEP_SIZE. A factor of two isn't
    *   important.
    */
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            constraints[i][j] = w[i][j];
        constraints[i][3] = -MAX_STEP_SIZE;
    }
}

static void regular_constraints(
    Constraint      *constraints,
    MatrixPairList *gen_list,
    ObjectiveFunction objective_function,
    Boolean         *may_be_saddle_point)
{
    /*
    *   The documentation at the top of this file shows that the image
    *   height h corresponding to a matrix m is
    *
    *       h = m00 + (m01 - m10)dx + (m02 - m20)dy + (m03 - m30)dz.
    *
    *   Each regular constraint will say that the image height h for the
    *   given matrix must remain greater than the image height h' of the
    *   matrix used for the objective function. In symbols, h >= h'.
    *   Since a Constraint says that some quantity must remain negative,
    *   we express the constraint as h' - h <= 0.
    *
    *   The Boolean *may_be_saddle_point will be set to TRUE if some

```

```

    * constraint suggests a saddle point. Otherwise it gets set to FALSE.
    */

    int i;
    MatrixPair *matrix_pair;
    Constraint *constraint;
    double h[4],
           c;

    /*
     * Assume we're not at a saddle point unless we encounter
     * evidence to the contrary.
     */
    *may_be_saddle_point = FALSE;

    /*
     * Skip the identity and the MatrixPair used to define the objective
     * function, and begin with the next MatrixPair on the list.
     * Write a constraint for it and each successive MatrixPair.
     *
     * Skip the first three Constraints in the constraints array.
     * They contain the step size constraints.
     */

    for ( matrix_pair = gen_list->begin.next->next->next,
          constraint = constraints + 3;

          matrix_pair != &gen_list->end;

          matrix_pair = matrix_pair->next,
          constraint++)
    {
        /*
         * Compute h. (As explained in set_objective_function(),
         * it doesn't matter which of the two matrices we use.)
         */

        for (i = 0; i < 3; i++)
            h[i] = matrix_pair->m[0][0][i+1] - matrix_pair->m[0][i+1][0];

        h[3] = matrix_pair->m[0][0][0];

        /*
         * Set the constraint to h' - h.
         * (The objective function is h' in the above notation.)
         */

        for (i = 0; i < 4; i++)
            (*constraint)[i] = objective_function[i] - h[i];

        /*
         * Does the constraint plane pass through the origin?
         */

        if ((*constraint)[3] > - CONSTRAINT_EPSILON)
        {
            /*
             * Does the constraint have nonzero derivative?
             * If not, then h and h' must have equal but nonzero derivatives.
             * We know h' has nonzero derivative because we checked it
             * when we computed the objective function.
             * Its OK for h and h' to have equal but nonzero derivatives --
             * it simply means that as we move avoid from the closest
             * translate of the basepoint, we're moving away from some
             * other translate as well -- be we don't want to divide by
             * length3(*constraint).
             */

            if (length3(*constraint) > ZERO_DERIV_EPSILON)
            {
                /*
                 * Check whether the constraint plane is parallel
                 * to the level sets of the objective function.
                 */
            }
        }
    }

```

```

        *   Use the formula  $\langle u, v \rangle = |u| |v| \cos(\text{angle})$ .
        */

    c = inner3(objective_function, *constraint) /
        (length3(objective_function) * length3(*constraint));

    /*
     *   If it is parallel, set *may_be_saddle_point to TRUE.
     */

    if (fabs(c) > 1.0 - SADDLE_EPSILON)
        *may_be_saddle_point = TRUE;

    /*
     *   If necessary we could be more sophisticated at this point,
     *   and check whether the gradients of h and h' are parallel
     *   or antiparallel. Typically one expects them to be
     *   antiparallel (the MatrixPairs are, after all, the face
     *   pairings of a Dirichlet domain, so we don't have to worry
     *   about squares of a matrix), but if they were parallel one
     *   might want to ask which is longer (depending on which is
     *   longer, you will or will not be able to move the basepoint
     *   in that direction).
     */
    }
}
}

static void linear_programming(
    ObjectiveFunction    objective_function,
    int                  num_constraints,
    Constraint            *constraints,
    Solution              solution)
{
    int                  i,
                        j,
                        k;

    Constraint            *active_constraints[3],
                        *new_constraints[3];
    Solution              apex,
                        new_apex,
                        max_apex;
    double                apex_height,
                        new_height,
                        max_height;
    int                   inactive_constraint_index;

    /*
     *   Initialize the three active_constraints to be the first three
     *   constraints on the list. (In the present context these are the
     *   step size constraints, but let's write the code so as not to
     *   rely on this knowledge.) Visually, we think of the intersection
     *   of the halfspaces defined by the three active_constraints as
     *   a pyramid, oriented so that the gradient of objective function
     *   points up.
     */
    for (i = 0; i < 3; i++)
        active_constraints[i] = constraints + i;

    /*
     *   Initialize the apex to be the vertex defined by the intersection
     *   of the three step size constraints.
     *
     *   Important note: We assume the maximum value for the objective
     *   function, subject to the active_constraints, occurs at the apex.
     *   (In the present context this is true by virtue of the way the
     *   step size constraints were written.)
     */
    if (solve_three_equations(active_constraints, apex) == func_failed)
        uFatalError("linear_programming", "Dirichlet_basepoint");
}

```

```

    * For future reference, set apex_height to the value of the
    * objective function at the apex.
    */
    apex_height = EVALUATE_EQN(objective_function, apex);

    /*
    * Go down the full list of constraints and see whether the apex
    * satisfies all of them.
    *
    * If it does, we've solved the linear programming problem
    * and we're done.
    *
    * If it doesn't, then slice the pyramid defined by the
    * active_constraints with the constraint which isn't satisfied.
    * Let the new apex be the highest point on the truncated pyramid,
    * and the new active_constraints be the three faces of the truncated
    * pyramid incident to the new apex. Repeat this procedure until
    * all constraints are satisfied. Note that we have to start from
    * the beginning of the constraint list each time, since even if the
    * old apex satisfied a given constraint, the new apex might not.
    *
    * If candidate apexes are always at distinct heights, then it's easy
    * to prove that this algorithm will terminate in a finite number of
    * steps. But if different candidate apexes sometimes lie at the
    * same height (as would happen if a constraint function were parallel
    * to the level sets of the objective function, for example) then we
    * cannot prove that the algorithm terminates. Indeed, the example
    * given in the documentation at the end of this function shows how
    * the algorithm might get into an infinite loop. To avoid this
    * problem, we add an infinitesimal perturbation to the objective
    * function. Say we add a term epsilon*dx to the objective function,
    * where epsilon is a true mathematical infinitesimal, not just a very
    * small number. Then if two heights are precisely equal as floating
    * point numbers, the one with the greater dx coordinate will be
    * considered greater than the one with the smaller dx coordinate.
    * But what if their dx coordinates are equal, too? And dy and dz?
    * Our official theoretical definition for the objection function is
    *
    *      (objective function as computed)    + dx*epsilon
    *                                           + dy*(epsilon^2)
    *                                           + dz*(epsilon^3)
    *
    * In practice, this means that we first compare heights based on the
    * objective function as computed. If they come out exactly equal,
    * then we compare based on dx coordinates. If the dx's are equal,
    * then we compare dy's. If the dy's are equal we compare dz's. If
    * all those things are equal, then the points coincide, and it makes
    * sense that their heights should be equal. Since there are only
    * finite number of possible apexes (one for each triple of constraints),
    * it follows that if the height is reduced at each step, then the
    * algorithm must terminate after a finite number of steps.
    */

    for (i = 0; i < num_constraints; i++)
        if (EVALUATE_EQN(constraints[i], apex) > CONSTRAINT_EPSILON)
        {
            /*
            * Uh-oh. The apex doesn't satisfy constraints[i].
            * Slice the pyramid with constraints[i], and see which
            * new vertex is highest. The new set of active constraints
            * will include two of the old active constraints, plus
            * constraints[i].
            */

            /*
            * Initialize max_height to -1.0.
            */
            max_height = -1.0;
            for (j = 0; j < 3; j++)
                max_apex[j] = 0.0;

            /*
            * Replace each active_constraint, in turn, with constraints[i].

```

```

    * The variable j will be the index of the active_constraint
    * currently being replaced with constraints[i].
    */

for (j = 0; j < 3; j++)
{
    /*
     * Assemble the candidate set of new_constraints.
     */
    for (k = 0; k < 3; k++)
        new_constraints[k] =
/* Cater to a DEC compiler error that chokes on &(array)[i] */
/* (k == j ? &constraints[i] : active_constraints[k]); */
        (k == j ? constraints + i : active_constraints[k]);

    /*
     * Find the common intersection of the new_constraints.
     *
     * The equations can't possibly be underdetermined, because
     * if the solution set were 1-dimensional it would have to
     * include the apex, which is known not to satisfy
     * constraints[i].
     *
     * The equations might, however, be inconsistent. In
     * this case, we continue with the loop, as if new_height
     * were greater than apex_height (cf. below). If the
     * equations are in principle inconsistent but roundoff
     * error gives a solution, that's OK too: the solution
     * will yield a new_height near +infinity or -infinity,
     * and new_apex will be ignored or fail to be maximal,
     * respectively.
     */
    if (solve_three_equations(new_constraints, new_apex) == func_failed)
        continue;

    /*
     * Compute the value of the objective function
     * at the new apex.
     */
    new_height = EVALUATE_EQN(objective_function, new_apex);

    /*
     * If new_height is greater than apex_height, then new_apex
     * is above apex, and is not actually a vertex of the
     * truncated pyramid. We ignore it and move on. (It's
     * easy to prove that *some* j will yield a valid maximum
     * height. When we slice the pyramid with constraints[i]
     * the resulting solid will somewhere obtain a maximum
     * height. The old apex is gone, so it can't be there.
     * And the origin is still present, so it must be higher
     * than the origin. The infinitesimal correction to the
     * objective function insures that no planes or lines are
     * ever truly horizontal, so the maximum must occur at a
     * vertex, where constraints[i] and two of the old
     * constraints intersect.)
     *
     * If new_height == apex_height, apex_is_higher() applies
     * the infinitesimal correction to the objective function,
     * as explained above.
     */
    if (apex_is_higher(new_height, apex_height, new_apex, apex) == TRUE)
        continue;

    /*
     * Is new_height greater than max_height?
     */
    if (apex_is_higher(new_height, max_height, new_apex, max_apex) == TRUE)
    {
        inactive_constraint_index = j;
        max_height = new_height;
        for (k = 0; k < 3; k++)
            max_apex[k] = new_apex[k];
    }
}

```

```

    /*
     * Swap constraints[i] into the active_constraints array
     * at index inactive_constraint_index.
     */
/* Cater to a DEC compiler error that chokes on &(array)[i] */
/* active_constraints[inactive_constraint_index] = &constraints[i]; */
   active_constraints[inactive_constraint_index] = constraints + i;

    /*
     * Set the apex to max_apex and apex_height to max_height.
     *
     * Note that we preserve the condition (cf. above) that the
     * maximum value of the objective function on the pyramid
     * occurs at the apex.
     */
    for (j = 0; j < 3; j++)
        apex[j] = max_apex[j];
    apex_height = max_height;

    /*
     * We've fixed up the active_constraints and the apex,
     * so now recheck the other constraints. Set i = -1,
     * so that after the i++ in the for(;;) statement it will
     * be back to i = 0.
     */
    i = -1;
}

/*
 * Hooray. All the constraints are satisfied.
 * Set the solution equal to the apex and we're done.
 */
for (i = 0; i < 3; i++)
    solution[i] = apex[i];

return;

/*
 * Here's an example in which an unperturbed objective function
 * may lead to an infinite loop.
 *
 * Make a sketch (in (dx, dy, dz) space) showing the following points:
 *
 *      s0 = (2, 0, 0)
 *      s1 = (-1, sqrt(3), 0) [twice a primitive cube root of unity]
 *      s2 = (-1, -sqrt(3), 0)
 *
 *      t0 = (1, 0, 1)
 *      t1 = (-1, sqrt(3)/2, 1)
 *      t2 = (-1, -sqrt(3)/2, 1)
 *
 *      u = (0, 0, 2)
 *
 * The objective function is 0*dx + 0*dy + 1*dz + constant.
 *
 * The constraints are defined by the following planes. (I'll give
 * a spanning set for each plane. The constraint itself will be an
 * equation saying that (dx, dy, dz) must lie on the same side of the
 * plane as the origin (0, 0, 0) (or on the plane itself is OK too.)
 * Sketch each of these planes in your picture.
 *
 *      a0 = {s1, s2, u}
 *      a1 = {s2, s0, u}
 *      a2 = {s0, s1, u}
 *
 *      b = {t0, t1, t2}
 *
 *      c0 = {s0, t1, t2}
 *      c1 = {t0, s1, t2}
 *      c2 = {t0, t1, s2}
 *
 * Now look what happens in the naive linear programming algorithm.
 * Say we start with constraints a0, a1 and a2, so the apex is at

```



```

    * point u. Then we consider constraint b. Constraint b will
    * replace one of {a0, a1, a2}; w.l.o.g. say it's a2. So we're
    * left with constraints {a0, a1, b} and the apex is at t2. The point
    * t2 satisfies all the constraints except c2, so eventually the
    * algorithm will swap c2 for either a0 or a1 (w.l.o.g. say its a1)
    * and we're left with equations {a0, c2, b}, and the apex moves to
    * t1. The point t1 satisfies all the constraints except c1, so
    * c1 gets swapped for either a0 or c2, the constraint set becomes
    * either {c1, c2, b} or {a0, c1, b}, and the apex moves to either
    * t0 or t2. This is the critical juncture for the algorithm. If
    * it swapped c1 for a0, then on the next iteration of the algorithm
    * it swaps c0 for b, and the constraint set {c1, c2, c0} gives us
    * the true solution. But if it swapped c1 for c2, then we run the
    * risk of getting into an infinite loop. But with the perturbed
    * objective function this will never happen, because we'll never
    * visit the same vertex twice.
    */
}

```

```

static Boolean apex_is_higher(
    double      height1,
    double      height2,
    Solution     apex1,
    Solution     apex2)
{
    int i;

    if (height1 > height2)
        return TRUE;
    if (height1 < height2)
        return FALSE;

    for (i = 0; i < 3; i++)
    {
        if (apex1[i] > apex2[i])
            return TRUE;
        if (apex1[i] < apex2[i])
            return FALSE;
    }

    return FALSE;
}

```

```

static FuncResult solve_three_equations(
    Constraint *equations[3],
    Solution    solution)
{
    /*
    * If the system of equations has a unique solution,
    * write it into the solution parameter and return func_OK.
    *
    * Otherwise return func_failed.
    */

    int      r,
            c,
            p;
    double   equation_storage[3][4],
            *eqn[3],
            *temp,
            pivot_value;

    /*
    * We store the set of equations as an array of three pointers,
    * to facilitate easy row swapping. Initialize eqn[i] to point
    * to equation_storage[i].
    */
    for (r = 0; r < 3; r++)
        eqn[r] = equation_storage[r];

    /*
    * Copy the original equations to avoid trashing them.
    */
}

```

```

    * Note that the constants are in eqn[r][3].
    */
    for (r = 0; r < 3; r++)
        for (c = 0; c < 4; c++)
            eqn[r][c] = (*equations[r])[c];

    /*
    * Do the forward part of Gaussian elimination.
    */

    /*
    * For each pivot position eqn[p][p] . . .
    */
    for (p = 0; p < 3; p++)
    {
        /*
        * Find the pivot row and swap it with row p.
        */
        for (r = p + 1; r < 3; r++)
            if (fabs(eqn[r][p]) > fabs(eqn[p][p]))
            {
                temp = eqn[p];
                eqn[p] = eqn[r];
                eqn[r] = temp;
            }

        /*
        * Note the pivot value.
        */
        pivot_value = eqn[p][p];

        /*
        * If the pivot_value is close to zero,
        * the equations won't have a stable, unique solution.
        * Return func_failed.
        */
        if (fabs(pivot_value) < MIN_PIVOT)
            return func_failed;

        /*
        * Divide the pivot row through by the pivot_value.
        *
        * The entries with c <= p needn't be computed,
        * since we know what they are.
        */
        for (c = p + 1; c < 4; c++)
            eqn[p][c] /= pivot_value;

        /*
        * Subtract multiples of row p from all subsequent rows,
        * so that only zeros appear below the pivot entry.
        *
        * The entries with c <= r needn't be computed, since
        * we know they'll all be zero. However, if we were
        * explicitly changing eqn[r][p], then we'd have to save
        * a copy of its initial value in a separate variable,
        * e.g. we'd replace
        *
        *         eqn[r][c] -= eqn[r][p] * eqn[p][c];
        * with
        *         eqn[r][c] -= multiple * eqn[p][c];
        */
        for (r = p + 1; r < 3; r++)
            for (c = p + 1; c < 4; c++)
                eqn[r][c] -= eqn[r][p] * eqn[p][c];
    }

    /*
    * Do the back substitution part of Gaussian elimination.
    *
    * We needn't bother computing the zeros which goes in the matrix.
    * We just need the constants.
    */
    for (c = 3; --c >= 0; )

```

```

    for (r = c; --r >= 0; )
        eqn[r][3] -= eqn[r][c] * eqn[c][3];

/*
 * Read off the solution.
 *
 * Note that the constants sit on the left side of the constraint
 * equations, so Gaussian elimination has brought us to the system
 *
 *      1.0 dx + 0.0 dy + 0.0 dz + eqn[0][3] = 0.0
 *      0.0 dx + 1.0 dy + 0.0 dz + eqn[1][3] = 0.0
 *      0.0 dx + 0.0 dy + 1.0 dz + eqn[2][3] = 0.0
 *
 * so
 *
 *      dx = - eqn[0][3]
 *      dy = - eqn[1][3]
 *      dz = - eqn[2][3]
 *
 * Note the minus signs.
 */
for (r = 0; r < 3; r++)
    solution[r] = - eqn[r][3];

return func_OK;
}

void conjugate_matrices(
    MatrixPairList *gen_list,
    double displacement[3])
{
/*
 * We want to conjugate each MatrixPair on the gen_list so as to move
 * the basepoint the distance given by the displacement. The
 * displacement, which is a translation (dx, dy, dz) in the tangent
 * space to H^3 at (1, 0, 0, 0), is a linear approximation to where the
 * basepoint ought to be.
 *
 * It won't do simply to conjugate by the matrix T1 =
 *
 *      1      dx      dy      dz
 *      dx      1      0      0
 *      dy      0      1      0
 *      dz      0      0      1
 *
 * Even though T1 is a linear approximation to the desired translation,
 * it's columns are not quite orthonormal (they are orthonormal to a
 * first order approximation, but not exactly). We need to use a
 * translation matrix which is exactly orthonormal, so that the
 * MatrixPairs on the gen_list remain elements of O(3,1) to full
 * accuracy.
 *
 * One approach would be to apply the Gram-Schmidt process to find an
 * element of O(3,1) close to T1. The problem here is that the
 * Gram-Schmidt process itself may introduce additional error.
 *
 * Better to start instead with a second order approximation to the
 * translation matrix, given by T2 =
 *
 *      1 + (dx^2 + dy^2 + dz^2)/2      dx      dy      dz
 *      dx      1 + (dx^2)/2      (dx dy)/2      (dx dz)/2
 *      dy      (dx dy)/2      1 + (dy^2)/2      (dy dz)/2
 *      dz      (dx dz)/2      (dy dz)/2      1 + (dz^2)/2
 *
 * Note that T2 is actually orthonormal to third order; that is,
 * its columns fail to be orthonormal only by fourth order terms.
 *
 * We will start with the second order approximation T2 and apply
 * Gram-Schmidt to it. The correction terms -- all of fourth order --
 * will not significantly affect the closeness of the approximation.
 *
 * Digression.
 */
}

```

```

*      You may be wondering where the matrix T2 came from.
*      Drop down a dimension, and consider the product of a translation
*      (dx, 0) and a translation (0, dy). The result you get depends
*      on the order of the factors.
*
*      ( 1  dx  0) ( 1  0  dy)   ( 1  dx  dy )
*      ( dx  1  0) ( 0  1  0 ) = ( dx  1  dxdy)
*      ( 0  0  1) ( dy  0  1 )   ( dy  0  1 )
*
*      ( 1  0  dy) ( 1  dx  0)   ( 1  dx  dy )
*      ( 0  1  0 ) ( dx  1  0 ) = ( dx  1  0 )
*      ( dy  0  1 ) ( 0  0  1 )   ( dy  dxdy 1 )
*
*      Averaging the two results leads to the matrix T2 shown above.
*
*      End of digression.
*/

O31Matrix    t2;
MatrixPair   *matrix_pair;

/*
 * Initialize t2 to be the second order approximation shown above.
 */
initialize_t2(displacement, t2);

/*
 * Apply the Gram-Schmidt process to bring t2 to a nearby element
 * of O(3,1).
 */
o31_GramSchmidt(t2);

/*
 * For each MatrixPair on gen_list . . .
 */
for (matrix_pair = gen_list->begin.next;
     matrix_pair != &gen_list->end;
     matrix_pair = matrix_pair->next)
{
    /*
     * Conjugate m[0] by t2.
     * That is, replace m[0] by (t2^-1) m[0] t2.
     */
    o31_conjugate(matrix_pair->m[0], t2, matrix_pair->m[0]);

    /*
     * Recompute m[1] as the inverse of m[0].
     */
    o31_invert(matrix_pair->m[0], matrix_pair->m[1]);

    /*
     * Set the height.
     */
    matrix_pair->height = matrix_pair->m[0][0][0];
}

static void initialize_t2(
    Solution    solution,
    O31Matrix    t2)
{
    /*
     * Initialize t2 to be the second order approximation to the
     * translation matrix, as explain in conjugate_matrices() above.
     *
     * This code is not optimized for computational efficiency, but
     * that's OK because it's called only once for each iteration
     * of the algorithm.
     */

    int        i,
               j;

```

```

/*
 * Set the linear terms.
 */
for (i = 0; i < 3; i++)
    t2[0][i+1] = t2[i+1][0] = solution[i];

/*
 * Set t2[0][0].
 */
t2[0][0] = 1.0;
for (i = 0; i < 3; i++)
    t2[0][0] += 0.5 * solution[i] * solution[i];

/*
 * Set the remaining second order terms.
 */
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        t2[i+1][j+1] = (i == j ? 1.0 : 0.0)
            + 0.5 * solution[i] * solution[j];
}

static void sort_gen_list(
    MatrixPairList *gen_list,
    int num_matrix_pairs)
{
    MatrixPair **array,
               *matrix_pair;
    int i;

    /*
     * Allocate an array to hold the addresses of the MatrixPairs.
     */
    array = NEW_ARRAY(num_matrix_pairs, MatrixPair *);

    /*
     * Copy the addresses into the array.
     */
    for (matrix_pair = gen_list->begin.next,
         i = 0;
         matrix_pair != &gen_list->end;
         matrix_pair = matrix_pair->next,
         i++)

        array[i] = matrix_pair;

    /*
     * Do a quick error check to make sure we copied
     * the right number of elements.
     */
    if (i != num_matrix_pairs)
        uFatalError("sort_gen_list", "Dirichlet_basepoint");

    /*
     * Sort the array of pointers.
     */
    qsort( array,
           num_matrix_pairs,
           sizeof(MatrixPair *),
           compare_image_height);

    /*
     * Adjust the MatrixPairs' prev and next fields
     * to reflect the new ordering.
     */

    gen_list->begin.next = array[0];
    array[0]->prev = &gen_list->begin;
    array[0]->next = array[1];

    for (i = 1; i < num_matrix_pairs - 1; i++)
    {
        array[i]->prev = array[i-1];
    }
}

```

```
        array[i]->next = array[i+1];
    }

    array[num_matrix_pairs - 1]->prev = array[num_matrix_pairs - 2];
    array[num_matrix_pairs - 1]->next = &gen_list->end;
    gen_list->end.prev = array[num_matrix_pairs - 1];

    /*
     * Free the array.
     */
    my_free(array);
}

static int CDECL compare_image_height(
    const void *ptr1,
    const void *ptr2)
{
    double diff;

    diff = (*((MatrixPair **)ptr1))->height
        - (*((MatrixPair **)ptr2))->height;

    if (diff < 0.0)
        return -1;
    if (diff > 0.0)
        return +1;
    return 0;
}

static double length3(
    double v[3])
{
    double length;
    int i;

    length = 0.0;

    for (i = 0; i < 3; i++)
        length += v[i] * v[i];

    length = sqrt(length); /* no need for safe_sqrt() */

    return length;
}

static double inner3(
    double u[3],
    double v[3])
{
    double sum;
    int i;

    sum = 0.0;

    for (i = 0; i < 3; i++)
        sum += u[i] * v[i];

    return sum;
}

static void copy3(
    Solution dest,
    const Solution source)
{
    int i;

    for (i = 0; i < 3; i++)
        dest[i] = source[i];
}
```