Software Implementation of Image Point Correspondence Algorithms for Motion Parameter Determination

Sunil Fotedar
Dr. R. J. P. deFigueiredo
Eelectrical and Computer Eengineering Department
Rice University, Houston, TX.
April 29, 1987

Supported by Contract NAS9-17145 and NASA/JSC Grant NCC 9-16.

```
/*************************************************************************/
/*              IMAGE POINT CORRESPONDENCE ALGORITHM             */
/*                        (MAIN FILE)                    */
/*************************************************************************/
/*   Author: Sunil Fotedar (LESC)                        */
/*   Place:  Rice University, Houston                    */
/*         Electrical & Computer Engineering Department          */
/*************************************************************************/
/* This is the main program for estimating 3-D motion parameters     */
/* using the  "Image Point Correspondence" method.                */
/*                                                       */
/* Notations/Terminology Used:                           */
/*                                                       */
/* x[], y[], z[] : 3-D coordinates of object features before motion  */
/* xr[], yr[], zr[]: 3-D coordinates of object features after motion */
/* xi[], yi[], xri[], yri[]: Corresponding 2-D image coordinates     */
/* R, T : Rotation matrix and Translational Vector                */
/* points: Number of feature points considered                    */
/*                                                       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "sunil.h"

#include "message.c"
#include "constr.c"
#include "rotation.c"
#include "motion.c"
#include "matrix.c"
#include "vector.c"
#include "ipc_meth.c"
#include "ipc_out.c"

main()

{

      int     i,j,data;
      double  g[MAX_POINTS][1],
              TE[3],
              R1[3][3], R2[3][3];

      FILE  *fp;

printf("\nTHE DETERMINATION OF 3-D MOTION PARAMETERS\n");
printf("\t USING THE IPC ALGORITHM");

printf("\n\nSpecify the type of data ( S or D ), the IPC\n");
```

```
printf("method ( I, II or III ), and the representation\n");
printf("for rotation matrix ( A or B ):\n\n");

printf("Options available:\n\n");

printf("0 : S I A");          printf("\t4  : S III A\n");
printf("1 : S I B");          printf("\t5  : S III B\n");
printf("2 : S II A");         printf("\t6  : D I A\n");
printf("3 : S II B");         printf("\t7  : D I B\n\n");
printf("8 : Quit.\n");

printf("\nEnter option number :\n");

      scanf("%d", &data );

switch(data) {

#include "ipccases.c"

      }

}
/*************************************************************************/


/*************************************************************************/
/*                      IPC  CASES                                       */
/*-----------------------------------------------------------------------*/
/* Abstract :   All the possible cases for IPC algorithm have been   */
/*            considered.                                                */
/*-----------------------------------------------------------------------*/
/* Author : Sunil Fotedar       Place : Rice University (1988)       */
/*************************************************************************/

      case 0 :

      message_stringI();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.a,&motion.b,&motion.theta,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);

      fp = fopen ( "test.dat","r" );

      for (i = 0; i < object.points; i++) {
      fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
      }

            constraints_A(&motion);
```

```
            rotation_matrix_A(&motion);

            move_features(&object,&motion);

            generate_essential_elements(&object,g);

            ipc_methodI(g,TE,R1,R2);

            data_output_A(&object,TE,R1,R2);

            break;

        case 1 :

        message_stringII();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.roll,&motion.yaw,&motion.pitch,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);

        fp = fopen ( "test.dat","r" );

        for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
        }

            constraints_B(&motion);

            rotation_matrix_B(&motion);

            move_features(&object,&motion);

            generate_essential_elements(&object,g);

            ipc_methodI(g,TE,R1,R2);

            data_output_B(&object,TE,R1,R2);

            break;


        case 2 :

        message_stringI();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.a,&motion.b,&motion.theta,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);
```

```
        fp = fopen ( "test.dat","r" );

        for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
        }

                constraints_A(&motion);

                rotation_matrix_A(&motion);

                move_features(&object,&motion);

                generate_essential_elements(&object,g);

                ipc_methodII(g,TE,R1,R2);

                data_output_A(&object,TE,R1,R2);

                break;

        case 3 :

        message_stringII();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.roll,&motion.yaw,&motion.pitch,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);

        fp = fopen ( "test.dat","r" );

        for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
        }

                constraints_B(&motion);

                rotation_matrix_B(&motion);

                move_features(&object,&motion);

                generate_essential_elements(&object,g);

                ipc_methodII(g,TE,R1,R2);

                data_output_B(&object,TE,R1,R2);

                break;
```

```
        case 4 :

        message_stringI();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.a,&motion.b,&motion.theta,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);

        fp = fopen ( "test.dat","r" );

        for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
        }

                constraints_A(&motion);

                rotation_matrix_A(&motion);

                move_features(&object,&motion);

                generate_essential_elements(&object,g);

                ipc_methodIII(g,TE,R1,R2);

                data_output_A(&object,TE,R1,R2);

                break;

        case 5 :

        message_stringII();

scanf("%d %lf %lf %lf %lf %lf %lf",
&object.points,&motion.roll,&motion.yaw,&motion.pitch,&motion.T[0],&motion.T[1],&mo-
tion.T[2]);

        fp = fopen ( "test.dat","r" );

        for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf\n",&object.x[i],&object.y[i],&object.z[i]);
        }

                constraints_B(&motion);

                rotation_matrix_B(&motion);

                move_features(&object,&motion);

                generate_essential_elements(&object,g);
```

```
            ipc_methodIII(g,TE,R1,R2);

            data_output_B(&object,TE,R1,R2);

            break;

        case 6 :

        message_stringIII();

        scanf("%d", &object.points);

        fp = fopen ( "nasa.dat","r" );

for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf %lf %lf
%lf\n",&object.x[i],&object.y[i],&object.z[i],&object.xr[i],&object.yr[i],&object.zr[i]);
}
            generate_essential_elements(&object,g);

            ipc_methodI(g,TE,R1,R2);

            data_output_A(&object,TE,R1,R2);

            break;

        case 7 :

        message_stringIV();

        scanf("%d", &object.points);

        fp = fopen ( "nasa.dat","r" );

for (i = 0; i < object.points; i++) {
        fscanf (fp,"%lf %lf %lf %lf %lf
%lf\n",&object.x[i],&object.y[i],&object.z[i],&object.xr[i],&object.yr[i],&object.zr[i]);
        }

            generate_essential_elements(&object,g);

            ipc_methodI(g,TE,R1,R2);

            data_output_B(&object,TE,R1,R2);

            break;

        case 8 :
```

```
            break;

/*************************************************************************/


/*************************************************************************/
/*            BASIC ROUTINES FOR THE IPC ALGORITHM                       */
/*-----------------------------------------------------------------------*/
/* Abstract : These are the basic routines for the IPC algorithm     */
/*           and cover all the cases.                                    */
/*-----------------------------------------------------------------------*/
/* Author : Sunil Fotedar       Place : Rice University (1988)       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>

static  int     i,j;
static  double  pi, trace, norm,

XY[MAX_POINTS][MAX_POINTS],XYI[MAX_POINTS][MAX_POINTS],m[MAX_POINT
S][1],

XYt[MAX_POINTS][MAX_POINTS],SF[MAX_POINTS][MAX_POINTS],SFi[MAX_POINT
S][MAX_POINTS],
            q1[3],q2[3],q3[3],q4[3],q5[3],q6[3],M[3],
            w1[3],w2[3],w3[3],w4[3],w5[3],w6[3],
            W1[3],W2[3],W3[3],W4[3],W5[3],W6[3],
            sf[3],mod[3],r1[3],r2[3],r3[3],
            L1[3],L2[3],L3[3],
            rr1[3],rr2[3],rr3[3],
            dl1l2,dl1l3,dl2l1,dl2l3,dl3l1,dl3l2,
            cl2l3[3],cl2l3l1[3],cr1l2[3],cr1l3[3],
            cl3l1[3],cl3l1l2[3],cr2l3[3],cr2l1[3],
            cl1l2[3],cl1l2l3[3],cr3l1[3],cr3l2[3],
            crr1l2[3],crr1l3[3],
            e[3][3],et[3][3],qr[3][3],ql[3][3],
            u[3][3],v[3][3],w[3][3],
            REt1[3][3],REt2[3][3],
            R1[3][3],R2[3][3];

FILE    *fp_essential_matrix, *fopen();

generate_essential_elements(obj,g)

struct  object_definition *obj;

double  g[MAX_POINTS][1];
```

```
{

        pi=4.0*atan(1.0);

        generate_image_coordinates(&(*obj));

        generate_coordinate_matrix (&(*obj),XY);

        if ( obj->points > MAX_POINTS ) {

                printf("\nSorry, the program won't work as there are\n");
                printf("too many data points.\n");
                exit(0);
        }

        if ( obj->points < POINTS ) {

                printf("\nSorry, the program won't work as we need\n");
                printf("a minimum of eight data points.\n");
                exit(0);

        }
        else if ( obj->points == POINTS ) {

                matrix_inverse(XY,POINTS,XYI);

        }

        else {

                pseudo_inverse(XY,obj->points,POINTS,XYI);

        }

                for(i = 0; i < obj->points; i++) {
                        m[i][0]=(-1);
                }

                matrix_product(XYI,POINTS,obj->points,m,1,g);

}
/*************************************************************************/
/*                    FIRST METHOD OF IPC                                */
/*************************************************************************/

ipc_methodI(g,TE,R1,R2)

double  g[MAX_POINTS][1],R1[3][3],R2[3][3],
```

```
        TE[3];

{

    pi=4.0*atan(1.0);

    fp_essential_matrix = fopen ("ESSENT.DAT","w");

    for(i=0;i<POINTS;i++) {
        fprintf(fp_essential_matrix,"%lf\n",g[i][0]);
    }
        fclose(fp_essential_matrix);

        for(i = 0; i < 3; i++) {
            e[0][i]=g[i][0];
        }
        e[1][0]=g[3][0]; e[1][1]=g[4][0]; e[1][2]=g[5][0];
        e[2][0]=g[6][0]; e[2][1]=g[7][0]; e[2][2]=1.0;

        matrix_transpose (e,3,3,et);

        matrix_product(e,3,3,et,3,qr);

    /* Computation of translational vector (up to a scale factor) */

        TE[0] = sqrt(0.5*(-qr[0][0]+qr[1][1]+qr[2][2]));
        TE[1] = sqrt(0.5*(qr[0][0]-qr[1][1]+qr[2][2]));
        TE[2] = sqrt(0.5*(qr[0][0]+qr[1][1]-qr[2][2]));

    printf("The Estimated Translational Vector (up to a\n");
    printf("scale factor) is:\n");

    for(i=0;i<3;i++) {
        printf("%lf\t",TE[i]);
    }
    printf("\n");

            svd(e,3,3,u,w,v);

            generate_R1R2_matrix(u,v,R1,R2);


}
/*************************************************************************/
/*                  SECOND METHOD OF IPC                                 */
/*************************************************************************/

ipc_methodII(g,TE,R1,R2)
```

```
double  g[MAX_POINTS][1],
            TE[3],
            R1[3][3],R2[3][3];

{

        pi=4.0*atan(1.0);

        fp_essential_matrix = fopen ("ESSENT.DAT","w");

        for(i=0;i<POINTS;i++) {
                fprintf(fp_essential_matrix,"%lf\n",g[i][0]);
        }
                fclose(fp_essential_matrix);

                for(i = 0; i < 3; i++) {
                        e[0][i]=g[i][0];
                }
                e[1][0]=g[3][0]; e[1][1]=g[4][0]; e[1][2]=g[5][0];
                e[2][0]=g[6][0]; e[2][1]=g[7][0]; e[2][2]=1.0;

                for(i=0;i<3;i++) {
                        L1[i] = e[0][i];
                        L2[i] = e[1][i];
                        L3[i] = e[2][i];
                }

                matrix_transpose (e,3,3,et);

                matrix_product(e,3,3,et,3,qr);

        /* Computation of translational vector (upto a scale factor) */

        sf[0] = 0.5*(-qr[0][0]+qr[1][1]+qr[2][2]);
        sf[1] = 0.5*(qr[0][0]-qr[1][1]+qr[2][2]);
        sf[2] = 0.5*(qr[0][0]+qr[1][1]-qr[2][2]);

        modulus(sf,mod);

        dot_product(L1,L2,&dl1l2);
        dot_product(L1,L3,&dl1l3);
        dot_product(L2,L3,&dl2l3);
        dot_product(L2,L1,&dl2l1);
        dot_product(L3,L1,&dl3l1);
        dot_product(L3,L2,&dl3l2);

        cross_product(L2,L3,cl2l3);
        cross_product(cl2l3,L1,cl2l3l1);
        cross_product(L3,L1,cl3l1);
```

```
	cross_product(cl3l1,L2,cl3l1l2);
	cross_product(L1,L2,cl1l2);
	cross_product(cl1l2,L3,cl1l2l3);

	if((mod[0] >= mod[1])&&(mod[0] >= mod[2])) {

		TE[0] = sqrt(mod[0]);
		TE[1] = (-1)*dl1l2/sqrt(mod[0]);
		TE[2] = (-1)*dl1l3/sqrt(mod[0]);

	printf("The Estimated Translational Vector (up to a\n");
	printf("scale factor) is:\n");

	for(i=0;i<3;i++) {
		printf("%lf\t",TE[i]);
	}
	printf("\n");

		dot_product(TE,TE,&norm);

		for( i = 0; i < 3; i++ ) {
	r1[i] = (cl2l3l1[i]+sqrt(mod[0])*cl2l3[i])/(norm*sqrt(mod[0]));
rr1[i] =(-cl2l3l1[i]+sqrt(mod[0])*cl2l3[i])/(norm*sqrt(mod[0]));
	r2[i] = (L3[i]+sqrt(mod[1])*r1[i])/(sqrt(mod[0]));
	rr2[i] = (-L3[i]+sqrt(mod[1])*rr1[i])/(sqrt(mod[0]));
	r3[i] = (-L2[i]+sqrt(mod[2])*r1[i])/(sqrt(mod[0]));
	rr3[i] = (L2[i]+sqrt(mod[2])*rr1[i])/(sqrt(mod[0]));

		}

	} else if (mod[1] >= mod[2]) {

		TE[0] = (-1)*dl2l1/sqrt(mod[1]);
		TE[1] = sqrt(mod[1]);
		TE[2] = (-1)*dl2l3/sqrt(mod[1]);

	printf("The Estimated Translational Vector (up to a\n");
	printf("scale factor) is:\n");

	for(i=0;i<3;i++) {
		printf("%lf\t",TE[i]);
	}
	printf("\n");

		dot_product(TE,TE,&norm);

		for( i = 0; i < 3; i++ ) {

	r2[i] = (cl3l1l2[i]+sqrt(mod[1])*cl3l1[i])/(norm*sqrt(mod[1]));
```

```
        rr2[i] = (-cl3l1l2[i]+sqrt(mod[1])*cl3l1[i])/(norm*sqrt(mod[1]));
        r1[i] = (-L3[i]+sqrt(mod[0])*r2[i])/(sqrt(mod[1]));
        rr1[i] = (L3[i]+sqrt(mod[0])*rr2[i])/(sqrt(mod[1]));
        r3[i] = (L1[i]+sqrt(mod[2])*r2[i])/(sqrt(mod[1]));
        rr3[i] = (-L1[i]+sqrt(mod[2])*rr2[i])/(sqrt(mod[1]));

                }

        } else {

                TE[0] = (-1)*dl3l1/sqrt(mod[2]);
                TE[1] = (-1)*dl3l2/sqrt(mod[2]);
                TE[2] = sqrt(mod[2]);

        printf("The Estimated Translational Vector (up to a\n");
        printf("scale factor) is:\n");

        for(i=0;i<3;i++) {
                printf("%lf\t",TE[i]);
        }
        printf("\n");

                dot_product(TE,TE,&norm);

                for( i = 0; i < 3; i++ ) {

        r3[i] = (cl1l2l3[i]+sqrt(mod[2])*cl1l2[i])/(norm*sqrt(mod[2]));
rr3[i] = (-cl1l2l3[i]+sqrt(mod[2])*cl1l2[i])/(norm*sqrt(mod[2]));
        r1[i] = (L2[i]+sqrt(mod[0])*r3[i])/(sqrt(mod[2]));
        rr1[i] = (-L2[i]+sqrt(mod[0])*rr3[i])/(sqrt(mod[2]));
        r2[i] = (-L1[i]+sqrt(mod[1])*r3[i])/(sqrt(mod[2]));
        rr2[i] = (L1[i]+sqrt(mod[1])*rr3[i])/(sqrt(mod[2]));

                }

        }

for(i = 0; i < 3; i++ ) {

        R1[0][i] = r1[i]; R1[1][i] = r2[i]; R1[2][i] = r3[i];
        R2[0][i] = rr1[i]; R2[1][i] = rr2[i]; R2[2][i] = rr3[i];

}

        printf("The Estimated Rotation Matrix R1 is:\n");

        for ( i = 0; i < 3; i++ ) {
                for ( j = 0; j < 3; j++ ) {
                        printf("%lf\t",R1[j][i]);
```

```
            }
            printf("\n");
        }
        printf("\nThe Estimated Rotation Matrix R2 is:\n");

        for ( i = 0; i < 3; i++ ) {
            for ( j = 0; j < 3; j++ ) {
                printf("%lf\t",R2[j][i]);
            }
            printf("\n");
        }

}
/*************************************************************************/
/*                    THIRD METHOD OF IPC                                */
/*************************************************************************/

ipc_methodIII(g,TE,R1,R2)

double  g[MAX_POINTS][1],
        TE[3],
        R1[3][3],R2[3][3];

{

        pi=4.0*atan(1.0);

        fp_essential_matrix = fopen ("ESSENT.DAT","w");

        for(i=0;i<POINTS;i++) {
            fprintf(fp_essential_matrix,"%lf\n",g[i][0]);
        }
            fclose(fp_essential_matrix);

            for(i = 0; i < 3; i++) {
                e[0][i]=g[i][0];
            }
            e[1][0]=g[3][0]; e[1][1]=g[4][0]; e[1][2]=g[5][0];
            e[2][0]=g[6][0]; e[2][1]=g[7][0]; e[2][2]=1.0;

            matrix_transpose (e,3,3,et);

        /* Computation of translational vector (upto a scale factor) */

        matrix_product(e,3,3,et,3,qr);

        TE[0] = sqrt(0.5*(-qr[0][0]+qr[1][1]+qr[2][2]));
        TE[1] = sqrt(0.5*(qr[0][0]-qr[1][1]+qr[2][2]));
        TE[2] = sqrt(0.5*(qr[0][0]+qr[1][1]-qr[2][2]));
```

```
printf("The Estimated Translational Vector (up to a\n");
printf("scale factor) is:\n");

        for(i=0;i<3;i++) {
                printf("%lf\t",TE[i]);
        }
        printf("\n");

        trace = sqrt(TE[0]*TE[0]+TE[1]*TE[1]+TE[2]*TE[2]);

for( i = 0; i < 3; i++ ) {
        M[i] = TE[i]/trace;
}

for( i = 0; i < 3; i++ ) {
                q1[i] = et[0][i]/trace;
                q2[i] = et[1][i]/trace;
                q3[i] = et[2][i]/trace;
}
for( i = 0; i < 3; i++ ) {
                q4[i] = (-1)*q1[i];
                q5[i] = (-1)*q2[i];
                q6[i] = (-1)*q3[i];
}

cross_product(q1,M,w1);
cross_product(q2,M,w2);
cross_product(q3,M,w3);
        cross_product(q4,M,w4);
        cross_product(q5,M,w5);
        cross_product(q6,M,w6);
                cross_product(w1,w2,W1);
                cross_product(w2,w3,W2);
                cross_product(w3,w1,W3);
                        cross_product(w4,w5,W4);
                        cross_product(w5,w6,W5);
                        cross_product(w6,w4,W6);

        /* RE is the estimated Rotation Matrix */

for ( i = 0; i < 3; i++ ) {
        REt1[0][i] = w1[i] + W2[i];
        REt1[1][i] = w2[i] + W3[i];
        REt1[2][i] = w3[i] + W1[i];
}

matrix_transpose(REt1,3,3,R1);
```

```
        printf("The Estimated Rotation Matrix R1 is:\n");

        for ( i = 0; i < 3; i++ ) {
                for ( j = 0; j < 3; j++ ) {
                        printf("%lf\t",R1[j][i]);
                }
        printf("\n");
        }
        for ( i = 0; i < 3; i++ ) {
                REt2[0][i] = w4[i] + W5[i];
                REt2[1][i] = w5[i] + W6[i];
                REt2[2][i] = w6[i] + W4[i];
        }

        matrix_transpose(REt2,3,3,R2);

        printf("The Estimated Rotation Matrix R2 is:\n");

        for ( i = 0; i < 3; i++ ) {
                for ( j = 0; j < 3; j++ ) {
                        printf("%lf\t",R2[j][i]);
                }
        printf("\n");
        }

}
/*************************************************************************/


/*************************************************************************/
/*                      DATA  OUTPUT                                     */
/*-----------------------------------------------------------------*/
/* Author: Sunil Fotedar        Place: Rice University (1988)       */
/*-----------------------------------------------------------------*/
/* Abstract: The output of the IPC algorithm (rotation matrix,      */
/*          translational vector, 3-D motion parameters, etc.)     */
/*          is computed.                                           */
/*************************************************************************/

#include <stdio.h>
#include <math.h>

data_output_A(obj,TE,R1,R2)

struct object_definition *obj;

double  TE[3],
            R1[3][3], R2[3][3];
```

```
{
        double  a1,b1,c1,angle1,
                a2,b2,c2,angle2;

        printf("\nThe direction cosines of the axis and\n");
        printf("the angle of rotation about the axis,\n");
        printf("(corresponding to R1 and R2) are respectively:\n");

        rotation_parameter_A (R1,&a1,&b1,&c1,&angle1);

        printf("\n");

        rotation_parameter_A (R2,&a2,&b2,&c2,&angle2);

        printf("\n");

        estimate_shape(R1,R2,TE,&(*obj));

}
/*************************************************************************/

data_output_B(obj,TE,R1,R2)

struct object_definition *obj;

double  TE[3],
                R1[3][3], R2[3][3];

{
        double  roll1, yaw1, pitch1,
                roll2, yaw2, pitch2;

        printf("\nThe `Euler angles' ( Roll, Yaw, and Pitch ),\n");
        printf("corresponding to R1 and R2, are respectively:");
        printf("\n");

        rotation_parameter_B (R1,&roll1,&yaw1,&pitch1);

        printf("\n");

        rotation_parameter_B (R2,&roll2,&yaw2,&pitch2);

        printf("\n");

        estimate_shape(R1,R2,TE,&(*obj));

}

/*************************************************************************/
```

```
/*  RIGHT SOLUTION FOR THE MOTION PARAMETERS & SHAPE DETERMINATION   */
/*----------------------------------------------------------------*/
/* Abstract :   This routine chooses the right solution of thev       */
/*              rotation matrix for the GIPC Algorithm.               */
/*****************************************************************/

int same_sign();

static  int     i,j;
static  double  t1[MAX_POINTS][3],S[3][1],R1t[3][3],R1S[3][1],K1[3],
                t2[MAX_POINTS][3],R2t[3][3],R2S[3][1],K2[3],
                nr1[MAX_POINTS],dr1[MAX_POINTS],nr2[MAX_POINTS],dr2[MAX_POINTS],
                pi;


estimate_shape (R1,R2,TE,obj)

struct object_definition *obj;

double  R1[3][3],R2[3][3],
        TE[3];

{

        for(i = 0; i < obj->points; i++) {
                for(j = 0; j < 3; j++) {
                        t1[i][j] = R1[0][j] - obj->xri[i]*R1[2][j];
                        t2[i][j] = R2[0][j] - obj->xri[i]*R2[2][j];
                }
        }
                for(i = 0; i < 3; i++) {
                        S[i][0] = TE[i];
                }
                matrix_transpose(R1,3,3,R1t);
                matrix_product(R1t,3,3,S,1,R1S);
                matrix_transpose(R2,3,3,R2t);
                matrix_product(R2t,3,3,S,1,R2S);

                for(i = 0; i < 3; i++) {
                        K1[i] = -R1S[i][0];
                        K2[i] = -R2S[i][0];
                }


        for(i = 0; i < obj->points; i++) {

        nr1[i] = t1[i][0]*K1[0] + t1[i][1]*K1[1] + t1[i][2]*K1[2];
dr1[i] = t1[i][0]*obj->xi[i] + t1[i][1]*obj->yi[i] + t1[i][2];
        nr2[i] = t2[i][0]*K2[0] + t2[i][1]*K2[1] + t2[i][2]*K2[2];
```

```
dr2[i] = t2[i][0]*obj->xi[i] + t2[i][1]*obj->yi[i] + t2[i][2];
            obj->z1[i] = nr1[i]/dr1[i]; obj->z2[i] = nr2[i]/dr2[i];
            obj->x1[i] = obj->z1[i]*obj->xi[i]; obj->y11[i] = obj->z1[i]*obj->yi[i];
            obj->x2[i] = obj->z2[i]*obj->xi[i]; obj->y2[i] = obj->z2[i]*obj->yi[i];

obj->zr1[i] = obj->x1[i]*R1[2][0]+obj->y11[i]*R1[2][1]+obj->z1[i]*R1[2][2]+TE[2];
obj->zr2[i] = obj->x2[i]*R2[2][0]+obj->y2[i]*R2[2][1]+obj->z2[i]*R2[2][2]+TE[2];
            obj->xr1[i] = obj->xri[i]*obj->zr1[i]; obj->yr1[i] = obj->yri[i]*obj->zr1[i];
            obj->xr2[i] = obj->xri[i]*obj->zr2[i]; obj->yr2[i] = obj->yri[i]*obj->zr2[i];


      }
      make_decision(&(*obj));
}

/*************************************************************************/

make_decision(obj)

struct object_definition *obj;

{
      int     ret1, ret2, ret3, ret4, ret5, ret6;

      ret1 = same_sign(obj->z[0],obj->z1[0]); ret2 = same_sign(obj->zr[0],obj->zr1[0]);
      ret3 = same_sign(obj->z[0],obj->z2[0]); ret4 = same_sign(obj->zr[0],obj->zr2[0]);

      for(i = 1; i < obj->points; i++) {
            ret1 += same_sign(obj->z[i],obj->z1[i]);
            ret2 += same_sign(obj->zr[i],obj->zr1[i]);
            ret3 += same_sign(obj->z[i],obj->z2[i]);
            ret4 += same_sign(obj->zr[i],obj->zr2[i]);
      }

      ret5 = fabs(ret1-ret2);  ret6 = fabs(ret3-ret4);

 printf("Diagnostics: ret1 = %d, ret2 = %d, ret3 = %d\n", ret1, ret2, ret3);
 printf("             ret4 = %d, ret5 = %d, ret6 = %d\n\n", ret4, ret5, ret6);

/* Make a Proper Decision */

      if((ret1==ret3)&&(ret2==ret4)) {
            printf("Sorry, we cannot choose the right solution\n");
            printf("under such conditions.\n");
            exit(0);
      }
      else if(((ret5==0)&&(ret6==0))&&((ret1>ret3)||(ret2>ret4))) {
            printf("Conclusion: Choose R1 and its motion parameters\n");
            printf("as the final solution. The estimated `range'\n");
```

```
            printf("coordinates (up to a scale factor) are:\n");
            printf("\nz\t\t\tz'\n");
            for (i = 0; i < obj->points; i++) {
            printf("%lf\t\t%lf\n",obj->z1[i],obj->zr1[i]);
            }
      }
      else if(((ret1>=obj->points-1)&&(ret2>=obj->points-
1))||((ret1<=1)&&(ret2<=1))||(ret5<ret6)) {
            printf("Conclusion: Choose R1 and its motion parameters\n");
            printf("as the final solution. The estimated `range'\n");
            printf("coordinates (up to a scale factor) are:\n");
            printf("\nz\t\t\tz'\n");
            for (i = 0; i < obj->points; i++) {
            printf("%lf\t\t%lf\n",obj->z1[i],obj->zr1[i]);
            }
      } else {
            printf("Conclusion: Choose R2 and its motion parameters\n");
            printf("as the final solution. The estimated `range'\n");
            printf("coordinates (up to a scale factor) are:\n");
            printf("\nz\t\t\tz'\n");
            for (i = 0; i < obj->points; i++) {
            printf("%lf\t\t%lf\n",obj->z2[i],obj->zr2[i]);
            }
      }

}

/*************************************************************************/

int same_sign(z,zr)

double z,zr;

{
      if((z >= 0) && (zr >= 0)) {
            return 1;
      } else if((z < 0) && (zr < 0)) {
            return 1;
      } else
            return 0;

}
/*************************************************************************/


/*************************************************************************/
/*          OBJECT MOTION & CONVERSION TO IMAGE COORDINATES       */
/*-----------------------------------------------------------------*/
/* Abstract :   The surface points are moved by R, followed by T.   */
```

```
/*-----------------------------------------------------------------*/
/* Author : Sunil Fotedar       Place : Rice University (1988)     */
/*******************************************************************/

#include <stdio.h>
#include <math.h>

/*******************************************************************/
/*                  MOTION OF FEATURES                             */
/*******************************************************************/

move_features (obj,motion)

struct object_definition *obj;
struct motion_parameters *motion;

{
      int     i;

      for(i=0; i<(obj->points); i++) {
            obj->xr[i] = motion->R[0][0]*obj->x[i]+motion->R[0][1]*obj->y[i]+motion-
>R[0][2]*obj-
>z[i]+motion->T[0];
            obj->yr[i] = motion->R[1][0]*obj->x[i]+motion->R[1][1]*obj->y[i]+motion-
>R[1][2]*obj-
>z[i]+motion->T[1];
            obj->zr[i] = motion->R[2][0]*obj->x[i]+motion->R[2][1]*obj->y[i]+motion-
>R[2][2]*obj-
>z[i]+motion->T[2];
      }
}
/*******************************************************************/
/*                  2D IMAGE COORDINATES                           */
/*-----------------------------------------------------------------*/
/* Abstract :   This routine finds the 2D image coordinates of     */
/*            any number of surface points.                        */
/*******************************************************************/

generate_image_coordinates (obj)

struct object_definition *obj;

{
      int i;
      double Y,Z;

      for(i = 0; i < obj->points; i++) {
            if(obj->z[i]==0.) {
                  Y = 0.000001;   /* approximation */
```

```
                obj->xi[i] = (obj->x[i])/Y; obj->yi[i] = (obj->y[i])/Y;
            } else {
                obj->xi[i] = (obj->x[i])/(obj->z[i]); obj->yi[i] = (obj->y[i])/(obj->z[i]);
            }

            if (obj->zr[i]==0.) {
                Z = 0.000001;   /* approximation */
                obj->xri[i] = (obj->xr[i])/Z; obj->yri[i] = (obj->yr[i])/Z;
            } else {
                obj->xri[i] = (obj->xr[i])/(obj->zr[i]); obj->yri[i] = (obj->yr[i])/(obj->zr[i]);
            }
        }
}
/*****************************************************************/
/*            GENERATE COORDINATE MATRIX                         */
/*---------------------------------------------------------------*/
/* Abstract :   This routine generates the coordinate matrix     */
/*            needed for the estimation of the E matrix.         */
/*****************************************************************/

generate_coordinate_matrix (obj,t)

struct object_definition *obj;

double  t[][POINTS];

{
        int     k;

        for (k=0; k< obj->points; k++) {
                t[k][0] = (obj->xri[k])*(obj->xi[k]);
                t[k][1] = (obj->xri[k])*(obj->yi[k]);
                t[k][2] = obj->xri[k];
                t[k][3] = (obj->yri[k])*(obj->xi[k]);
                t[k][4] = (obj->yri[k])*(obj->yi[k]);
                t[k][5] = obj->yri[k];
                t[k][6] = obj->xi[k];
                t[k][7] = obj->yi[k];
        }

}
/*****************************************************************/


/*****************************************************************/
/*            ROTATION MATRIX AND ITS PARAMETERS                 */
/*---------------------------------------------------------------*/
/* Author: Sunil Fotedar        Place: Rice University (1988)    */
/*---------------------------------------------------------------*/
```

```
/* Abstract: Computations of first and second representations of     */
/*          rotation matrix from its parameters and vice-versa      */
/*          are done in these routines (see Appendix A of the       */
/*          report).                                                */
/*******************************************************************/

#include <stdio.h>
#include <math.h>

/*******************************************************************/
/*            ROTATION MATRIX (FIRST REPRESENTATION)               */
/*-----------------------------------------------------------------*/
/* Abstract  : This routine forms a rotation matrix R of first kind  */
/*          when the directional cosines of any arbitrary axis     */
/*          and the angle of rotation around the axis are given.  */
/*          a, b:  the directional cosines, where a <= 1 & b <= 1 */
/*          theta: the angle of rotation                           */
/*******************************************************************/

rotation_matrix_A(motion)

struct motion_parameters *motion;

{
      double  th, pi;

      pi = 4.0*atan(1.0);

      motion->c = sqrt(1 - (motion->a)*(motion->a) - (motion->b)*(motion->b));

      th = (pi/180.0)*(motion->theta);

            motion->R[0][0] = (motion->a)*(motion->a)+(1-(motion->a)*(motion->a))*cos(th);
            motion->R[0][1] = (motion->a)*(motion->b)*(1-cos(th))-(motion->c)*sin(th);
            motion->R[0][2] = (motion->a)*(motion->c)*(1-cos(th))+(motion->b)*sin(th);
            motion->R[1][0] = (motion->a)*(motion->b)*(1-cos(th))+(motion->c)*sin(th);
            motion->R[1][1] = (motion->b)*(motion->b)+(1-(motion->b)*(motion->b))*cos(th);
            motion->R[1][2] = (motion->b)*(motion->c)*(1-cos(th))-(motion->a)*sin(th);
            motion->R[2][0] = (motion->a)*(motion->c)*(1-cos(th))-(motion->b)*sin(th);
            motion->R[2][1] = (motion->b)*(motion->c)*(1-cos(th))+(motion->a)*sin(th);
            motion->R[2][2] = (motion->c)*(motion->c)+(1-(motion->c)*(motion->c))*cos(th);

}
/*******************************************************************/
/*            ROTATION MATRIX (SECOND REPRESENTATION)              */
/*-----------------------------------------------------------------*/
/* Abstract  : This routine forms a rotation matrix R of second kind */
/*          when the angles of rotation                            */
/*          around x-, y- and z- axes ( euler angles )are given.  */
```

```
/*            theta, phi, psi are the `Euler angles'                */
/*********************************************************************/

rotation_matrix_B(motion)

struct motion_parameters *motion;

{
      double  th,ph,ps,
            pi;

      pi = 4.0*atan(1.0);

      th = (pi/180.0)*(motion->roll);
      ph = (pi/180.0)*(motion->yaw);
      ps = (pi/180.0)*(motion->pitch);

      motion->R[0][0] = cos(ph)*cos(ps) - sin(ps)*sin(ph)*sin(th);
      motion->R[0][1] = cos(ph)*sin(ps) + cos(ps)*sin(ph)*sin(th);
      motion->R[0][2] = -cos(th)*sin(ph);
      motion->R[1][0] = -sin(ps)*cos(th);
      motion->R[1][1] = cos(ps)*cos(th);
      motion->R[1][2] = sin(th);
      motion->R[2][0] = cos(ps)*sin(ph) + sin(ps)*sin(th)*cos(ph);
      motion->R[2][1] = sin(ps)*sin(ph) - cos(ps)*sin(th)*cos(ph);
      motion->R[2][2] = cos(ph)*cos(th);

}
/*********************************************************************/
/*                  ROTATION PARAMETERS(A)                           */
/*-------------------------------------------------------------*/
/* Abstract :   This function returns the rotation parameters     */
/*            when the first representation of R is used.       */
/*********************************************************************/

extern double power();

rotation_parameter_A (R,a,b,c,roll)

  double  R[3][3],
         *a,*b,*c,
         *roll;
{
      double  f,d,pi,A,B,C,
            roll1,sinr,cosr;


      f = power (R[0][2]-R[2][0],2) + power (R[1][0]-R[0][1],2);
```

```
d = power (R[2][1]-R[1][2],2) + f;
/*****************************************************************/
/*    Computation of Direction Cosines a,b,c of the axis      */
/*****************************************************************/

*a = (R[2][1]-R[1][2])/sqrt(d);
*b = (R[0][2]-R[2][0])/sqrt(d);
*c = (R[1][0]-R[0][1])/sqrt(d);

A = (-1)*(*a); B = (-1)*(*b); C = (-1)*(*c);
if ( (*a>-0.000001))
       if ( *b>(-0.000001))
              if ( *c>(-0.000001))

                     printf("%lf\t%lf\t%lf\n",*a,*b,*c);
               else
                     printf("%lf\t%lf\t%lf\n",*a,*b,C);
        else
              printf("%lf\t%lf\t%lf\n",*a,B,C);
 else
       printf("%lf\t%lf\t%lf\n",A,B,C);


/*****************************************************************/
/*             Computation of the Angle of Rotation          */
/*****************************************************************/

sinr = sqrt(d)*0.5;

if (((*a)==1)||((*a)==(-1))) {
cosr = R[1][1];
}
else {
cosr = (R[0][0]-power(*a,2))/(1-power(*a,2));
}

 quadset (&sinr,&cosr,&(*roll));


if((*a<(-0.000001))) {
              roll1 = 360 - (*roll);
              printf("%lf\n",roll1);
} else {
              if((*b<(-0.000001))) {
                     roll1 = 360 - (*roll);
              printf("%lf\n",roll1);
              } else {

                     if((*c<(-0.000001))) {
```

```
                    roll1 = 360 - (*roll);
                printf("%lf\n",roll1);
                    } else {
                printf("%lf\n",*roll);
                }
            }
        }
}
/*************************************************************************/
/*                  ROTATION PARAMETERS(B)                               */
/*-----------------------------------------------------------------------*/
/* Abstract :   This function returns the rotation parameters            */
/*            when the second representation of R is used.               */
/*************************************************************************/

rotation_parameter_B (R,roll,yaw,pitch)

  double  R[3][3],
         *roll,*yaw,*pitch;

{
      double  sin_roll, cos_roll,
              sin_pitch, cos_pitch,
              sin_yaw, cos_yaw,
              pi;

      sin_roll = R[1][2];
      cos_roll = sqrt(R[1][0]*R[1][0]+R[1][1]*R[1][1]);

      quadset (&sin_roll,&cos_roll,&(*roll));

      sin_pitch = (-1)*(R[1][0])/(sqrt(1-R[1][2]*R[1][2]));
      cos_pitch = (R[1][1])/(sqrt(1-R[1][2]*R[1][2]));

      quadset (&sin_pitch,&cos_pitch,&(*pitch));

      sin_yaw = (-1)*(R[0][2])/cos(*roll);
      cos_yaw = R[2][2]/cos(*roll);

      quadset (&sin_yaw,&cos_yaw,&(*yaw));

      if(*yaw < 0)
            *yaw += 360;

      printf("%lf\t%lf\t%lf\n", *roll, *yaw, *pitch );

}
```

```
/*****************************************************************/
/*                  GENERATE R1&R2 MATRICES                      */
/*---------------------------------------------------------------*/
/* Abstract :   This routine returns the matrices R1 and R2      */
/*              using the singular matrices u and v.             */
/*****************************************************************/

extern double matrix_inverse();

generate_R1R2_matrix (u,v,R1,R2)

double  u[3][3];
double  v[3][3];
double  R1[3][3];
double  R2[3][3];
{
      int i, j;
      double  s, det_u, det_v;
      double  vt[3][3];
      double  w[3][3],w1[3][3],ui[3][3],vi[3][3];
      double  wvt[3][3],wvt1[3][3];
      double  wk[3][3];

      det_u = matrix_inverse(u,3,ui);
      det_v = matrix_inverse(v,3,vi);

      s = det_u/det_v;

      w[0][0] = 0.; w[0][1] = (-1.); w[0][2] = 0.;
      w[1][0] = 1.; w[1][1] = 0. ;   w[1][2] = 0.;
      w[2][0] = 0.; w[2][1] = 0. ;   w[2][2] = s;

      matrix_transpose (v,3,3,vt);
      matrix_product (w,3,3,vt,3,wvt);
      matrix_product (u,3,3,wvt,3,R1);

      printf("The Estimated Rotation Matrix R1 is:\n");

      for (i=0;i<3;i++) {
            for (j=0;j<3;j++) {
                  printf("%lf\t",R1[i][j]);
            }
      printf("\n");
      }

      w1[0][0] = 0.;    w1[0][1] = 1. ;   w1[0][2] = 0.;
      w1[1][0] = (-1.); w1[1][1] = 0. ;   w1[1][2] = 0.;
      w1[2][0] = 0.;    w1[2][1] = 0. ;   w1[2][2] = s;
```

```
	matrix_transpose (v,3,3,vt);
	matrix_product (w1,3,3,vt,3,wvt1);
	matrix_product (u,3,3,wvt1,3,R2);

	printf("The Estimated Rotation Matrix R2 is:\n");

	for (i=0;i<3;i++) {
		for (j=0;j<3;j++) {
			printf("%lf\t",R2[i][j]);
		}
	printf("\n");
	}

}
/*************************************************************************/
/*                        QUADSET                                        */
/*-----------------------------------------------------------------------*/
/*      This program takes the values of sine and cosine for an angle */
/*      and returns the proper offset in degrees to place the angle   */
/*      in the proper quadrant.                                        */
/*************************************************************************/

quadset (sine, cosine, angle)

double  *sine, *cosine, *angle;


{
	double  ryp,
		pi,
		CF;     /* CF converts radians into degrees */

	ryp = atan((*sine / *cosine));

	pi = 4.0*atan(1.0);

	CF = (180.0/pi);

	if ((*sine)>0) {
		if((*cosine)>0) {       /*  1st quadrant  */
			*angle = ryp*CF;
		} else {                /*  2nd quadrant  */
			*angle = 180 + ryp*CF;
		}
	}
	else {
		if((*cosine)<0) {       /*  3rd quadrant  */
			*angle = ryp*CF - 180;
		} else {                /*  4th quadrant  */
```

```
                *angle = ryp*CF;
            }
        }
}


/*************************************************************************/
/*     ROTATION MATRIX AS A RESULT OF THE MOTION OF AN OBJECT AND    */
/*                  FRAME TRANSFORMATIONS                            */
/*-----------------------------------------------------------------*/
/*  Abstract :  This routine determines the resultant rotation matrix */
/*            of a moving object when the camera itself is moving.  */
/*            R     Rotation matrix due to the motion of the object */
/*            R12   Rotation matrix between frames F1 and F2        */
/*            R1    Rotation matrix between frame F1 and the        */
/*                  standard frame S                                */
/*            RE    The resultant rotation matrix                   */
/*************************************************************************/

resultant_rotation_matrix(R12,R1,R,R_res)

double  R12[3][3],
        R1[3][3],
        R[3][3],
        R_res[3][3];

{

        double  R12t[3][3],R1t[3][3],
                RR1[3][3],RR2[3][3];

                matrix_transpose(R12,3,3,R12t);
                matrix_transpose(R1,3,3,R1t);

                matrix_product(R,3,3,R1t,3,RR1);
                matrix_product(R12t,3,3,R1,3,RR2);

                matrix_product(RR2,3,3,RR1,3,R_res);

}
/*************************************************************************/
/*                  DETERMINATION OF A ROTATION MATRIX                   */
/*-----------------------------------------------------------------*/
/*  Abstract :  This routine estimates the  rotation matrix of a      */
/*            moving object when the camera itself is moving.       */
/*            R     Rotation matrix found using IPC algorithm       */
/*            R12   Rotation matrix between frames F1 and F2        */
/*            R1    Rotation matrix between frame F1 and the        */
/*                  standard frame S                                */
```

```
/*            RE    The estimated rotation matrix for the motion    */
/*                  of the object                                   */
/*******************************************************************/

determine_rotation_matrix( R, R12, R1, RE )

double  R[3][3], R12[3][3],
       R1[3][3], RE[3][3];

{
       double  R1t[3][3],
              RR1[3][3],RR2[3][3];

              matrix_transpose(R1,3,3,R1t);

              matrix_product(R1t,3,3,R12,3,RR1);
              matrix_product(R,3,3,R1,3,RR2);
              matrix_product(RR1,3,3,RR2,3,RE);

}
/*******************************************************************/


/*******************************************************************/
/*                  VECTOR PRODUCTS AND POWER                      */
/*-----------------------------------------------------------------*/
/* Author: Sunil Fotedar        Place: Rice University (1988)      */
/*-----------------------------------------------------------------*/
/* Abstract: Vector Products and Power function are presented.     */
/*******************************************************************/

#include <math.h>
#include <stdio.h>

/*******************************************************************/
/*                        CROSS PRODUCT                            */
/*******************************************************************/

cross_product(x,y,z)

double x[3],y[3],z[3];

{

       z[0] = (x[1]*y[2])-(x[2]*y[1]);
       z[1] = (x[2]*y[0])-(x[0]*y[2]);
       z[2] = (x[0]*y[1])-(x[1]*y[0]);

}
```

```
/*************************************************************************/
/*                        DOT PRODUCT                                    */
/*************************************************************************/

 dot_product(x,y,z)

double x[3],y[3],
       *z;

{
       int     i;

       *z = x[0]*y[0]+x[1]*y[1]+x[2]*y[2];

}

/*************************************************************************/
/*                        POWER                                          */
/*************************************************************************/

double  power (x,n)

double  x;
int     n;
{
       double  p;

       if ( n<0 ) {
              n = -n;

              for (p=1.; n>0; --n)
                     p = p * x;
              return (1./p);
       } else {
              for(p = 1.; n>0; --n)
                     p = p * x;
              return (p);
       }
}
/*************************************************************************/


/*************************************************************************/
/*                  LINEAR ALGEBRA ROUTINES                              */
/*-----------------------------------------------------------------------*/
/* Authors : W. D. Myrick (LINCOM) and Sunil Fotedar (LESC)              */
/*-----------------------------------------------------------------------*/
/* Abstract: Some linear algebra routines are presented.                 */
```

```
/*****************************************************************/

#include <stdio.h>
#include <math.h>

/*****************************************************************/
/*                  MATRIX PRODUCT                               */
/*---------------------------------------------------------------*/
/* Abstract :   This function finds the product of               */
/*            two matrices.                                      */
/*            X  first matrix of dimension p*q                   */
/*            Y  second matrix of dimension q*r                  */
/*            XY product matrix of dimension p*r                 */
/*****************************************************************/

static  double temp[MAX_POINTS*MAX_POINTS];

matrix_product (X,p,q,Y,r,XY)

int     p,q,r;
double
X[MAX_POINTS*MAX_POINTS],Y[MAX_POINTS*MAX_POINTS],XY[MAX_POINTS*
MAX_POINTS];

{
      int     i,j,k,t;
      double  sum;

      matrix_transpose (Y,q,r,temp);

      t = 0;
      for (i=0; i<p; i++) {
            for (j=0; j<r; j++) {
                  sum = 0.;
                  for (k=0; k<q; k++)
                        sum = sum + X[k+i*q]*temp[k+j*q];
                  XY[t] = sum;
                  t = t + 1;
            }
      }
}
/*****************************************************************/
/*                  MATRIX TRANSPOSE                             */
/*---------------------------------------------------------------*/
/* Abstract :   This function finds the transpose of             */
/*            a given matrix                                     */
/*            X  matrix of dimension p*q                         */
/*            XT transposed matrix of dimension q*p              */
/*****************************************************************/
```

```
matrix_transpose (X,p,q,XT)

int     p,q;
double  X[MAX_POINTS*MAX_POINTS],XT[MAX_POINTS*MAX_POINTS];

{
      int     i,j,k,t;

      j = 0;
      for (k=0; k<q; k++) {
            i = k;
            for (t=0; t<p; t++) {
                  XT[j] = X[i];
                  j = j + 1;
                  i = k + (t+1)*q;
            }
      }
}
/*****************************************************************/
/*                        MODULUS                                */
/*---------------------------------------------------------------*/
/* Abstract :   This routine returns the modulus of an array     */
/*            n   a 3D array                                     */
/*            mod the modulus of n                               */
/*****************************************************************/


modulus(n,mod)

double  n[],mod[];

{
      int i;

      for(i = 0; i < 3; i++ ) {
            if(n[i]>=0) {
                  mod[i] = n[i];
            } else {
                  mod[i] = (-1)*n[i];
            }
      }
}

/*****************************************************************/
/*                  INVERSE OF A MATRIX                          */
/*****************************************************************/
/* Abstract: This routine finds the inverse of a non-singular    */
/* square matrix.                                                */
```

```
/*****************************************************************/


double matrix_inverse( mat, dim, inv )

double  mat[] ;        /* IN:  SQUARE MATRIX */
double  inv[] ;        /* OUT: INVERSE OF mat */
int     dim ;          /* IN:  SQUARE DIMENSION OF mat */

{

int    col_pivot[MAX_POINTS*MAX_POINTS] ;          /* COLUMN LOCATION OF LARG-
EST PIVOT
                                  FOR EACH DIAGONAL ELEMENT */
int    row_pivot[MAX_POINTS*MAX_POINTS] ;          /* ROW LOCATION OF LARGEST
PIVOT
                                  FOR EACH DIAGONAL ELEMENT */
int     i , j , k , ii , jj , kk ;     /* LOOP COUNTERS */
int     col_start ;                    /* START INDEX FOR EACH COLUMN */
int     diag ;                         /* INDEX FOR EACH DIAGONAL ELEMENT */
double  largest ;                      /* LARGEST ELEMENT */
double  temp ;                         /* TEMPORARY WORKSPACE */
double  det ;                          /* DETERMINATE */

if( dim > MAX_DIM ) {
      fprintf( stderr , "\
ERROR: DIMENSION OF INPUT MATRIX TO `matrix_inverse' is greater than %d.\n\ Exit-
ing...\n" ,
MAX_DIM ) ;
      exit( 0 ) ;
}

      /*INITIALIZE INVERSE MATRIX */
for( i = 0 ; i <= dim*dim ; i++ )       inv[i] = mat[i] ;

det = 1.0 ;

      /* SEARCH FOR LARGEST ELEMENT  */
for( col_start = 0 , k = 1 ; k <= dim ; k++ , col_start += dim ) {
      col_pivot[k-1] = k ;
      row_pivot[k-1] = k ;
      diag = col_start + k ;
      largest = inv[diag-1] ;

      for( j = k ; j <= dim ; j++ ) {
            for( i = k ; i <= dim ; i++ ) {
                  ii = dim * ( j - 1 ) + i ;

                  if( fabs( largest ) < fabs( inv[ii-1] ) ) {
```

```
                    largest = inv[ii-1] ;
                    col_pivot[k-1] = i ;
                    row_pivot[k-1] = j ;
                }
        }
}
if( largest == 0.0 )    return( 0.0 ) ;

        /* INTERCHANGE ROWS  */

j = col_pivot[k-1] ;

if( j > k ) {
        for( i = k ; i <= dim*dim ; i += dim ) {
                temp = -inv[i-1] ;
                ii = i - k + j ;
                inv[i-1] = inv[ii-1] ;
                inv[ii-1] = temp ;
        }
}

        /* INTERCHANGE COLUMNS  */

i = row_pivot[k-1] ;

if( i > k ) {
        kk = dim * (i - 1) ;

        for( j = 1 ; j <= dim ; j++ ) {
                jj = col_start + j ;
                ii = kk + j ;
                temp =  - inv[jj-1] ;
                inv[jj-1] = inv[ii-1] ;
                inv[ii-1] = temp ;
        }
}

        /* DIVIDE COLUMN BY MINUS PIVOT (VALUE OF PIVOT ELEMENT IS
                CONTAINED IN BIGA)  */

for( i = 1 ; i <= dim ; i++ ) {
        if( i != k ) inv[col_start+i-1] = inv[col_start+i-1] / -largest;
}

        /* REDUCE MATRIX  */

for( i = 1 ; i <= dim ; i++ ) {
        temp = inv[col_start+i-1] ;
```

```
            for( ii = i , j = 1 ; j <= dim ; j++ , ii += dim ) {
                  if( i != k  &&  j != k ) {
                        kk = ii - i + k ;
                        inv[ii-1] = temp * inv[kk-1] + inv[ii-1] ;
                  }
            }
      }

            /* DIVIDE ROW BY PIVOT  */

      for( kk = k , j = 1 ; j <= dim ; j++ , kk += dim ) {
            if( j != k )    inv[kk-1] = inv[kk-1] / largest ;
      }

            /* PRODUCT OF PIVOTS  */

      det *= largest ;

            /* REPLACE PIVOT BY RECIPROCAL  */

      inv[diag-1] = 1.0 / largest ;
}

      /* FINAL ROW AND COLUMN INTERCHANGE  */

for( k = dim - 1 ; k > 0 ; k-- ) {
      i = col_pivot[k-1] ;

      if( i > k ) {
            jj = dim * (k - 1) ;
            ii = dim * (i - 1) ;

            for( j = 1 ; j <= dim ; j++ ) {
                  temp = inv[jj+j-1] ;
                  inv[jj+j-1] = -inv[ii+j-1] ;
                  inv[ii+j-1] = temp ;
            }
      }
      j = row_pivot[k-1] ;

      if( j > k ) {
            for( i = k ; i <= dim*dim ; i += dim ) {
                  temp = inv[i-1] ;
                  ii = i - k + j ;
                  inv[i-1] =  - inv[ii-1] ;
                  inv[ii-1] = temp ;
            }
      }
}
```

```
return( det );
}
/*************************************************************************/

pseudo_inverse(XY,row,column,XYI)

double  XY[MAX_POINTS][MAX_POINTS], XYI[MAX_POINTS][MAX_POINTS];
int     row, column;

{

double  XYt[MAX_POINTS][MAX_POINTS], SF[MAX_POINTS][MAX_POINTS],
SFi[MAX_POINTS][MAX_POINTS];

            matrix_transpose(XY,row,column,XYt);
            matrix_product(XYt,column,row,XY,column,SF);
            matrix_inverse(SF,column,SFi);
            matrix_product(SFi,column,column,XYt,row,XYI);

}


/*************************************************************************/
/*            SINGULAR VALUE DECOMPOSITION OF A MATRIX           */
/*************************************************************************/
/* Abstract : This routine returns the SVD of a matrix.           */
/*************************************************************************/

svd( x, m, n, u, s, v )

double x[][3] ;        /* INPUT: x matrix */
int m ;               /* INPUT: row dimension -- 3 in this case       */
int n ;               /* INPUT: column dimension -- 3 in this case    */
double u[][3] ;        /* OUTPUT: matrix of left singular values       */
double s[][3] ;        /* OUTPUT: diagonal matrix of eigen values      */
double v[][3] ;        /* OUTPUT: matrix of right singular values      */

{

int i, j ;            /* loop counters */

double xtemp[DIM] ;     /* x matrix -- the 1-d version          */
double utemp[DIM] ;     /* u matrix -- the 1-d version          */
double stemp[DIM] ;     /* s matrix -- the 1-d version          */
double vtemp[DIM] ;     /* v matrix -- the 1-d version          */
double etemp[DIM] ;     /* e vector -- calculated by svd        */
double work[DIM] ;      /* scratch array used by svd            */
int job ;             /* variable specifying job -- refer to svd.c    */
```

```
int info ;              /* error variable                    */

for( i = 0 ; i < m ; i++ ) {
        for( j = 0 ; j < n ; j++ ) {
                xtemp[i + j*m] = x[i][j] ;
        }
}

dsvdc_( xtemp, 3, 3, 3, stemp, etemp, utemp, 3, vtemp, 3, work, 11, info ) ;

for( i = 0 ; i < m ; i++ ) {
        for( j = 0 ; j < n ; j++ ) {
                u[i][j] = utemp[i + j*m] ;
                s[i][j] = stemp[i + j*m] ;
                v[i][j] = vtemp[i + j*m] ;
        }
}

return ;

}
/*-----------------------------------------------------------------------*/
/*  FUNCTION:   */

/*  The eigenvalues are stored in the column vector s.  Before being
    returned, 's' is transformed into an mm by mm matrix with the
    eigenvalues stored on the diagonal.                          */


extern double sign() ;  /* sign( a, b ) : Returns value of a with sign
                           of b.                                 */
extern double snrm2() ;
extern double sdot() ;


dsvdc_( x, ldx, n, p, s, e, u, ldu, v, ldv, work, job, info )

int ldx ;               /* INPUT:  Leading dimension of array x      */
int n ;                 /* INPUT:  Row dimension of array x          */
int p ;                 /* INPUT:  Column dimension of array x       */
int ldu ;               /* INPUT:  Leading dimension of array u      */
int ldv ;               /* INPUT:  Leading dimension of array v      */
int job ;               /* INPUT:  Controls computation of the singular
                           vectors.                                  */
int info ;              /* OUTPUT: Return 0 if singular values 'S' are
                           correct.                                  */

double x[] ;            /* INPUT:  2-d. x contains the matrix whose
                           singular value decomposition is to
```

```
                        be computed.  x is destroyed by
                        ssvdc().                    */
double s[] ;        /* OUTPUT: The first [minimum of n,p] values of
                        s contain the singular values of x
                        arranged in descending order.
                        s will be transformed into a 2-D array
                        with the singular values stored on the
                        diagonal.                   */
double e[] ;        /* OUTPUT: elements on the superdiagonal of the
                        bidiagonal matrix b -- usually zeros */
double u[] ;        /* OUTPUT: 2-d.  Contains matrix of left singular
                        vectors.                    */
double v[] ;        /* OUTPUT: 2-d.  Contains matrix of right singular
                        vectors.                    */
double work[] ;     /* Scratch array               */

/* ENTRY POINT */

{

int i, iter, j, jobu, k, kase, kk, l, ll, lls, lm1, lp1, ls, lu, m ;
int mm, mm1, mp1, nct, nctp1, ncu, nrt, nrtp1 ;
double b, c, cs, el, emm1, f, g, scale, shift, sl, sm, sn, smm1 ;
double t, t1, test, ztest, temp1 ;
double s_temp[25] ;     /* Temporary storage for s vector      */
int wantu = 0 ;         /* LOGICAL:  0 = false, 1 = true       */
int wantv = 0 ;         /* LOGICAL:  0 = false, 1 = true       */


jobu = ( job % 100 ) / 10 ;
ncu = n ;
if( jobu > 1 ) {
    ncu = ( n >= p ) ? p : n ;     /* minimum of n,p           */
}
if( jobu != 0 ) wantu = 1 ;
if( ( job % 10 ) != 0 ) wantv = 1 ;

/* Reduce x to bidiagonal form, storing the diagonal elements in s and
   the superdiagonal elements in e.                         */

info = 0 ;
nct = ( n-1 < p ) ? ( n-1 ) : p ;
temp1 = ( p-2 < n ) ? (p-2) : n ;
nrt = ( 0 > temp1 ) ? 0 : temp1 ;
lu = ( nct > nrt ) ? nct : nrt ;

if( lu >= 1 ) {

for( l = 0 ; l < lu ; l++ ) {
```

```
lp1 = l + 1 ;
if( l < nct ) {     /* Compute the transformation for the (l+1)-th
                       column and place the diagonal in s[l]     */
    s[l] = snrm2( n-l, &(x[l+l*n]), 1 ) ;
    if( s[l] != 0.0 ) {
        if( x[l+l*n] != 0.0 ) s[l] = sign( s[l], x[l+l*n] ) ;
        sscal( n-l, 1.0/s[l], &(x[l+l*n]), 1 ) ;
        x[l+l*n] = 1.0 + x[l+l*n] ;
    }
    s[l] = -s[l] ;
}

if( p > lp1 ) {
for( j = lp1 ; j < p ; j++ ) {
    if( l < nct ) {
        if( s[l] != 0.0 ) {     /* Apply transformation          */
            t = -sdot( n-l, &(x[l+l*n]), 1, &(x[l+j*n]), 1 ) /
                    x[l+l*n] ;
            saxpy( n-l, t, &(x[l+l*n]), 1, &(x[l+j*n]), 1 ) ;
        }
    }
    /* Place the (l+1)-th row of x into e for the subsequent
       calculation of the row transformation                    */
    e[j] = x[l+j*n] ;
}
}
if( wantu && l < nct ) {
    /* Place the transformation in u for subsequent back
       multiplication                                           */
    for( i = l ; i < n ; i++ ) {
        u[i+l*ldu] = x[i+l*n] ;
    }
}
if( l < nrt ) {
    /* Compute the (l+1)-th row transformation and place the
       (l+1)-th super-diagonal in e[l]                          */
    e[l] = snrm2( p-l-1, &(e[lp1]), 1 ) ;
    if( e[l] != 0.0 ) {
        if( e[lp1] != 0.0 ) e[l] = sign( e[l], e[lp1] ) ;
        sscal( p-l-1, 1.0/e[l], &(e[lp1]), 1 ) ;
        e[lp1] = 1.0 + e[lp1] ;
    }
    e[l] = -e[l] ;
    if( lp1 < n && e[l] != 0.0 ) {
        /* Apply the transformation                             */
        for( i = lp1 ; i < n ; i++ )  work[i] = 0.0 ;
        for( j = lp1 ; j < p ; j++ )
            saxpy( n-l-1, e[j], &(x[lp1+j*n]), 1, &(work[lp1]), 1) ;
        for( j = lp1 ; j < p ; j++ )
```

```
                saxpy( n-l-1, -e[j]/e[lp1], &(work[lp1]), 1,
                      &(x[lp1+j*n]), 1) ;
            }
            if( wantv ) {
                /*  Place the transformation in v for subsequent back
                   multiplication.                              */
                for( i = lp1 ; i < p ; i++ ) v[i+l*ldv] = e[i] ;
            }
        }
    }
    }
/* set up the final bidiagonal matrix of order m                */

m = ( p < n+1 )? p : n+1 ;
nctp1 = nct + 1 ;
nrtp1 = nrt + 1 ;
if( nct < p ) s[nct] = x[nct+nct*n] ;
if( n < m ) s[m-1] = 0.0 ;
if( nrtp1 < m ) e[nrt] = x[nrt+(m-1)*n] ;
e[m-1] = 0.0 ;

/* if required, generate u                                      */

if( wantu ) {

if( ncu >= nctp1 ) {
    for( j = nct ; j < ncu ; j++ ) {
        for( i = 0 ; i < n ; i++ ) u[i+j*ldu] = 0.0 ;
        u[j+j*ldu] = 1.0 ;
    }
}

if( nct >= 1 ) {
for( ll = 0 ; ll < nct ; ll++ ) {
    l = nct - ll - 1 ;
    if( s[l] == 0.0 ) {
        for( i = 0 ; i < n ; i++ ) u[i+l*ldu] = 0.0 ;
        u[l+l*ldu] = 1.0 ;
    }
    else {
        lp1 = l + 1 ;
        if( ncu > lp1 ) {
            for( j = lp1 ; j < ncu ; j++ ) {
                t = -sdot( n-l, &(u[l+l*ldu]), 1, &(u[l+j*ldu]), 1 ) /
                        u[l+l*ldu] ;
                saxpy( n-l, t, &(u[l+l*ldu]), 1, &(u[l+j*ldu]), 1) ;
            }
        }
        sscal( n-l, -1.0, &(u[l+l*ldu]), 1 ) ;
```

```
        u[l+l*ldu] = 1.0 + u[l+l*ldu] ;
        lm1 = l - 1 ;
        if( l > 0 ) {
            for( i = 0 ; i <= lm1 ; i++ ) u[i+l*ldu] = 0.0 ;
        }
    }
}
}
}
/* If it is required, generate v                                    */
if( wantv ) {
    for( ll = 0 ; ll < p ; ll++ ) {
        l = p - ll - 1 ;
        lp1 = l + 1 ;
        if( l < nrt )  {
            if( e[l] != 0.0 ) {
                for( j = lp1 ; j < p ; j++ ) {
                    t = -sdot( p-l-1, &(v[lp1+l*ldv]), 1, &(v[lp1+j*ldv]),
                            1 ) / v[lp1+l*ldv] ;
                    saxpy( p-l-1, t, &(v[lp1+l*ldv]),1,&(v[lp1+j*ldv]),1) ;
                }
            }
        }
        for( i = 0 ; i < p ; i++ ) v[i+l*ldv] = 0.0 ;
        v[l+l*ldv] = 1.0 ;
    }
}

/****  main iteratin loop for singular values ****/

mm = m ;
iter = 0 ;

/**** quit if all the singular values have been found ****/

while( iter < MAXIT ) {

if( m == 0 ) break ;

/*     This section of the program inspects for negigible elements in the
       s and e arrays.  On completion the variables kase and l are set as
       follows:

        kase = 1     if s[m-1] and e[l-1] are neglible & l<(m-1)
        kase = 2     if s[l] is negligible and l < (m-1)
        kase = 3     if e[m-2] is negligible, l < (m-1), and
                s[l],...,s[m-1] are not convergible (qr step).
        kase = 4     if e[m-2] is negigible (convergence)
*/
```

```
for( ll = 0 ; ll < m ; ll++ ) {
    l = m - ll - 2 ;
    if( l < 0 )  {
       break ;
    }
    test = fabs( s[l] ) + fabs( s[l+1] ) ;
    ztest = test + fabs( e[l] ) ;
    if( ztest == test ) {
        e[l] = 0.0 ;
        break ;
    }
}

/***** begin if *****/

if( l != m-2 ) {
    lp1 = l + 1 ;
    mp1 = m + 1 ;

    for( lls = lp1 ; lls < mp1 ; lls++ ) {
        ls = m - lls + l ;
        if( ls == l ) break ;
        test = 0.0 ;
        if( ls != m-1 ) test = test + fabs( e[ls] ) ;
        if( ls != l+1 ) test = test + fabs( e[ls-1] ) ;
        ztest = test + fabs( s[ls] ) ;
        if( ztest == test ) {
            s[ls] = 0.0 ;
            break ;
        }
    }

    if( ls == l ) kase = 3 ;
    else if( ls == m-1 ) kase = 1 ;
    else {
        kase = 2 ;
        l = ls ;
    }
}
else kase = 4 ;

/***** end if *****/

l = l + 1 ;

switch( kase ) {
    case 1:         /* Deflate negligible s[m-1] */
        mm1 = m - 1 ;
```

```
    f = e[m-2] ;
    e[m-2] = 0.0 ;
    for( kk = l ; kk < mm1 ; kk++ ) {
        k = mm1 - kk + l - 1 ;
        t1 = s[k] ;
        srotg( &t1, &f, &cs, &sn ) ;
        s[k] = t1 ;
        if( k != l ) {
            f = -sn * e[k-1] ;
            e[k-1] = cs * e[k-1] ;
        }
        if( wantv ) srot( p, &(v[0+k*ldv]), 1, &(v[0+(m-1)*ldv]), 1,
                    cs, sn ) ;
    }
    break ;
case 2:         /* Split at negigible s[l]                      */
    f = e[l-1] ;
    e[l-1] = 0.0 ;
    for( k = l ; k < m ; k++ ) {
        t1 = s[k] ;
        srotg( &t1, &f, &cs, &sn ) ;
        s[k] = t1 ;
        f = -sn * e[k] ;
        e[k] = cs * e[k] ;
        if( wantu ) srot( n, &(u[0+k*ldu]), 1, &(u[0+(l-1)*ldu]), 1,
                    cs, sn ) ;
    }
    break ;
case 3:         /* Perform one QR step                          */
        /* Calculate the shift                          */
    scale = (fabs(s[m-1] ) > fabs(s[m-2]))? fabs(s[m-1]) : fabs(s[m-2]);
    scale = (scale > fabs( e[m-2] ) ) ? scale : fabs( e[m-2] ) ;
    scale = (scale > fabs( s[l] ) ) ? scale : fabs( s[l] ) ;
    scale = (scale > fabs( e[l] ) ) ? scale : fabs( e[l] ) ;
    sm = s[m-1] / scale ;
    smm1 = s[m-2] / scale ;
    emm1 = e[m-2] / scale ;
    sl = s[l] / scale ;
    el = e[l] / scale ;
    b = ( ( smm1 + sm ) * ( smm1 - sm ) + ( emm1 * emm1 ) ) / 2.0 ;
    c = ( sm * emm1 ) * ( sm * emm1 ) ;
    shift = 0.0 ;
    if( b != 0.0 || c != 0.0 ) {
        shift = sqrt( b*b + c ) ;
        if( b < 0.0 ) shift = - shift ;
        shift = c / ( b + shift ) ;
    }
    f = ( sl + sm ) * ( sl - sm ) - shift ;
    g = sl * el ;
```

```
        /* chase zeros */
        mm1 = m - 1 ;
        for( k = l ; k < mm1 ; k++ ) {
            srotg( &f, &g, &cs, &sn ) ;
            if( k != l ) e[k-1] = f ;
            f = cs * s[k] + sn * e[k] ;
            e[k] = cs * e[k] - sn * s[k] ;
            g = sn * s[k+1] ;
            s[k+1] = cs * s[k+1] ;
            if( wantv ) srot( p, &(v[k*ldv]), 1, &(v[(k+1)*ldv]), 1,cs,sn) ;
            srotg( &f, &g, &cs, &sn ) ;
            s[k] = f ;
            f = cs * e[k] + sn * s[k+1] ;
            s[k+1] = -sn * e[k] + cs * s[k+1] ;
            g = sn * e[k+1] ;
            e[k+1] = cs * e[k+1] ;
            if( wantu && k < n-1 ) srot( n, &(u[k*ldu]), 1, &(u[(k+1)*ldu]),
                                    1, cs, sn ) ;
        }
        e[m-2] = f ;
        iter = iter + 1 ;
        break ;
    case 4:     /* Make the singular value positive                */
        if( s[l] < 0.0 ) {
            s[l] = -s[l] ;
            if( wantv ) sscal( p, -1.0, &(v[l*ldv]), 1 ) ;
        }
        /* Order the singular value */
        while( l != mm - 1 ) {
            if( s[l] >= s[l+1] ) break ;
            t = s[l] ;
            s[l] = s[l+1] ;
            s[l+1] = t ;
            if( wantv && l < (p-1) ) sswap( p, &(v[l*ldv]), 1,
                                        &(v[(l+1)*ldv]), 1 ) ;
            if( wantu && l < (n-1) ) sswap( n, &(u[l*ldu]), 1,
                                        &(u[(l+1)*ldu]), 1 ) ;
            l = l + 1 ;
        }
        iter = 0 ;
        m = m - 1 ;
        break ;
    default:
        fprintf( stderr," error in svd.c -- switch /n" ) ;
        break ;
}
}
```

```
/**** if too many iterations have been performed, set
      flag and return                                    ****/

/* Transform 's' into an mm by mm matrix with the eigenvalues stored on
   the diagonal.                                          */

for( i = 0 ; i < mm ; i++ ) s_temp[i] = s[i] ;

for( i = 0 ; i < mm ; i++ ) {
     for( j = 0 ; j < mm ; j++ ) s[i+j*mm] = 0.0 ;
     s[i+i*mm] = s_temp[i] ;
}

if( iter >= MAXIT ) info = m ;

return ;

}
/*-----------------------------------------------------------------*/

/* FUNCTION:    Constant times a vector plus a vector.
                Uses unrolled loops for increments equal to one.        */


saxpy( n, sa, sx, incx, sy, incy )

double sa ;             /* INPUT: constant                              */
double sx[] ;           /* INPUT: vector to be multiplied by sa         */
double sy[] ;           /* INPUT/OUTPUT: sy = sy + sa * sx              */
int incx ;              /* INPUT: increment of sx vector (usually one)  */
int incy ;              /* INPUT: increment of sy vector (usually one)  */
int n ;                 /* INPUT: number of elements in vector to change
                               (usually the vector dimension)        */

{

int i, ix, iy, m ;

if( n <= 0 ) return ;
if( sa == 0.0 ) return ;
if( incx != 1 || incy != 1 ) {
       ix = 0 ;
       iy = 0 ;
       if( incx < 0 ) ix = (-n+1) * incx ;
       if( incy < 0 ) iy = (-n+1) * incy ;
       for( i = 0 ; i < n ; i++ ) {
              sy[iy] = sy[iy] + sa * sx[ix] ;
              ix = ix + incx ;
              iy = iy + incy ;
```

```
        }
        return ;
}

/***** Code for both increments equal to one *****/
/***** Clean-up loop *****/

m = n % 4 ;

if( m != 0 ) {
        for( i = 0 ; i < m ; i++ ) sy[i] = sy[i] + sa * sx[i] ;
        if( n < 4 ) return ;
}

for( i = m ; i < n ; i += 4 ) {
        sy[i] = sy[i] + sa * sx[i] ;
        sy[i+1] = sy[i+1] + sa * sx[i+1] ;
        sy[i+2] = sy[i+2] + sa * sx[i+2] ;
        sy[i+3] = sy[i+3] + sa * sx[i+3] ;
}

return ;

}

/*-----------------------------------------------------------------*/

/* FUNCTION:    Copies a vector, x, to a vector, y.  Uses unrolled loops
                for increments equal to 1.                          */

scopy( n, sx, incx, sy, incy )

int n ;              /* INPUT: number of elements in sx to change
                        (usually vector dimension)          */
int incx ;           /* INPUT: increment of x                  */
int incy ;           /* INPUT: increment of y                  */
double sx[] ;        /* INPUT: vector to copy from             */
double sy[] ;        /* OUTPUT: vector to copy to              */

{

int i ;              /* Loop counter                           */
int ix, iy ;         /* Incremented vector element number      */
int m ;              /* modulus of n divided by 7              */

if( n <= 0 ) return ;

/*      Code for unequal increments or equal increments not equal to 1  */
```

```
if( incx != 1 || incy != 1 ) {
        ix = 0 ;
        iy = 0 ;
        if( incx < 0 ) ix = ( -n + 1 ) * incx ;
        if( incy < 0 ) iy = ( -n + 1 ) * incy ;
        for( i = 0 ; i < n ; i++ ) {
                sy[iy] = sx[ix] ;
                ix = ix + incx ;
                iy = iy + incy ;
        }

        return ;

}

/*      Code for both increments equal to 1     */

/*      Clean-up loop   */

m = n % 7 ;

if( m != 0 ) {
        for( i = 0 ; i < m ; i++ ) sy[i] = sx[i] ;
        if( n < 7 ) return ;
}

for( i = m ; i < n ; i+=7 ) {
        sy[i] = sx[i] ;
        sy[i+1] = sx[i+1] ;
        sy[i+2] = sx[i+2] ;
        sy[i+3] = sx[i+3] ;
        sy[i+4] = sx[i+4] ;
        sy[i+5] = sx[i+5] ;
        sy[i+6] = sx[i+6] ;
}

return ;

}

/*----------------------------------------------------------------------*/

/* FUNCTION:    Forms the dot product of two vectors.
                Uses unrolled loops for increments equal to one.        */


double sdot( n, sx, incx, sy, incy )

int n ;                 /* INPUT: Number of vector elements to be changed
```

```
                        (usually the vector dimension)       */
int incx ;            /* INPUT: Increment of sx vector (usually one)  */
int incy ;            /* INPUT: Increment of sy vector (usually one)  */
double sx[] ;          /* INPUT: vector                        */
double sy[] ;          /* INPUT: vector                        */

{

int i, ix, iy, m ;
double stemp ;

stemp = 0.0 ;
if( n <= 0 ) return( stemp ) ;

/*      Code for unequal increments or equal increments not equal to 1  */

if( incx != 1 || incy != 1 ) {
        ix = 0 ;
        iy = 0 ;
        if( incx < 0 ) ix = ( -n + 1 ) * incx ;
        if( incy < 0 ) iy = ( -n + 1 ) * incy ;
        for( i = 0 ; i < n ; i++ ) {
                stemp = stemp + sx[ix] * sy[iy] ;
                ix = ix + incx ;
                iy = iy + incy ;
        }
        return( stemp ) ;
}

/*      Code for both increments equal to one   */

/*      Clean-up loop   */

m = n % 5 ;

if( m != 0 ) {
        for( i = 0 ; i < m ; i++ ) stemp = stemp + sx[i] * sy[i] ;
}

if( n >= 5 ) {
        for( i = m ; i < n ; i+=5 ) {
                stemp = stemp + sx[i] * sy[i] + sx[i+1] * sy[i+1] +
                        sx[i+2] * sy[i+2] + sx[i+3] * sy[i+3] + sx[i+4] *
                        sy[i+4] ;
        }
}


return( stemp ) ;
```

```
}

/*---------------------------------------------------------------------*/

/* FUNCTION:    Receives two arguments and returns the first argument with
                the sign of the second argument.                       */


/* ENTRY POINT:  */

double sign( first , second )  /* RETURN: value of "first" with the sign
                                  of "second"                          */

double first ;          /* INPUT: first value -- will be sent back with
                             sign of the second value                  */
double second ;         /* INPUT: second value                       */

{

double third ;          /* OUTPUT: first value with sign of second value */

if( second >= 0.0 )
       third = fabs( first ) ;
else
       third = fabs( first ) * ( -1.0 ) ;

return( third ) ;

}

/*---------------------------------------------------------------------*/
/* FUNCTION:    Returns the euclidean norm of the n-vector stored in sx[]
                with storage increment incx.                           */

/*      If n < 0 , return with result = 0.
       if n >= 1 then incx must be >= 1.

       Four phase method using two built-in constants that are
       hopefully applicable to all machines.
              cutlo = maximum of sqrt( u/eps ) over all machines.
              cuthi = minimum of sqrt( v )     over all machines.
       where:
              eps = smalles no. such that eps + 1 > 1
              u   = smallest positive no.    ( underflow limit )
              v   = largest no.              ( overflow limit )

       Brief outline of algorithm --
```

```
        Phase 1 scans zero components.
        Move to phase 2 when a component is nonzero and <= cutlo
        Move to phase 3 when a component is > cutlo
        Move to phase 4 when a component is >= cuthi/m
        Where m = n for x[] real and m = 2*n for complex.

        Values for cutlo and cuthi.
*/

double snrm2( n, sx, incx )

int n ;         /* INPUT: Input vector dimension                    */
int incx ;      /* INPUT: Increment of x                            */
double sx[] ;   /* INPUT: Vector sx                                 */

{

int next_phase ;              /* Flag indicating next_phase phase of
                                 algorithm                          */
int next_calc ;         /* Flag indicating which sequential step algorithm
                           level must execute                       */
int i ;                 /* Loop counter                             */
int j ;                 /* Loop counter                             */
int n_x_incx ;          /* Vector dimension times increment of x    */

double xmax ;           /* Absolute value of array element          */
double hitest ;         /* cuthi / double( n )                      */
double zero = 0.0 ;             /* Machine dependent 'zero'         */
double one = 1.0 ;              /* Machine dependent 'one'          */
double cutlo = 4.441e-16 ;      /* Machine dependent                */
double cuthi = 1.304e19 ;       /* Machine dependent                */
double sum ;                    /* Algorithm parameter              */


if( n <= 0 ) return( zero ) ;

next_phase = PRE_PHASE_CHECK ;
next_calc = LEVEL_1 ;
sum = zero ;
n_x_incx = n * incx ;
i = 0 ;

for( ;; ) {


switch( next_phase ) {
    case PRE_PHASE_CHECK :
        if( fabs( sx[i] ) > cutlo ) {
            next_calc = LEVEL_2 ;
```

```
            break ;
        }
        else {
            next_phase = PHASE_1 ;
            xmax = zero ;
        }
    case PHASE_1 :
        if( sx[i] == zero ) {
            next_calc = LEVEL_6 ;
            break ;
        }
        else if( fabs( sx[i] ) > cutlo ) {
            next_calc = LEVEL_2 ;
            break ;
        }
        else {
            next_phase = PHASE_2_AND_4 ;
            next_calc = LEVEL_4 ;
            break ;
        }
    case PHASE_2_AND_4 :
        if( fabs( sx[i] ) > cutlo ) {
            next_calc = LEVEL_1 ;
            break ;
        }
    case PHASE_3 :
        if( fabs( sx[i] ) <= xmax ) {
            next_calc = LEVEL_5 ;
            break ;
        }
        else {
            sum = one + sum * ( xmax / sx[i] ) * ( xmax / sx[i] ) ;
            xmax = fabs( sx[i] ) ;
            next_calc = LEVEL_6 ;
            break ;
        }
    default :
        fprintf(stderr,"PROGRAM FAILED IN 'snmr2'- switch 'next_phase'/n" ) ;
        exit(1) ;
        break ;
}    /* end switch( next_phase ) */



switch( next_calc ) {
    case LEVEL_1:
        sum = sum * xmax * xmax ;
    case LEVEL_2 :
        hitest = cuthi / ( ( double ) ( n ) ) ;
```

```
        for( j = i ; j < n_x_incx ; j += incx ) {
            if( fabs( sx[j] ) >= hitest ) {
                next_calc = LEVEL_3 ;
                break ;
            }
            sum = sum + sx[j] * sx[j] ;
        }
        if( next_calc != LEVEL_3 ) return( sqrt( sum ) ) ;
    case LEVEL_3 :
        i = j ;
        next_phase = PHASE_3 ;
        sum = ( sum / sx[i] ) / sx[i] ;
    case LEVEL_4 :
        xmax = fabs( sx[i] ) ;
    case LEVEL_5 :
        sum = sum + ( sx[i] / xmax ) * ( sx[i] / xmax ) ;
    case LEVEL_6 :
        i = i + incx ;
        if( i < n_x_incx ) break ;
        else {
            return( xmax * sqrt( sum ) ) ;
        }
    default :
        fprintf(stderr,"PROGRAM FAILED IN snmr2 switch 'next_calc'/n") ;
        exit ( 1 ) ;
        break ;
}    /* end switch( next_calc ) */

}    /* end for( ;; ) */

}

/*---------------------------------------------------------------------*/
/* FUNCTION:    */


srot( n, sx, incx, sy, incy, c, s )

int n ;              /* INPUT: Number of elements in vector to change
                            (usually vector dimension)          */
int incx ;           /* INPUT: Increment of sx vector (usually one)  */
int incy ;           /* INPUT: Increment of sy vector (usually one)  */
double sx[] ;        /* INPUT/OUTPUT: vector                         */
double sy[] ;        /* INPUT/OUTPUT: vector                         */
double c ;           /* INPUT: constant                              */
double s ;           /* INPUT: constant                              */

{
```

```
int i, ix, iy ;
double stemp ;

if( n <= 0 ) return ;

if( incx != 1 || incy != 1 ) {
        ix = 0 ;
        iy = 0 ;
        if( incx < 0 ) ix = ( -n + 1 ) * incx ;
        if( incy < 0 ) iy = ( -n + 1 ) * incy ;
        for( i = 0 ; i < n ; i++ ) {
                stemp = c * sx[ix] + s * sy[iy] ;
                sy[iy] = c * sy[iy] - s * sx[ix] ;
                sx[ix] = stemp ;
                ix = ix + incx ;
                iy = iy + incy ;
        }
return ;
}

/*      Code for both increments equal to 1                     */

for( i = 0 ; i < n ; i++ ) {
        stemp = c * sx[i] + s * sy[i] ;
        sy[i] = c * sy[i] - s * sx[i] ;
        sx[i] = stemp ;
}

return ;

}

/*-----------------------------------------------------------------*/

/* FUNCTION:    Construct givens plane rotation                    */


extern double sign() ;          /* sign( a, b ) : returns the value of a
                                   with the sign of b               */

srotg( sa, sb, c, s )

double *sa ;
double *sb ;
double *c ;
double *s ;

{
```

```
double roe, scale, r, z ;

roe = *sb ;
if( fabs(*sa) > fabs(*sb) ) roe = *sa ;
scale = fabs(*sa) + fabs(*sb) ;
if( scale == 0.0 ) {
        *c = 1.0 ;
        *s = 0.0 ;
        r = 0.0 ;
}
else {
        r = scale * sqrt( (*sa/scale) * (*sa/scale) + (*sb/scale) *
                        (*sb/scale) ) ;
        r = sign( 1.0, roe ) * r ;
        *c = *sa / r ;
        *s = *sb / r ;
}

z = 1.0 ;

if( fabs( *sa ) > fabs( *sb ) ) z = *s ;
if( fabs( *sb ) >= fabs( *sa ) && *c != 0.0 ) z = 1.0 / *c ;
*sa = r ;
*sb = z ;

return ;

}

/*-----------------------------------------------------------------*/
/* FUNCTION:    Scales a vector by a constant.
                Uses unrolled loops for increment equal to 1          */


sscal( n, sa, sx, incx )

int n ;                 /* INPUT: vector dimension -- n by 1          */
int incx ;              /* INPUT: increment to allow multiplication of
                             only the 'incx-th' elements of the
                             input vector by scalar sa             */
double sa ;             /* INPUT: scalar to multiply the vector by    */
double sx[] ;           /* INPUT/OUTPUT:  Vector to be multiplied by
                                scalar sa                  */

{

int i, m ;              /* Loop counters                              */
int nincx ;             /* Vector dimension multiplied by incx        */
```

```
if( n <= 0 ) return ;

if( incx != 1 ) {
        nincx = n * incx ;
        for( i = 0 ; i < nincx ; i+=incx ) sx[i] = sa * sx[i] ;
}
else {  /* Code for increment equal to 1 */
        /* Clean-up loop */
        m = n % 5 ;
        if( m != 0 ) {
                for( i = 0 ; i < m ; i++ ) sx[i] = sa * sx[i] ;
                if( n < 5 ) return ;
        }
        for( i = m ; i < n ; i+=5 ) {
                sx[i] = sa * sx[i] ;
                sx[i+1] = sa * sx[i+1] ;
                sx[i+2] = sa * sx[i+2] ;
                sx[i+3] = sa * sx[i+3] ;
                sx[i+4] = sa * sx[i+4] ;
        }
}
return ;
}

/*---------------------------------------------------------------------*/

/* FUNCTION:    Interchanges two vectors.                              */


sswap( n, sx, incx, sy, incy )

int n ;              /* INPUT: number of elements to change in vector
                        (usually vector dimension)          */
int incx ;           /* INPUT: Increment of sx vector              */
int incy ;           /* INPUT: Increment of sy vector              */
double sx[] ;        /* INPUT/OUTPUT:  sx vector returned with sy
                        vector values              */
double sy[] ;        /* INPUT/OUTPUT:  sy vector returned with sx
                        vector values              */

{

int i ;              /* Loop counter                          */
int ix ;             /* Incremented sx array indeces               */
int iy ;             /* Incremented sy array indeces               */
int m ;              /* Modulus of m % 3                      */
double stemp ;       /* Temporary storage variable                 */

if( n <= 0 ) return ;
```

```
if( incx != 1 || incy != 1 ) {
        ix = 0 ;
        iy = 0 ;
        if( incx < 0 ) ix = ( -n + 1 ) * incx ;
        if( incy < 0 ) iy = ( -n + 1 ) * incy ;
        for( i = 0 ; i < n ; i++ ) {
                stemp = sx[ix] ;
                sx[ix] = sy[iy] ;
                sy[iy] = stemp ;
                ix = ix + incx ;
                iy = iy + incy ;
        }
        return ;
}

/*      Code for both increments equal to one                   */
/*      Clean-up loop                                   */

m = n % 3 ;
if( m != 0 ) {
        for( i = 0 ; i < m ; i++ ) {
                stemp = sx[i] ;
                sx[i] = sy[i] ;
                sy[i] = stemp ;
        }
        if( n < 3 ) return ;
}

for( i = m ; i < n ; i+=3 ) {
                stemp = sx[i] ;
                sx[i] = sy[i] ;
                sy[i] = stemp ;
                stemp = sx[i+1] ;
                sx[i+1] = sy[i+1] ;
                sy[i+1] = stemp ;
                stemp = sx[i+2] ;
                sx[i+2] = sy[i+2] ;
                sy[i+2] = stemp ;
}

return ;

}
/*****************************************************************************/


/*****************************************************************************/
/*                      MESSAGES                                */
/*---------------------------------------------------------------*/
```

```
/* Author : Sunil Fotedar       Place : Rice University              */
/*-----------------------------------------------------------------*/
/* Abstract: Various message strings are displayed under different   */
/*          conditions.                                            */
/*******************************************************************/

void    message_string()

{

printf("\nEnter the set of `Euler angles' (roll1, yaw1, & pitch1\n");
printf("for F12; and roll2, yaw2, & pitch2 for F1S), where\n\n");
printf("F12: Frame transformation matrix between frames F1 and F2, and\n");
printf("F1S: Frame transformation matrix between F1 and the standard frame S.\n");
printf("\nHit <ENTER> when ready, or <Control-C> to exit the program.\n");

}

void    message_stringI()

{

printf("\nThe program works for any angle of rotation;\n");
printf("n1, n2, & n3 <= 1; and n1*n1 + n2*n2 + n3*n3 = 1.0.\n");
printf("\nEnter the number of data-points used,\n");
printf("the `directional cosines' ( n1 & n2 ) of an arbitrary\n");
printf("axis, the angle of rotation `theta' around the axis,\n");
printf("and the translations along the three axes ( tx,ty,tz ).\n");
printf("\nHit <ENTER> when ready, or <Control-C> to exit the program.\n");

}

/*******************************************************************/

void message_stringII()

{

printf("\nThe program works for : \n");
printf("-90 deg <= Roll < 90 deg;");
printf("\n0 deg <= Yaw < 360 deg;");
printf("\n-180 deg <= Pitch < 180 deg.\n");

printf("\nEnter the number of data-points used,\n");
printf("\nthe `Euler angles' ( Roll, Yaw & Pitch),\n");
printf("and the translations along the three axes ( tx,ty,tz ).\n");
printf("\nHit <ENTER> when ready, or <Control-C> to exit the program.\n");

}
```

```
/*************************************************************************/

void message_stringIII()

{

printf("\nThe program works for any angle of rotation, and\n");
printf("n1, n2, & n3 <= 1 and n1*n1 + n2*n2 + n3*n3 = 1.0.\n");

printf("\nEnter the number of data-points used.\n");
printf("\nHit <ENTER> when ready, or <Control-C> to exit the program.\n");


}

/*************************************************************************/

void message_stringIV()

{

printf("\nThe program works for : \n");
printf("-90 deg <= Roll < 90 deg;");
printf("\n0 deg <= Yaw < 360 deg;");
printf("\n-180 deg <= Pitch < 180 deg.\n");

printf("\nEnter the number of data-points used.\n");
printf("\nHit <ENTER> when ready, or <Control-C> to exit the program.\n");

}
/*************************************************************************/


/*************************************************************************/
/*            CONSTRAINTS OF MOTION PARAMETERS                           */
/*-----------------------------------------------------------------------*/
/* Author: Sunil Fotedar       Place: Rice University  (1988)      */
/*-----------------------------------------------------------------------*/
/* Abstract: The following routines impose certain restrictions on   */
/*          choosing values of motion parameters (elements of first */
/*          and second representations of rotation matrix) for      */
/*          running and testing the IPC/GIPC algorithm.             */
/*************************************************************************/

constraints_A(motion)

struct motion_parameters *motion;
```

```
{
if((motion->a>1) || (motion->b>1) || (motion->a*motion->a+motion->b*motion->b>1) || (motion-
>theta
== 0) || (motion->theta ==  180) || (motion->theta == 90) || (motion->theta == 360) ||  (motion->the-
ta ==
270)) {
printf("\nSorry, the program won't work under such conditions.\n");
        exit(0);
}
        if(motion->a < 0) motion->a = (-1)*motion->a;
        if(motion->b < 0) motion->b = (-1)*motion->b;
}

/*****************************************************************************/

constraints_B(motion)

struct motion_parameters *motion;

{

if((motion->roll >= 90) || (motion->roll < -90) || (motion->yaw >= 360) || (motion->yaw < 0) || (mo-
tion-
>pitch >= 180) || (motion->pitch  < -180)) {
printf("\nSorry, the program won't work under such conditions.\n");
        exit(0);
}
}
/*****************************************************************************/


/*****************************************************************************/
/*            CONSTRAINTS OF MOTION PARAMETERS                   */
/*-------------------------------------------------------------*/
/* Author: Sunil Fotedar        Place: Rice University  (1988)       */
/*-------------------------------------------------------------*/
/* Abstract: The following routines impose certain restrictions on   */
/*          choosing values of motion parameters (elements of first */
/*          and second representations of rotation matrix) for      */
/*          running and testing the IPC/GIPC algorithm.             */
/*****************************************************************************/

constraints_A(motion)

struct motion_parameters *motion;

{
if((motion->a>1) || (motion->b>1) || (motion->a*motion->a+motion->b*motion->b>1) || (motion-
>theta
```

```
== 0) || (motion->theta ==  180) || (motion->theta == 90) || (motion->theta == 360) ||  (motion->the-
ta ==
270)) {
printf("\nSorry, the program won't work under such conditions.\n");
        exit(0);
}
        if(motion->a < 0) motion->a = (-1)*motion->a;
        if(motion->b < 0) motion->b = (-1)*motion->b;
}

/*************************************************************************/

constraints_B(motion)

struct motion_parameters *motion;

{

if((motion->roll >= 90) || (motion->roll < -90) || (motion->yaw >= 360) || (motion->yaw < 0) || (mo-
tion-
>pitch >= 180) || (motion->pitch  < -180)) {
printf("\nSorry, the program won't work under such conditions.\n");
        exit(0);
}
}
/*************************************************************************/


*********
sunil.h
*********

#define MAX_POINTS 20           /* no. of rows of a real symmetric matrix */
#define POINTS 8
#define ERROR 0.1
#define MAX_DIM 20   /* LARGEST SQUARE DIMENSION FOR MATRIX INVERSION */
#define DIM     20

#define MAXIT   30      /* Maximum allowable number of iterations       */

#define         PRE_PHASE_CHECK 1       /* Check absolute value of sx[i]
                                to see if >= cutlo          */
#define         PHASE_1         2       /* Scan zero components        */
#define         PHASE_2_AND_4   3       /* Component nonzero and <= cutlo or
                                >= ( cuthi / n )            */
#define         PHASE_3         4       /* Component > cutlo           */
#define         LEVEL_1         5       /* 'next_calc' top level       */
#define         LEVEL_2         6
#define         LEVEL_3         7
```

```
#define       LEVEL_4       8
#define       LEVEL_5       9
#define       LEVEL_6       10

/* Structure Definitions */

struct object_definition {
       int     points;                          /* # of feature points on object's surface */
      double  x[MAX_POINTS], y[MAX_POINTS], z[MAX_POINTS]; /* 3D coordinates before
motion */
      double  xr[MAX_POINTS], yr[MAX_POINTS], zr[MAX_POINTS]; /* 3D coordinates after
motion
*/
       double  xi[MAX_POINTS], yi[MAX_POINTS];        /* 2D coordinates before motion */
       double  xri[MAX_POINTS], yri[MAX_POINTS];  /* 2D coordinates after motion */
      double  x1[MAX_POINTS], y11[MAX_POINTS], z1[MAX_POINTS]; /* Estimated coordi-
nates of
features, */
     double  xr1[MAX_POINTS], yr1[MAX_POINTS], zr1[MAX_POINTS]; /* corresponding to
two
solutions of */
       double  x2[MAX_POINTS], y2[MAX_POINTS], z2[MAX_POINTS];  /* Rotation matrix */
       double  xr2[MAX_POINTS], yr2[MAX_POINTS], zr2[MAX_POINTS];
} object;

struct motion_parameters {
       double T[3];                      /* Reference Translational vector */
       double  a, b, c, theta;        /* First representation of Rotation matrix */
       double  roll, yaw, pitch;   /* Second representation of Rotation matrix */
       double  R[3][3];                  /* Reference Rotation matrix */
} motion;



**********
ESSENT.DAT:
**********
6.619719
27.771080
-19.695007
-28.613628
6.076448
8.347507
16.869179
-13.307998

**********
NASA.DAT:
**********
```

0	-1	-1	-4.44949	-1.303225	1.00000
-1.5	-1	-0.5	-1.936348	0.633123	1.00000
-2	1	1.0	-0.449490	0.767327	1.00000
-1.5	1	-0.5	-0.644449	2.700675	1.00000
-1.5	1	2.5	-0.136105	0.359012	1.00000
0	1	-1	3.414214	7.595754	1.00000
1.5	1	-0.5	3.222247	2.700675	1.00000
2	1	1.0	1.348469	0.767327	1.00000