Appendix A

Cavity Simulator Codes

```
/* Cavity Sim
 * MSci Physics Project QOLS07, Imperial College
 * Jarvist Frost & Benjamin Hall 2005-2006
 */

#include <complex.h>
#include <fftw3.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "pnpoly.c"              //points in a polygon detection code

//#define TWO_DIMENSIONAL

#define N (1024*16)
#define G (0.2)          //guard band - expressed as fraction of grid we
    SHOULD uss
#define TOLERANCE 0.0001         //tolerance of points in testing for
    eigenmode
#define SAMPLEPOINTS 200         //number of random points to sample to
    test for Eigenmode
#define MAX_EIGENS 10

#define SPACERS 10 //number of non-shift applying 'spacers' in
    eigenvalue meth

#define A 1.414E-2                        //was 1mm
double L = 1.0; //3.14;          //was 0.001
double FOCAL = -100000;
double LAMBDA= 10e-08; //1.0E-7; //0.4488E-6; //0.11225E-6; //0.069754E
    -6; //0.067349E-6; //0.069754E-6 //1 micron

int CROP=1; //crop output photos?
fftw_plan fft, fftr;
double M;

#define R 500000.0               //radius curvature gaussian spherical
    beam
#define WAIST 1E-3               //waist of g.s. beam

#ifdef TWO_DIMENSIONAL
fftw_complex ap[N][N];
fftw_complex out[N][N];
fftw_complex old_ap[N][N];
int filter[N][N];
#endif

#ifndef TWO_DIMENSIONAL
fftw_complex ap[N][1];
```

```
fftw_complex out[N][1];
fftw_complex old_ap[N][1];
int filter[N][1];
#endif

double RS[N];

fftw_complex gamma_shift[MAX_EIGENS];
fftw_complex gamma_old, gamma_new;

struct coord
{
  int x, y;
} samples[SAMPLEPOINTS];

double rescaled_range(int start, int length);
double hurst();
void input_ap_picture (void);
void output_ap_slice (char *name);
void output_ap_picture (char *name);
void aperture_filter (void);
void propogate (double LENGTH);
void normalise_intensity_in_cavity (void);
void generate_initial_intensity ();
void scale_fft(); //correct for scale caused by FFT routines

void
input_ap_picture ()
{
  char buffer[200];
  int i, j, p;
  FILE *fo;

  fo = fopen ("in.raw", "r");

/*    fscanf(fo,"%s\n",&buffer);
    fscanf(fo,"%s\n",&buffer);
    fscanf(fo,"%s\n",&buffer);
    fscanf(fo,"%s\n",&buffer);
  */
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      {
        fscanf (fo, "%d\n", &p);          //red
//          fscanf(fo,"%d\n",&p); //green
//          fscanf(fo,"%d\n",&p); //blue
        ap[i][j] = (double) p;
//          printf("%d %d %d\n",i,j,p);
      }

  fclose (fo);
}

void
output_ap_slice (char *name)
{
  int i,j=0;
  FILE *fo;

  #ifdef TWO_DIMENSIONAL
    j=N/2; //take slice across centre if in 2D, otherwise just read out 1
        D array
  #endif
  fo = fopen (name, "w");
  fprintf (fo, "#index creal cimag cabs cabs*cabs\n");
  for (i = N/2; i < N; i++)
    fprintf (fo, "%f %f %f %f %f %f %f\n", (double)(i-N/2)/((double)N*G
        *0.5/sqrt(2)),
             (double)(i-N/2)/((double)N*G*0.5/sqrt(2))*sqrt(M),
             (double)(i-N/2)/((double)N*G*0.5/sqrt(2))*M,
             creal (ap[i][j]),
             cimag (ap[i][j]), cabs (ap[i][j]),
             cabs(ap[i][j]*ap[i][j])   );
  fclose(fo);
```

```
}

void
output_ap_picture (char *name)
{
  double scr = 0.0, sci = 0.0, phi, s, v, p, q, t, f, r, g, b;
  int i, j, mag, magmax = 0, Hi;
  int size,bottom,top;
  FILE *fo;

  fo = fopen (name, "w");

//    fprintf(stderr,"Output Apperture to: %s\n",name);

    if (CROP==1)
      {
//        size=(int)((float)N*G);
          size=N/2;
          bottom=N/4;
          top=3*N/4;
//        bottom=1+(int)((1-G)*(float)N/2);
//        top=N-(int)((1-G)*(float)N/2);
      }
    else
      {
          size=N; bottom=0; top=N;
      }

  fprintf (fo, "P6\n%d %d\n%d\n", size, size, 254);

  for (i = 0; i < N; i++)        //calculate scale factor
    for (j = 0; j < N; j++)
      {
        if (cabs (ap[i][j]) * cabs (ap[i][j]) > scr)
          scr = cabs (ap[i][j]) * cabs (ap[i][j]);
/*        if (cimag(ap[i][j])>sci)
            sci=cimag(ap[i][j]);*/
      }

  for (i = bottom; i < top; i++)
      //apply scale factor to normalise amount of light in cavi$
    {
      for (j = bottom; j < top; j++)
        {
          phi = M_PI + atan2 (cimag (ap[i][j]), creal (ap[i][j]));
//            phi=2*M_PI*(cabs(ap[i][j])*cabs(ap[i][j]))/scr;

          s = 1.0;

          v = (cabs (ap[i][j]) * cabs (ap[i][j])) / scr;
//            v=1.0;

//            printf("HSV: %f %f %f\n",phi,s,v);

          //HSV->RGB formula from http://en.wikipedia.org/wiki/
              HSV_color_space
          Hi = (int) (floor (phi / (M_PI / 3.0))) % 6;
          f = phi / (M_PI / 3.0) - floor (phi / (M_PI / 3.0));
          p = v * (1.0 - s);
          q = v * (1.0 - f * s);
          t = v * (1.0 - (1.0 - f) * s);

          if (Hi == 0)
            {
              r = v;
              g = t;
              b = p;
            }
          if (Hi == 1)
            {
              r = q;
              g = v;
              b = p;
```

```
            }
          if (Hi == 2)
            {
              r = p;
              g = v;
              b = t;
            }
          if (Hi == 3)
            {
              r = p;
              g = q;
              b = v;
            }
          if (Hi == 4)
            {
              r = t;
              g = p;
              b = v;
            }
          if (Hi == 5)
            {
              r = v;
              g = p;
              b = q;
            }
//            printf("%f\n",f);

          if (filter[i][j] != 0)        //if we're displaing the mask...
            r = g = b = v;       //make it greyscale!
          fprintf (fo, "%c%c%c", (int) (254.0 * r), (int) (254.0 * g),
                   (int) (254.0 * b));

//            fprintf(fo,"%d %d %d ",65535-mag,65535-mag,65535-mag);
        }
    }

  fclose (fo);

}

void
make_filter (int nsides)
{
  int i, j, n;
  float x[100], y[100];
  double grad, xinit;
  //first draw a circle
/*   for (i=0;i<N;i++) for (j=0;j<N;j++)
      if ((float)((i-N/2)*(i-N/2)+(j-N/2)*(j-N/2)) > (G*G*(float)N/2.0*(
          float)N/2.0))
        filter[i][j]=1; //masked
    else
      filter[i][j]=0; //not masked*/
  //then draw polygon within circle by chopping off edges
#ifdef TWO_DIMENSIONAL
  for (n = 0; n < nsides; n++)
    {
      x[n] =
        N / 2 -
        G * (N / 2) * cos ((M_PI/(double)nsides)+ (double) n * (2.0 *
            M_PI / (double) nsides));
      y[n] =
        N / 2 +
        G * (N / 2) * sin ((M_PI/(double)nsides)+(double) n * (2.0 *
            M_PI / (double) nsides));
    }

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      if (pnpoly (nsides, &x[0], &y[0], (float) i, (float) j))
        filter[i][j] = 0;
      else
        filter[i][j] = 1;
#endif
```

```
#ifndef TWO_DIMENSIONAL
    for (i = 0; i < N; i++)
      {

          if (i<N/2-(G*(N/2)/sqrt(2)) || i>N/2+(G*(N/2)/sqrt(2)))
            filter[i][0]=1;
          else
            filter[i][0]=0;
      }

#endif
}


void
aperture_filter ()
{
  int i, j=0;
  for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
      for (j = 0; j < N; j++)
#endif
       if (filter[i][j] != 0)    //do this with unary logic
         ap[i][j] = 0.0 + 0.0I;
}

void
propogate (double LENGTH)
{
  int i, j=N/2; //set j at centre of cavity
  double shift;

  fftw_execute (fft);

  for (i = 0; i < N; i++)        //plane wave propogation
#ifdef TWO_DIMENSIONAL
      for (j = 0; j < N; j++)
        //we need to use the modulus operator to flip the quadrants -
        // the FFT algo changes location of the high-spectral freq
       out[i][j] *= cexp (   I * M_PI*
           (1.0/(double)A)*(1.0/(double)A)*
                  (double)
           (((i + N / 2) % N - N / 2) *
                   ((i + N / 2) % N - N / 2) +
           ((j + N / 2) % N - N / 2) *
                   ((j + N / 2) % N - N / 2))
                           * ( LENGTH * (double)LAMBDA));
#endif

#ifndef TWO_DIMENSIONAL
       out[i][0] *= cexp (   I * M_PI*
           (1.0/(double)A)*(1.0/(double)A)*
                  (double)
           (((i + N / 2) % N - N / 2) *
                   ((i + N / 2) % N - N / 2))
                           * ( LENGTH * (double)LAMBDA));
#endif
/*    for (i=0;i<N;i++) //Apply Hanning Window to spectral form
           for (j=0;j<N;j++
             out[i][j]*= 0.54-0.46 *
                         cos(2*M_PI*i/(N-1))*cos(2*M_PI*j/(N-1));
*/
  fftw_execute (fftr);
  scale_fft(); //corrects for scale in FFT algorithm
}
void lens(double f) //apply spherical lens curvature to wavefrount
  //i.e. phase retardation dependent on distance from axis
{
   int i,j=0;

for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
  for (j = 0; j < N; j++)
     ap[i][j] *=
```

```
      cexp (I
            *M_PI
            / ((double)f*(double)LAMBDA)
            * (((double)A/(double)N) * ((double)A/(double)N)) *
             (double) ((i - N / 2) * (i - N / 2) +
                          (j - N / 2) * (j - N / 2)));
#endif
#ifndef TWO_DIMENSIONAL
      ap[i][0] *=
       cexp (I
            *M_PI
            / ((double)f*(double)LAMBDA)
            * (((double)A/(double)N) * ((double)A/(double)N)) *
             (double) ((i - N / 2) * (i - N / 2) ));
#endif
}

void
normalise_intensity_in_cavity ()
{
  double sc = 0.0;
  int i, j=0;

  for (i = 0; i < N; i++)        //calculate scale factor
#ifdef TWO_DIMENSIONAL
      for (j = 0; j < N; j++)
#endif
        if (filter[i][j]==0) //if within aperture
       sc += cabs (ap[i][j])*cabs(ap[i][j]);

   sc=(double)sc; //discard imaginery part
#ifdef TWO_DIMENSIONAL
   sc=sc/((double)N*(double)N*G*G); //average abs. value of pixel in
        cavity
#endif
#ifndef TWO_DIMENSIONAL
   sc=sc/((double)N*G);
#endif
   sc=sqrt(sc); //take sqrt to get
//    fprintf(stderr,"Normalise intensity: sc %f\n",sc);

  for (i = 0; i < N; i++)        //apply scale factor to normalise amount
        of light in cavity
#ifdef TWO_DIMENSIONAL
      for (j = 0; j < N; j++)
#endif
        ap[i][j]=ap[i][j]/sc;
}

void
generate_initial_intensity ()
{
  int i, j=0;

  for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
      for (j = 0; j < N; j++)
#endif
        {
//            ap[i][j]=(i-N/2)+((j-N/2)*I);

/*            if (i>0.25*N
                 && i<0.75*N
                 && j>0.25*N
                 && j<0.75*N )
                   ap[i][j]=1.0+0.0I;
              else
                ap[i][j]=0.0+0.0I;
  */
          ap[i][j] = cexp (
//                      -I*2*M_PI/LAMBDA*
//                      ((double)((i-N/2)*(i-N/2)+(j-N/2)*(j-N/2))/(double)(
    N))
//                      /R
```

```
                         _
                        (A*A)* ((double)
                          ((i - N / 2) * (i - N / 2) +
                           (j - N / 2) * (j - N / 2))
                               / (double) (N * N)) /
                         (WAIST*WAIST));
          ap[i][j] = 1.0 + 0.0I;          //DIRTY! :)

//            fprintf(stderr,"i: %d j: %d\t%f + %f i\n",i,j,creal(ap[i][j
    ]),cimag(ap[i][j]));
        }

}


void scale_fft() //correct for scale caused by FFT routines
{
   int i,j;
     for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
       for (j = 0; j < N; j++)
           ap[i][j]/=(double)N*(double)N; //correct for scaling of FFT
               algo
#endif
#ifndef TWO_DIMENSIONAL
       ap[i][0]/=(double)N; //correct for scaling of 1D FFT algo
#endif
}

void
output_filter ()
{
  int i, j;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      ap[i][j] = 1.0;
  aperture_filter ();
}

fftw_complex calculate_gamma()
//Calculate Gamma shift factor by comparing average of succesive pixels
    once stabilised on eigenmode
{
        int i,j=0,ap_points = 0;
   fftw_complex gamma_new=0.0;
            for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
     for (j = 0; j < N; j++)
#endif
                if (filter[i][j] ==0 && cabs(ap[i][j])>0.05)
                  {
                    ap_points++;
                    gamma_new += ap[i][j] / old_ap[i][j];
                  }
            gamma_new /= (double) ap_points;
//   fprintf(stderr,"%d",ap_points);
   return(gamma_new);
}

void apply_gamma_shift(int shift)
{
   int i,j=0;
                 for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
     for (j = 0; j < N; j++)
#endif
                    ap[i][j] -= old_ap[i][j] * gamma_shift[shift];
                        //rotate between gamma_shift shifts on each
                        successive pass
                //So - just what are we meant to do here? Apply
                    subtractive shift to previous frame?
//          fprintf(stderr,"Apply Gam: %f + %f I\n",creal(gamma_shift[1+
    passes%eigenmode_count])
//                  ,cimag(gamma_shift[1+passes%eigenmode_count]));
```

```
}


double hurst(char *name)
  //Derived from equations at
  // http://www.bearcave.com/misl/misl_tech/wavelets/hurst/index.html
{
  FILE *fo;
   int i=0,n;
   int bits=0,size=512;

   double data[20][2];

   double avg;
   double extent=0.4; //fraction of guard band we'll be using
   size=(int)(extent*G*N)/2.0;
   printf("%d\n",size);

   fo = fopen (name, "w");

   bits=size;

   while (bits>=8)
     {
        avg=0.0;
        for (n=0;n<size;n+=bits)
             avg+=rescaled_range(n+N/2,bits);

        avg/=(double)(size/bits); //was size/bits

        fprintf(fo,"%f %f %f %f\n",1.0/(double)bits,avg,log((double)bits
            )/log(10.0),log(avg)/log(10.0));

        data[i][0]=log((double)bits)/log(10.0); data[i][1]=log(avg)/log
            (10.0);

        i++;
        bits/=2;
     }

        fprintf(fo,"#D: Approx: %f\n",(data[0][1] - data[i-1][1])/(data
            [0][0] - data[i-1][0]) );
   fclose(fo);

   return(0.0); //FIXME
}

double rescaled_range(int start, int length)
  //Dervied from equations at
  // http://www.bearcave.com/misl/misl_tech/wavelets/hurst/index.html
{
   int i;
   double avg=0.0,min=0.0,max=0.0,sd=0.0;

   for (i=start;i<start+length;i++)
     avg+=(double)cabs(ap[i][0]);
   avg/=(double)length; //calculate avg. value

   RS[0]=0.0;
   for (i=1;i<=length;i++)
     {
     RS[i]=RS[i-1]+(double)cabs(ap[i+start][0])-avg;
     if (RS[i]<min) min=RS[i];
     if (RS[i]>max) max=RS[i];
     }
   for (i=start;i<start+length;i++)
     sd+=((double)cabs(ap[i][0])-avg)*((double)cabs(ap[i][0])-avg);

   sd/=(double)length; //now contains variance
   sd=sqrt(sd); //now S.D.

  return ((max-min)/sd);
}
```

```
main ()
{
  int i, j=0, k, l, passes, n, eigenmode_flag, eigenmode_count,
      ap_points,shift;
   int framecount=0;
  char name[100],tmp[100];
  double tmpr, tmpi, sc, total_error,Neq,conjugate_plane;
   double gimag,greal;
   double g1,g2,FOCAL_CONVERSION; //g-factors for laser cavity


#ifdef TWO_DIMENSIONAL
  fft = fftw_plan_dft_2d (N, N,
                          &ap[0][0], &out[0][0], FFTW_FORWARD,
                              FFTW_ESTIMATE);
  fftr = fftw_plan_dft_2d (N, N,
                           &out[0][0],
                           &ap[0][0], FFTW_BACKWARD, FFTW_ESTIMATE);
#endif

#ifndef TWO_DIMENSIONAL
  fft = fftw_plan_dft_2d (N, 1,
                          &ap[0][0], &out[0][0], FFTW_FORWARD,
                              FFTW_ESTIMATE);
  fftr = fftw_plan_dft_2d (N, 1,
                           &out[0][0],
                           &ap[0][0], FFTW_BACKWARD, FFTW_ESTIMATE);
#endif
  srand (time (NULL));

  fprintf (stderr, "Plans created...N:%d\n",N);

// for (M=1.5;M<1.6;M+=0.6)
  for (n = 5; n < 6; n++) //n-sided polygon for aperture
//     for (FOCAL = -2.0; FOCAL > -20.0; FOCAL -= 1.0)
      {
//        g1= -1.0526; //-1.01; //-1.055;
          g1=-1.01;
//        g1=-1.002;
//        M=1.9;
//        g1=(M+1.0) /(2.0*M);
          g2=g1;
          printf("g1=%f g2=%f\n",g1,g2);
//        FOCAL=(-M*L)/((M-1)*(M-1));
//        FOCAL=0.225;
          FOCAL_CONVERSION=-g2*L/(g2-1);

          FOCAL=1/(2-2*g1);

          conjugate_plane=(L/2)*sqrt((g1+1)/(g1-1)); //distance x from
              centre of cavity
          M=(-g1+sqrt(g1*g1 - 1))/(-g1-sqrt(g1*g1-1)); //Magnification of
               cavity
          Neq=12.0;
          LAMBDA=((M*M)-1)/(2*M)*(A*G*A*G/8.0)/(L*Neq); //choose lambda
              from previous Neq
          fprintf(stderr,"Lamda: %e Neq: %f\n",LAMBDA,Neq);
          Neq=((M*M)-1)/(2*M) * (A*G*A*G/8.0)/(L*LAMBDA); //calculate Neq
               from A,Lambda,L & M
         fprintf(stderr,"Congjugate planes: u:%f v:%f M: %f x: %f\n",L/2-
             conjugate_plane,L/2+conjugate_plane,M,conjugate_plane);

          //EQUIVALENT LENSGUIDE CONVERSIONS
//        FOCAL=-(g2*L)/(2*(g1*g2-1)); //focal length of equiv lensguide
    - Eqn 16, GHCN notes
//        L=2*g1*L; //equivalennt freescale length - Eqn 15, GHCN notes



          fprintf(stderr,"M: %f L: %f Focal: %f Focal_Conversion %f N: %f
               Neq: %f\n",
                  M,L,FOCAL,FOCAL_CONVERSION,
```

```
                (0.5*A*G*0.5*A*G/2.0)/(L*LAMBDA),
                //((1-L/FOCAL)-1)/2.0    *(0.5*A*G*0.5*A*G/2.0)/(L*
                    LAMBDA));
                Neq);
//        ((M-1)/2.0 * (A*G*A*G/8.0)/(L*LAMBDA)));


//    for (i=0;i<N;i++) for (j=0;j<N;j++)
        //      shift[i][j]=0.0+0.0I;
        make_filter (n);
        fprintf (stderr, "Npolygon: %d M: %f Focal: %f\n", n, M,FOCAL);

//    for (L=0.001;L<=0.024;L+=0.001) //10 240 10
//      {
//        fprintf(stderr,"Going for Length %f\n",L);


//    input_ap_picture(); //Lena
//
        gamma_old = gamma_new = 0.0 + 0.0I;

        for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
          for (j = 0; j < N; j++)
#endif
            old_ap[i][j] = 0.0 + 0.0I;

//        for (greal=-1.0;greal<1.0;greal+=0.1)
//          for (gimag=-1.0;gimag<1.0;gimag+=0.1)
//            {
        eigenmode_count = 0;
//            gamma_shift[0]=greal+gimag*I;

        generate_initial_intensity ();

//                sprintf(name,"%.10d.pnm",0);
//                  output_ap_picture(name);
//            sprintf(name,"%.10d.log",0);
//                output_ap_slice(name);

        aperture_filter ();

//                lens(-FOCAL_CONVERSION);
          for (passes = 0; passes < 10000; passes++)
            {
//        fprintf(stderr,"Nsides: %d Passes %d\n",n,passes);
//          sprintf(name,"%.10d.pnm",framecount++);
//          output_ap_picture(name);

            //EQUIV LENSGUIDE
//                lens(FOCAL);
//                propogate (L);

            lens(FOCAL);
            propogate(L/2-conjugate_plane);
            aperture_filter();
            propogate(L/2+conjugate_plane);
            lens(FOCAL);

            propogate(L/2+conjugate_plane);
            aperture_filter();
            propogate(L/2-conjugate_plane);

            //propogate(L);

             //Gamma shift application
             //Start of SHIFT selection
            /* the following code applies the shifts in a straight
                 series
             * with a gap of SPACERS between each rotated application.
             * So it looks like abc......abc.....abc.... etc.
             */

             shift=(passes+(SPACERS-1))%(SPACERS+eigenmode_count) -
```

```
                SPACERS;

//              if (shift>=0)
//                shift=eigenmode_count-shift-1;

//            fprintf(stderr," %d ",shift);
              /* the following code applies the shifts in rotation,
                  spaced by
               * a gap of SPACERS between each single application of a
                   shift.
               * So it looks like a......b......c......a......b......c
                   .... etc.
               */
/*            if (passes%SPACERS==0 && eigenmode_count>0)
                shift=(passes/SPACERS)%eigenmode_count;
              else
                shift=-1;
*/
              //End of SHIFT selection
               if (shift<0) { fprintf(stderr,"."); sprintf(tmp,"X");}
               else
                 {
                    fprintf(stderr,"%c",'a'+shift);

                    for (i=1;i<=shift+1;i++)
                      tmp[i]='A'+i-1;
                    tmp[shift+2]=0;

                    apply_gamma_shift(shift);
                 }

            gamma_new = calculate_gamma();
              framecount++;
//                sprintf(name,"%.10d_%s_Mode:%d_G:%f+%fI.pnm",framecount,
    tmp,eigenmode_count,creal(gamma_new),cimag(gamma_new));
//                  output_ap_picture(name);
//            normalise_intensity_in_cavity ();
//            sprintf(name,"%.5d.pnm",framecount);
  //                output_ap_picture(name);
//               output_ap_slice(name);
//            exit(-1);


/*        printf("G_new %f + %fI  cabs: %f old:new  %f\n",
               creal(gamma_new),cimag(gamma_new),
             cabs(gamma_new),
               cabs(gamma_new-gamma_old)
             );
*/

          if ( passes>15 && cabs (gamma_new - gamma_old) < (double)
               TOLERANCE)      //see if stabailised to eigenmode by non-
               varying Gamma shift
               {
                 fprintf (stderr, "c@%d\n", passes);
//          fprintf(stderr,"Convergence to Eigenmode, with %f Tolerance.\n
    ",TOLERANCE);

                 gamma_shift[eigenmode_count] = gamma_new;        //save
                     gamma into shift table

                 printf ("Avg Gamma[%d]: %f + %fI\tAbs:%f\n",
                     eigenmode_count,
                          creal (gamma_shift[eigenmode_count]),
                          cimag (gamma_shift[eigenmode_count]),
                          cabs(gamma_shift[eigenmode_count]));

             sprintf(name,"%dr.log",eigenmode_count);
                 output_ap_slice(name);

                 sprintf(name,"h%d.log",eigenmode_count);
                 hurst(name);

                 //remove the following cludge
```

```
/*                  lens(FOCAL);
                    propogate (L);
                    lens(FOCAL_CONVERSION);
                    propogate (-(0.5-conjugate_plane));
*/
//                   lens(FOCAL_CONVERSION); //back to full cavity
                     lens(1/(2-2*g1)); //FOCUS as actually mirror
                     propogate(0.5+conjugate_plane); //propogate forwards

                     normalise_intensity_in_cavity ();

                     sprintf(name,"%du_lr.log",eigenmode_count);
                     output_ap_slice(name);

                     sprintf(name,"hc%d.log",eigenmode_count);
                     hurst(name);


#ifdef TWO_DIMENSIONAL //if making a 2D eigenmode, output the pretty
     eigenmode!
                    sprintf(name,"%du_lr.pnm",eigenmode_count);
                    output_ap_picture (name);
#endif

                     aperture_filter();
                     propogate (2*conjugate_plane);
                     sprintf(name,"%dv_lr.log",eigenmode_count);
                     output_ap_slice(name);

                     propogate(0.5-conjugate_plane);
                     lens(1/(2-2*g1)); //FOCUS as actually mirror
                     propogate(0.5-conjugate_plane);
                     sprintf(name,"%dv_rl.log",eigenmode_count);
                     output_ap_slice(name);

                     propogate (2*conjugate_plane);
                     sprintf(name,"%du_rl.log",eigenmode_count);
                     output_ap_slice(name);
#ifdef TWO_DIMENSIONAL //if making a 2D eigenmode, output the pretty
     eigenmode!
                                     sprintf(name,"%du_rl.pnm",
                                         eigenmode_count);
                                     output_ap_picture (name);
#endif

                    eigenmode_count++;       //keep count of already
                        discovered eigenmodes
                    passes = 0;      //reset passes so we have full range to
                        settle to next mode
                     gamma_old=gamma_new=0.0+0.0I; //reset gamma factors
                 }

              aperture_filter ();
              normalise_intensity_in_cavity ();

              for (i = 0; i < N; i++)
#ifdef TWO_DIMENSIONAL
                  for (j = 0; j < N; j++)
#endif
                    old_ap[i][j] = ap[i][j];
//        memcpy(old_ap,ap,sizeof(fftw_complex)*N*N);

              gamma_old = gamma_new;

              if (eigenmode_count >= MAX_EIGENS)  //once we've gathered
                  this many modes
                break;              //break out the for-loop!

           }
         fprintf (stderr, "Reset\n");
//            }

//       sprintf(name,"npoly%d_Foc%f_passes%.5d.pnm",n,FOCAL,passes);
//       output_ap_picture(name);
```

```
//          output_ap_slice((int)(L*1000));
      }
  fftw_destroy_plan (fft);
  fftw_destroy_plan (fftr);
}
```

Appendix B

Video Fractal Codes

```
/* Jarvist Frost 2004-2006
 * Program to create 'video-fractals'
 */

//#include <file.h>
#include <stdio.h>
#include <math.h>
#include <limits.h>

#define MAG (1.4)
#define X_RES 500
#define Y_RES 500

#define X_OFF 0 //offset of newcenter in pixels
#define Y_OFF 0

#define TWIST 3.14/6 //radians twist between zoom's

#define PIXW 0.65 //0.65 //width of sensor pixel in display pixels

#define BACKGROUND 140

int curpic[X_RES][Y_RES];
int newpic[X_RES][Y_RES];
void outputpic(char *filename);
void inputpic(char *filename);

main()
{
   int x,y,loops;
   char filename[20];

   //fill display with white noise
   srand(123);
   for (x=0;x<X_RES;x++)
     for (y=0;y<Y_RES;y++)
       curpic[x][y]=rand();

   inputpic("begin.pgm");



   outputpic("first.pgm");
//    zoom();
//    swap();
//    outputpic("test2.pgm");

   for(loops=0;loops<150;loops++)
     {
        printf("%d\n",loops);
        sprintf(filename,"pic%.3d.pgm",loops);
        if (loops%10==0)
```

```
            outputpic(filename);

         zoom(); swap();
      }

   outputpic("last.pgm");
}

swap()
{
  int x,y,max=0;
   double light;

   for (x=0;x<X_RES;x++)
      for(y=0;y<Y_RES;y++)
         {
         curpic[x][y]=newpic[x][y];
         if (curpic[x][y]>max)
            max=curpic[x][y];
         }

   for (x=0;x<X_RES;x++)
      for(y=0;y<Y_RES;y++)
         curpic[x][y]*=INT_MAX/max;
}


zoom()
{
 int x,y;
   double nx,ny,np,dx,dy,r,theta,phi;

   for (x=0;x<X_RES;x++)
      for(y=0;y<Y_RES;y++)
         {
            dx=(0.5+(double)(x-(X_RES/2)))/MAG;
            dy=(0.5+(double)(y-(Y_RES/2)))/MAG;

            r=sqrt(dx*dx+dy*dy);
            theta=atan2(dy,dx);
            phi=theta+TWIST;

            ny=r*sin(phi);
            nx=r*cos(phi);

            nx=nx+(double)(X_RES/2+X_OFF);
            ny=ny+(double)(Y_RES/2+Y_OFF);

//          printf("x: %d nx: %f y: %d ny: %f\n",x,nx,y,ny);

            np=0;
            np+=vo((int)nx+1,(int)ny+1) *(nx-(double)((int)nx))*(ny-(
                double)((int)ny)); //top-right pixel
            np+=vo((int)nx,(int)ny+1) *(PIXW-(nx-(double)((int)nx)))*(ny-(
                double)((int)ny)); //top-left pixel
            np+=vo((int)nx+1,(int)ny) *(nx-(double)((int)nx))*(PIXW-(ny-(
                double)((int)ny))); //bot-right pixel
            np+=vo((int)nx,(int)ny) *(PIXW-(double)(nx-(double)((int)nx)))
                *(PIXW-(ny-(double)((int)ny))); //bot-left pixel

            np*=1.0/(PIXW*PIXW); //compensates for size of pixel otherwise
                 'losing' light from the feedback

            newpic[x][y]=(int)np;
//          printf("np: %f\n",np);
         }


}

int vo(int x, int y) //value of a particular pixel; with bounds checking
{
   if (x<0||x>X_RES) return(BACKGROUND);
   if (y<0||y>Y_RES) return(BACKGROUND);
```

```
    return (curpic[x][y]);
}


void outputpic(char *filename)
{
    int x,y;
    FILE * f;
    f=fopen(filename,"w");

    fprintf(f,"P5 %d %d 255\n",X_RES,Y_RES); //.pgm filetype - binary
        form

    for (x=0;x<X_RES;x++)
       for (y=0;y<Y_RES;y++)
          fprintf(f,"%c",curpic[x][y]/(INT_MAX/255));

    fclose(f);
}

void inputpic(char *filename)
{
  int x,y;
    int tmp;

    FILE * f;
    f=fopen(filename,"r");

    fscanf(f,"P2 500 500\n"); //.pgm filetype

    for (x=0;x<X_RES;x++)
         for (y=0;y<Y_RES;y++)
        {
              fscanf(f,"%d",&tmp);
//          printf("%d\n",curpic[x][y]);
// printf("tmp: %d\n",tmp);
           curpic[x][y]=tmp*(INT_MAX/255);
        }


    fclose(f);
}
```