SYS_VIEW: A VISUALIZATION TOOL FOR VIEWING THE REGIONS OF VALIDITY AND ATTRACTION OF NONLINEAR SYSTEMS

by

Raja R. Kadiyala

Memorandum No. UCB/ERL M92/21

1 March 1992

SYS_VIEW: A VISUALIZATION TOOL FOR VIEWING THE REGIONS OF VALIDITY AND ATTRACTION OF NONLINEAR SYSTEMS

by

Raja R. Kadiyala

Memorandum No. UCB/ERL M92/21

1 March 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Sys_View : A Visualization Tool for Viewing the Regions of Validity and Attraction of Nonlinear Systems *

Raja R. Kadiyala

Department of Electrical Engineering
and Computer Science
207-59 Cory Hall
University of California
Berkeley, CA 94720
email: raja@robotics.berkeley.edu

March 1, 1992

Abstract

We present a visualization tool which allows one to view the stability characteristics of nonlinear ordinary differential equations in three dimensions. We find that these computations may be carried out in parallel and present an algorithm for multiple networked workstations. We also discuss various viewing alternatives for the visualization of these dynamics.

*Research supported in part by the Army under grants DAAL-88-K-0106 and DAAL-91-G-0191, and NASA under grant NAG 2-243

1 Introduction

So called phase portraits have been used to study the stability characteristics of planar two dimensional dynamical systems with a good deal of success, but we are truly limited by this visualization scheme since we have the restriction of two dimensions. We extend the concept of phase portraits to three dimensions with *Sys_View* .

Sys_View does not create three dimensional phase portraits per se. Rather, it allows one to view the stability characteristics of a three dimensional subset of the dynamics. By this we mean that one is able to see where a system is stable and hence the region of attraction and validity of a control law. *Sys_View* also allows one to view the characteristics of how a system is unstable. In short, *Sys_View* allows the user to interactively view the stability characteristics of a three dimensional system or a three dimensional submanifold of the system.

As one might imagine the task of computing the stability characteristics of a three dimensional system is a rather large computational undertaking. Fortunately we note that a majority of the calculations do not depend on each other, hence the algorithm is ideal for parallel computing. We will show such a procedure for this problem using multiple networked workstations to offload the computation.

We start with some simple definitions and an overview of the types of systems we can handle in section 2 and discuss the parallel algorithm in section 3, and lastly in section 4 we look at the visualization methods developed to survey all the information inherent in these plots.

2 Extension into Three Dimensions

Sys_View was motivated by a discrete time version which looked at extended two dimensional Mandelbrot sets into three dimensions. We first discuss the discrete time case.

2.1 Discrete Time

The defining equation for a Mandelbrot set is given by:

$$x_{k+1} = f(x_k) + R \tag{1}$$

where $x, R \in \mathbf{C}$. This gives rise to the familiar vanilla mandelbrot set shown in figure 1, where one plots all the points which converge (a discrete version

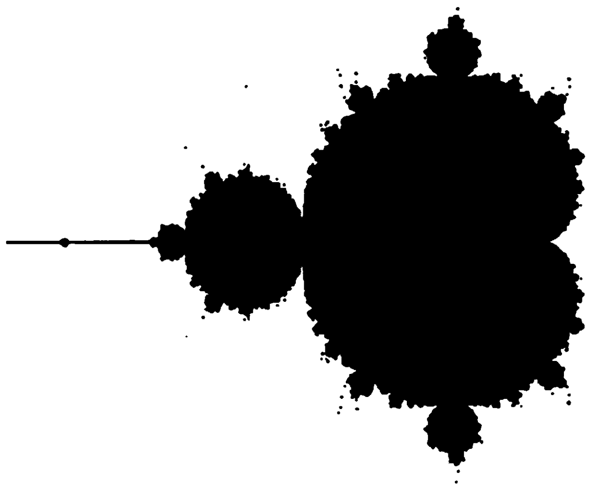

Figure 1: Unenhanced Mandelbrot Set

of the region of attraction).

These sets may be enhanced by color to give a more artistic flair and display more information about the system dynamics. A typical mapping is to assign each point a color according to how many iterations it takes to escape some finite area (the so called *escape time*).

Equation (1) is very similar to a general class of nonlinear discrete time systems which is affine in the control input. These systems may be described by:

$$x_{k+1} = f(x_k) + g(x_k)u_k \tag{2}$$

where

- x is known as the *state vector* ($\in \mathbf{R}^n$)
  - The state vector x describes the current configuration of the system.
- u is called the *control input* ($\in \mathbf{R}$)
  - The control input u is what you have control over to *move* the system.
- $f(\cdot)$ and $g_i(\cdot)$ describe how the system behaves.

It is quite easy to see that the Mandelbrot sets are a subclass of the above system definition. Hence we have a simple natural extension into three dimensions. Interestingly enough, it was found that systems such as

$$\begin{aligned} x_1(k+1) &= e^{\sin(x_1(k))*x_1(k)} \\ x_2(k+1) &= e^{\sin(x_2(k))*x_2(k)} \\ x_3(k+1) &= e^{\sin(x_3(k))*x_3(k)} \end{aligned} \tag{3}$$

and other similar equations did indeed create Mandelbrot like subsets filling three space.

2.2 Continuous Time

The continuous time story is similar to discrete time, except we have the defining equation as

$$\dot{x} = f(x) + \sum_{i=1}^{m} g_i(x)u_i \tag{4}$$

we may broaden this class to include systems which are not affine in the control input u such as:

$$\dot{x} = f(x, u) \tag{5}$$

where we now have $u \in \mathbf{R}^m$.

The evaluation of (5) is not as straight forward as (2), and requires numerical integration to achieve a solution for x. Simple iteration would achieve a solution for x_k in the discrete time case.

The inherent complexity in integrating differential equations causes the computation time to naturally rise and also leads to problems under certain conditions. One in particular is the integration of *stiff* systems.

Definition 1 Stiff System
A differential equation is said to be stiff *if for some* i, j $\|\dot{x}_i\| \gg \|\dot{x}_j\|$.

A typical method of integrating stiff systems is to us the *slow* variable $(\dot{x}_j)$ as a parameter in the equation for the fast dynamics $(\dot{x}_i)$. Then one updates the fast dynamics at one rate and updates the slow dynamics at a slower rate.

We use the *Lsoda* ordinary differential integration package developed at the Lawrence Livermore Labs (see [Hindmarsh, 1983, Petzold, 1983]). This package has automatic detection of stiff systems and switches integration algorithms accordingly. Hence we avoid the above problem.

2.3 Description of Dynamics

The user may describe the dynamics of his system by a simple *C* program which defines the state equations. A template is given below.

```
usr(init, x, xd, t, neq)

int
          init;

double
          x[], xd[], t;

int
          neq;

{

register
          i;

          if (init)
            {
          / put initialization code here          /
            }

          / compute xd here          /

          return;
}
```

The routine has five arguments. The first (*init*) is a flag variable that is *true* if it is the first call to the routine and allows the subroutine to initialize any variables. The second argument is the state vector x, which contains the current state of the system. xd is equivalent to $\dot{x}$ in (5) and t is the current time. Finally *neq* is the number of equations (currently this is fixed at three).

In summary the user is supplied with the current state, simulation time, and number of state equations and then must generate $\dot{x}$.

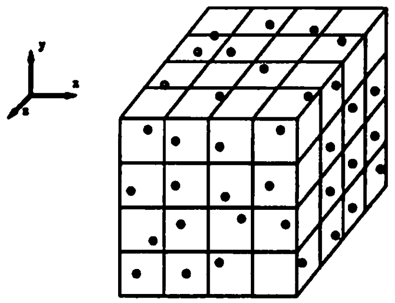
Figure 2: Gridding of the State Space

3 Computational aspects

The computation of the dynamics in three space can become quit large very fast since we essentially grid up a cube of specified size (see figure 2) and compute the characteristics of each subcube based on the characteristics for a random point within the subcube (the necessity for the randomness will be explained later). Since the calculation of the dynamics in each subcube depends only on itself we can carry on the computations in parallel.

With this in mind, a graphics server/computational client scheme was set up to offload the calculations. This also alleviated the need to write a full blown parser to gather the information necessary to define the system equations since some prewritten utilities could be used which would dynamically link in a subroutine.

These utilities exist for Sun workstations hence the logical choice of using the Sun as the computational server was made. The inherent graphics power of the Silicon Graphics series of workstation made it the obvious choice as the graphical server. This in turn talks to the various Sun computational clients using BSD sockets with the TCP/IP protocol over ethernet to disseminate information.

The steps to compute the stability characteristics may be listed as follows:

1. Server initiates communication to each client (max of 64).
2. Server downloads (to each client) a source file which describes the dynamics of the system.
3. Each client compiles and dynamically links in the routine and is ready to compute.

4. Server grids up the state space and divides it into *equal* portions for each client.

5. Clients integrate all initial conditions and assign each with a number representing if a point is stable or unstable. If the point is unstable it is also assigned a number which represents how fast the trajectory is moving away from the initial condition.

6. Clients report back with results.

Note: if we are only changing the grid size/shape parameters and not the subroutine describing the system, then only steps 4 and 5 need be repeated.

One may argue that a better way to compute the region of attraction would be to start a cluster of points about an equilibrium point and integrate backwards in time. One must continually add points to fill in three space while the trajectories are dispersing. Then the boundary of the trajectories will eventually form the region of attraction. We feel that this approach may be somewhat faster but would also be more ill conditioned and easily *fooled.*

The problem of determining if a point is stable or not is not particularly well defined. For example, imagine a trajectory which initially leaves the region around the initial condition and then slowly comes back in. Currently *Sys_View* integrates the system for a user specified time and then checks to see if the trajectory at the end time is closer to the center point; the center point is a user defined point which defines the point to compute the stability characteristics about – the center point is assumed to be an equilibrium point. Thus the above problem may be alleviated if the end time was made large, but computational speed suffers as the end time is increased.

It would be easy to have *Sys_View* allow the user define the criterion for stability by having a *C* routine dynamically linked in to return whether a point is stable or not. Furthermore, the user could be allowed to increase the end time dynamically to alleviate the above problem in a more intelligent way.

3.1 Server-Client Communication

A simple language had to be defined for communication between the server and each client. These include commands to download code, receive a set of points and send back the results. Error and warning commands also had to be incorporated along with another set of commands to handle the computation of surfaces.

Extra effort was made to insure the connections be as robust as possible. Communication takes place asynchronously with a maximum latency response to a message of 1.5 seconds and is carried through at a maximum rate of 1 Mbit/sec. If a client for some reason fails, the server is notified and redistributes the computation while closing the connection to the defunct client. In addition, if a client does not receive a command for some period of time (currently 30 minutes), the client reports to the server and terminates itself. This prevents *rogue* processes from running indefinitely. Furthermore, a client does not use any resources while it is waiting for a new set of computations.

Dynamic load scheduling is also used. This basically distributes the computational load *equally* amongst the clients. Prior to a computation each client is asked to return the time it took to compute a benchmark problem. This benchmark is included in each client as a subroutine and does not have to be sent nor is it linked in dynamically. This computation typically takes a couple of seconds and based on the returned result the server then prorates the number of points to be integrated. Hence if a machine is either inherently slow or is currently heavily loaded then its work will be accordingly scaled back.

Thus, with this setup we gained the ability to do parallel computation and had a simple way to specify the system equations (i.e. the equations could be described in a simple C or FORTRAN subroutine). In the current version there can be up to sixty four computational clients that would receive the subroutine, dynamically link it in and wait for commands to act on and report back to the server. A pictorial description of the setup is given in figure 3

A benchmark problem which required the integration of over 14,000 initial conditions on a system which was stiff and included trigonometric functions in its dynamics was completed in less than a minute on ten Sparc Station 1's as computational clients.

Since computations are essentially done in parallel we get an N times speed up, with hardly any overhead, and we again get the advantage of using dynamic linking to pull in the system description in the form of a C or FORTRAN subroutine.

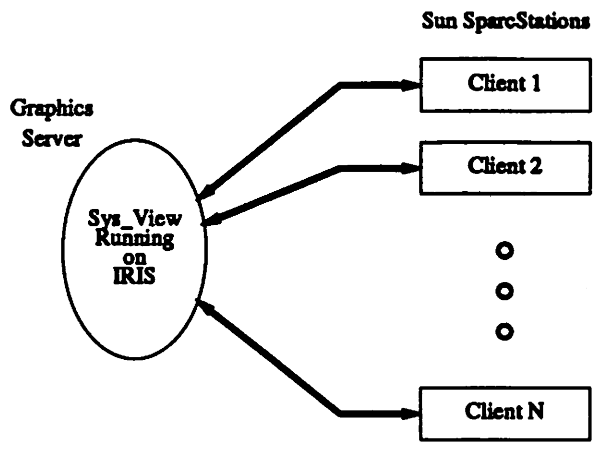

Figure 3: Graphics Server/Computational Client Set Up

4 Viewing the Results

One of the original ideas to view the data was to traverse along some axis and look at 2-D slices of the system (much like CAT scans). Multiple slices could be stored and in conjunction with transparency one could look past the initial slice to get more global information. This type of visualization may still be implemented but we feel that it would not be particularly easy to visually parse these images into a meaningful picture. that the global information lost would make this option not as attractive.

The route taken was to create a portrait cloud of points that was not fully dense (thus allowing us to see *through* things to view the behavior of the system at various points). This was accomplished by defining a random point constrained to be in a subcube of the gridded cube from figure 2. This allows us to see quite a bit of detail in the global aspect of the dynamics while still allowing us to view the local nature. Depth cueing is also available to give some depth perception in the visualization.

The randomness of the state space points was necessary to see the distant points. It should also be noted that it is much easier to view the data in an orthonormal projection then in a perspective projection.

A control panel was created as the main user interface (see figure 4). It allows the user to chose from the plethora of combinations given to view the data, and a complete description is given in the man pages for *sys-view*. A couple of features worth mentioning are the ability to record a sequence

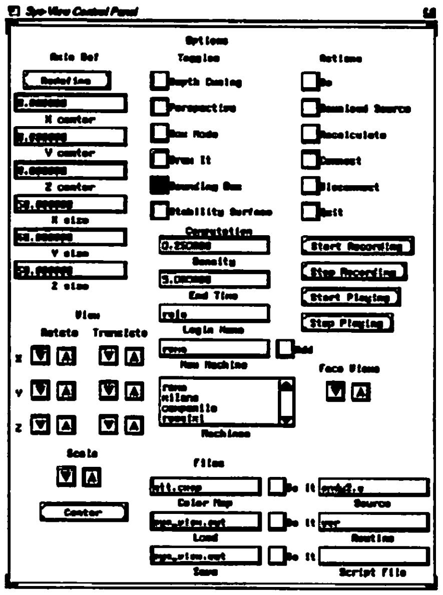

Figure 4: Sys-view Control Panel

of button clicks in the control panel and then play them back for a sort of movie to traverse the escape portrait and secondly the ability to define your own color map for the escape portrait. Along with the control panel a plotting window to view the data is opened (see figure 5 for a black and white version).

The region of attraction for a system is the locus of initial conditions whose trajectories steer towards the origin and is valuable piece of information to compute for a system. The region of attraction may be simply plotted by using a predefined color map which defines the 0^{th} entry to be white and all others to be black. Hence the points that do not leave the region are white while all points that do leave are black, thus giving us the region of attraction. One can play similar games for viewing all unstable points.

4.1 Creating Approximating Volume for the Region of Attraction

While the above solution for viewing the region attraction is quick, it is not the most visually pleasing. We would like to construct a a polyhedron which closely resembles the cloud of points.

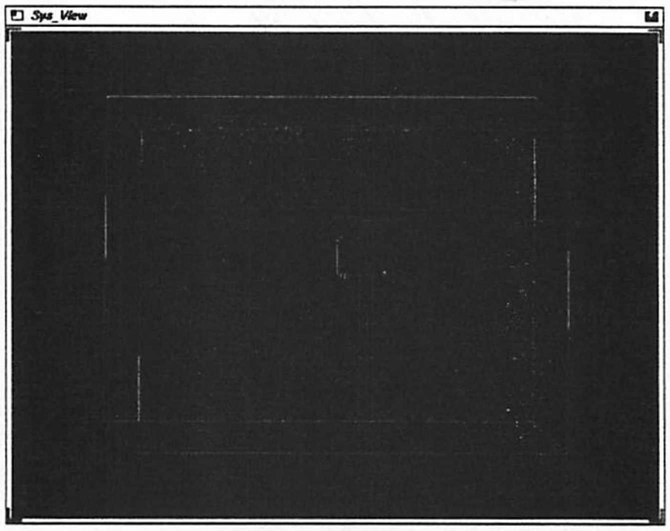

Figure 5: Sys-view Plotting Window

Interestingly enough this problem is very similar to one investigated in the post processing of CAT, MRI, and SPECT scans in the medical information field. The problem facing them is to reconstruct surfaces based on density slices made through the body.

A CAT scan creates a two dimensional slice through the body with each slice containing a grid of numbers representing how *dense* that particular point was. Thus if we wanted to reconstruct a face or the bone structure of an individual we simply focus on the points which have the desired density.

This is virtually the same problem we face with the construction of a polyhedron which represents the region of attraction. We have two dimensional slices which contain data representing stability of a point and we wish to focus in on only the stable points.

The approach taken by [Lorenson and Cline, 1987], which is the algorithm we choose, was the so called marching cubes algorithm. In short one basically looks at two slices of data at a time (see figure 6). We then restrict ourselves to look at eight vertices which form the so called *marching cube* as in figure 7. The three vertices define a face which constructs the local surface for the particular cube. We continue on repeating the process for each cube to construct the whole surface.

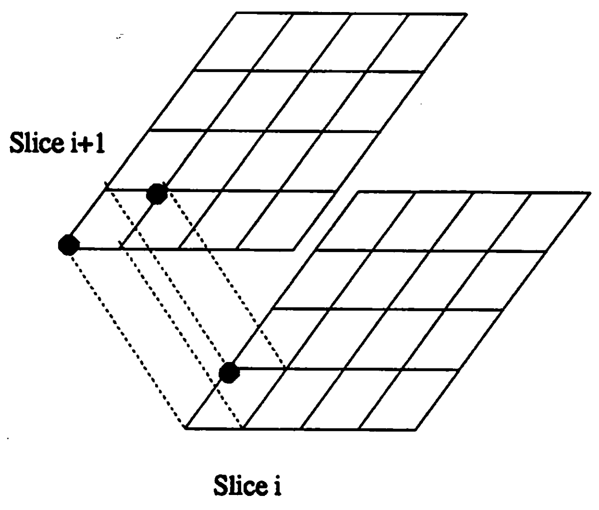

Figure 6: Slice view for Surface Reconstruction

It turns out that if one factors in rotational symmetry and symmetry due to vertex complements (ie. same configuration as case i except where there were points there are not and vice versa) there are only 14 unique configuration of the points within a cube. Thus we examine each cube in the data set and match it with one of the 14 basis cubes modulo rotational and complementary symmetry. Then retrieve the face structure for the basis cube and include it in the surface data structure with the rotation and complement operations applied.

It should be noted that these computations are carried out in a parallel fashion similar to the computation of the stability characteristics. The only drawback with this algorithm for our task at hand is that it may create an overabundance of triangles depending on the density of the point clouds. Alternatively, we do achieve an accurate approximation of the region of attraction.

5 Conclusions

A tool for viewing the dynamics of three dimensional continuous time systems has been developed in an interactive environment. With the addition of computational clients on remote machines the calculations necessary can be carried through relatively quickly. Possible additions for the future would be to include the CAT scan like slices mentioned above as another possible

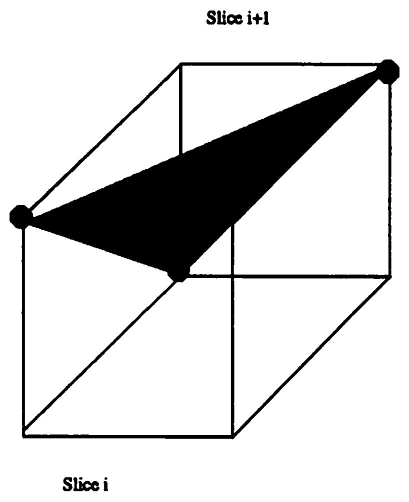

Figure 7: Marching Cube

viewing scheme. *Sys_View* combined with *AP_LIN* (see [Kadiyala, 1992]) provides a powerful set of utilities for controlling and viewing nonlinear dynamics.

References

[Hindmarsh, 1983] A. C. Hindmarsh. Odepack, a systematized collection of ode solvers. In R. S. Stepleman et al., editor, *Scientific Computing*, pages 55–64. North-Holland, Amsterdam, 1983.

[Kadiyala, 1992] R. R. Kadiyala. Ap_lin: A tool box for approximate linearization of nonlinear systems. In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design*, 1992.

[Lorenson and Cline, 1987] W. E. Lorenson and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Association of Computing Machinery, Computer Graphics*, 21, no. 4:163–169, 1987.

[Petzold, 1983] L. R. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *Siam Journal of Scientific Statistical Computing*, 4:136–148, 1983.

A Manual Pages for *Sys_View*

NAME

sys_view – Interactively view a three dimensional nonlinear dynamic system

SYNOPSIS

sys_view [-b *box mode*] [-c *color mapping*] [-d *cloud density*] [-h *hush mode*] [-i -j -k *position of bounding box*] [-l *login name to use on client*] [-m *remote machine*] [-o *output file*] [-p *use perspective projection*] [-q *turn on depth cueing*] [-r *routine*] [-s *source_file*] [-t *end time*] [-x -y -z *dimensions of bounding box*]

DESCRIPTION

sys_view creates a three dimensional stability portrait of a dynamical system described by the subroutine in *source_file*, which may be either C or FORTRAN and is called by the name in *routine*. **sys_view** creates connections up to the machines specified through repetitive uses of the -m option (i.e. sys_view -m mach1 -m mach2 ...) and then downloads the file to the clients. The clients then dynamically link in this routine and compute equal portions of the trajectories. The results are then reported back to the server machine and a plot is created which the user may interactively view. Up to 64 machines may be used as clients.

OPTIONS

–b go into box mode which will draw cubes instead of pixels at the grid point. This is useful if a low density is used.

–c color mapping; Use the file contained in *color mapping* to define the colors to use in the escape portrait. This should be an ASCII file with each line containing four integers ranging from 0 to 255 specifying the index and the standard red, green, and blue (rgb) color values. (i.e. 0 12 87 39 would specify entry 0 to have a red value of 12, a green value of 87, and a blue value of 39). Typically entry 0 will be black and entry 255 will be white.

–d density; Use floating point value *density* as the parameter defining how dense of a pixel cloud to use when generating the escape portrait; valid range is 0 to 1, the default is 0.25 (anything larger than 0.6 will take more than a few minutes to run).

–h go into hush mode which limits the number of messages sent to the window which sys_view was initiated in.

–i –j –k center of bounding box; Use these three numbers to define the origin of the bounding box. This is useful for studying behavior away from the origin; the default is zero.

–l login name; If the account name on the remote machine is different than on the local machine then this option must be set accordingly. The remote machine should allow entry of the local account through an entry in .rhosts.

–m machine; Specifies which machines to run the calculations of the trajectories on. For a low density it is not necessary to specify too many machines. In fact this may hurt you since the overhead involved in the communication may slow things down. For large densities it will be well worth the effort to spread the computing across as many machines as possible. A maximum of 64 machines may be specified in the following fashion: sys_view -m mach -m mach2 ...

–o output file; Use *output file* as the file to save to; the default is sys_view.out.

–p perspective projection; This allows the user to specify a perspective projection be used as opposed to the default orthonormal projection.

–q turn on depth cueing; This will allow the user to gain some depth perception as points further away are darker than closer points. This will, however, slow down the redrawing of the plot in interactive mode.

–r routine; Use *routine* to tell sys_view the name of the routine to call to execute the system equations.

–s source_file; Use *source_file* as the file to dynamically link in to get the system equations.

–t end time; Use the floating point value *end time* as the length to numerically integrate the system before determining the stability characteristics; the default is 2.0 seconds.

–x –y –z bounding box dimensions; Use these three numbers to define the boundary of the plot. If not specified sys_view tries to set them automatically.

USER INTERFACE

The user interface may be divided into two sections. The first being the control panel and the second being the interactive or plot window. The control panel is partitioned into seven groups. The characteristics of the bounding box may be defined in the *Axis Def* group, while various attributes may be toggled on and off in the *Toggles* group. These attributes include depth cueing, perspective projection, box/cube mode, draw it, turn on and off the bounding box, and create a surface approximation for the region of attraction. This last toggle will create another graphics window with the surface displayed. The third subgroup is the *Actions* group which allows the user to start calculations, download a new source file, disconnect, reconnect to the clients and to quit. Note that this is the only way to quit sys_view. The *Computation* group allows the user to specify the density and end time and also allows the addition of a new machine to the client list which is also displayed. It is suggested that a disconnect occur before the addition of a new client followed by a connect. The next subgroup is the *Scripting* group which allows the user to record a series of button clicks thus creating a movie to be played back. Typically one records the actions from the *View* group which allows the user to rotate, translate, and scale the portrait in a precise manner. The *Face Views* section is a simple pair of up-down buttons which allows the user to view all six of the viewing cube's faces in an easy manner. The final group is the *Files* group which allows the user to specify the load/save files and the source and routine names.

The plot window allows for interactive viewing of the portrait much like the *View* group above, but in a less precise but faster manner. The left mouse button controls scaling, while the middle mouse button controls rotations and the right mouse button controls translations. All operations are made with the given mouse button down and moving the mouse on the pad in the appropriate direction. Z axis rotation and translation may be obtained by holding the shift key down and performing the normal rotation or translation. Pressing the c key will recenter the portrait.

Example C Program

The following is an example C program which will show the format necessary to be linked in and run by the remote computational server. The routine takes five arguments with the first being *init* which is true for the initialization call (ie. if init = 1 then the routine should do any initialization it needs to do). The second argument is the state variable *x* which is of length *neq* (currently neq is three). The third argument is *xd* which is the return information for the system (ie. the user sets xd to the proper dynamics for the differential equation). The variable *t* represents the current simulation time.

```
#include <math.h>

usr(init, x, xd, t, neq)
int init, neq;
double *x, *xd;
{

        if (init)
         {
          Initialize code here
```

```
                ]

                **Compute dynamics here**
            }
```

AUTHOR

Raja R. Kadiyala, Dept. of EECS U.C. Berkeley. *email: raja@robotics.berkeley.edu*

BUGS

There is no error checking on the validity of the subroutine specified in source_file.

Sys_View : A Visualization Tool for Viewing the Regions of Validity and Attraction of Nonlinear Systems [*]

Raja R. Kadiyala

Department of Electrical Engineering
and Computer Science
207-59 Cory Hall
University of California
Berkeley, CA 94720
email: raja@robotics.berkeley.edu

March 1, 1992

Abstract

We present a visualization tool which allows one to view the stability characteristics of nonlinear ordinary differential equations in three dimensions. We find that these computations may be carried out in parallel and present an algorithm for multiple networked workstations. We also discuss various viewing alternatives for the visualization of these dynamics.

[*]Research supported in part by the Army under grants DAAL-88-K-0106 and DAAL-91-G-0191, and NASA under grant NAG 2-243

1 Introduction

So called phase portraits have been used to study the stability characteristics of planar two dimensional dynamical systems with a good deal of success, but we are truly limited by this visualization scheme since we have the restriction of two dimensions. We extend the concept of phase portraits to three dimensions with *Sys_View* .

Sys_View does not create three dimensional phase portraits per se. Rather, it allows one to view the stability characteristics of a three dimensional subset of the dynamics. By this we mean that one is able to see where a system is stable and hence the region of attraction and validity of a control law. *Sys_View* also allows one to view the characteristics of how a system is unstable. In short, *Sys_View* allows the user to interactively view the stability characteristics of a three dimensional system or a three dimensional submanifold of the system.

As one might imagine the task of computing the stability characteristics of a three dimensional system is a rather large computational undertaking. Fortunately we note that a majority of the calculations do not depend on each other, hence the algorithm is ideal for parallel computing. We will show such a procedure for this problem using multiple networked workstations to offload the computation.

We start with some simple definitions and an overview of the types of systems we can handle in section 2 and discuss the parallel algorithm in section 3, and lastly in section 4 we look at the visualization methods developed to survey all the information inherent in these plots.

2 Extension into Three Dimensions

Sys_View was motivated by a discrete time version which looked at extended two dimensional Mandelbrot sets into three dimensions. We first discuss the discrete time case.

2.1 Discrete Time

The defining equation for a Mandelbrot set is given by:

$$x_{k+1} = f(x_k) + R \tag{1}$$

where $x, R \in \mathbb{C}$. This gives rise to the familiar vanilla mandelbrot set shown in figure 1, where one plots all the points which converge (a discrete version

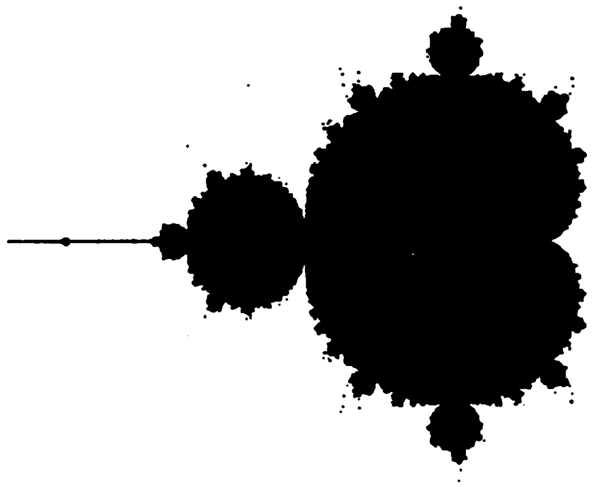

Figure 1: Unenhanced Mandelbrot Set

of the region of attraction).

These sets may be enhanced by color to give a more artistic flair and display more information about the system dynamics. A typical mapping is to assign each point a color according to how many iterations it takes to escape some finite area (the so called *escape time*).

Equation (1) is very similar to a general class of nonlinear discrete time systems which is affine in the control input. These systems may be described by:

$$x_{k+1} = f(x_k) + g(x_k)u_k \tag{2}$$

where

- x is known as the *state vector* ($\in \mathbf{R}^n$)
  - The state vector x describes the current configuration of the system.
- u is called the *control input* ($\in \mathbf{R}$)
  - The control input u is what you have control over to *move* the system.
- $f(\cdot)$ and $g_i(\cdot)$ describe how the system behaves.

It is quite easy to see that the Mandelbrot sets are a subclass of the above system definition. Hence we have a simple natural extension into three dimensions. Interestingly enough, it was found that systems such as

$$\begin{aligned} x_1(k+1) &= e^{\sin(x_1(k))*x_1(k)} \\ x_2(k+1) &= e^{\sin(x_2(k))*x_2(k)} \\ x_3(k+1) &= e^{\sin(x_3(k))*x_3(k)} \end{aligned} \tag{3}$$

and other similar equations did indeed create Mandelbrot like subsets filling three space.

2.2 Continuous Time

The continuous time story is similar to discrete time, except we have the defining equation as

$$\dot{x} = f(x) + \sum_{i=1}^{m} g_i(x)u_i \tag{4}$$

we may broaden this class to include systems which are not affine in the control input u such as:

$$\dot{x} = f(x, u) \tag{5}$$

where we now have $u \in \mathbf{R}^m$.

The evaluation of (5) is not as straight forward as (2), and requires numerical integration to achieve a solution for x. Simple iteration would achieve a solution for x_k in the discrete time case.

The inherent complexity in integrating differential equations causes the computation time to naturally rise and also leads to problems under certain conditions. One in particular is the integration of *stiff* systems.

Definition 1 Stiff System
A differential equation is said to be stiff *if for some i, j $\|\dot{x_i}\| \gg \|\dot{x_j}\|$.*

A typical method of integrating stiff systems is to us the *slow* variable $(\dot{x_j})$ as a parameter in the equation for the fast dynamics $(\dot{x_i})$. Then one updates the fast dynamics at one rate and updates the slow dynamics at a slower rate.

We use the *Lsoda* ordinary differential integration package developed at the Lawrence Livermore Labs (see [Hindmarsh, 1983, Petzold, 1983]). This package has automatic detection of stiff systems and switches integration algorithms accordingly. Hence we avoid the above problem.

2.3 Description of Dynamics

The user may describe the dynamics of his system by a simple *C* program which defines the state equations. A template is given below.

```
usr(init, x, xd, t, neq)

int
        init;

double
        x[], xd[], t;

int
        neq;

{

register
        i;

        if (init)
          {
        / put initialization code here        /
          }

        / compute xd here        /

        return;
}
```

The routine has five arguments. The first (*init*) is a flag variable that is *true* if it is the first call to the routine and allows the subroutine to initialize any variables. The second argument is the state vector x, which contains the current state of the system. xd is equivalent to $\dot{x}$ in (5) and t is the current time. Finally *neq* is the number of equations (currently this is fixed at three).

In summary the user is supplied with the current state, simulation time, and number of state equations and then must generate $\dot{x}$.

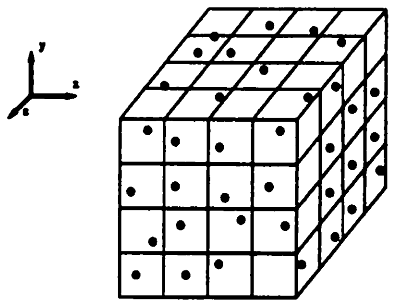

Figure 2: Gridding of the State Space

3 Computational aspects

The computation of the dynamics in three space can become quit large very fast since we essentially grid up a cube of specified size (see figure 2) and compute the characteristics of each subcube based on the characteristics for a random point within the subcube (the necessity for the randomness will be explained later). Since the calculation of the dynamics in each subcube depends only on itself we can carry on the computations in parallel.

With this in mind, a graphics server/computational client scheme was set up to offload the calculations. This also alleviated the need to write a full blown parser to gather the information necessary to define the system equations since some prewritten utilities could be used which would dynamically link in a subroutine.

These utilities exist for Sun workstations hence the logical choice of using the Sun as the computational server was made. The inherent graphics power of the Silicon Graphics series of workstation made it the obvious choice as the graphical server. This in turn talks to the various Sun computational clients using BSD sockets with the TCP/IP protocol over ethernet to disseminate information.

The steps to compute the stability characteristics may be listed as follows:

1. Server initiates communication to each client (max of 64).
2. Server downloads (to each client) a source file which describes the dynamics of the system.
3. Each client compiles and dynamically links in the routine and is ready to compute.

4. Server grids up the state space and divides it into *equal* portions for each client.

5. Clients integrate all initial conditions and assign each with a number representing if a point is stable or unstable. If the point is unstable it is also assigned a number which represents how fast the trajectory is moving away from the initial condition.

6. Clients report back with results.

Note: if we are only changing the grid size/shape parameters and not the subroutine describing the system, then only steps 4 and 5 need be repeated.

One may argue that a better way to compute the region of attraction would be to start a cluster of points about an equilibrium point and integrate backwards in time. One must continually add points to fill in three space while the trajectories are dispersing. Then the boundary of the trajectories will eventually form the region of attraction. We feel that this approach may be somewhat faster but would also be more ill conditioned and easily *fooled.*

The problem of determining if a point is stable or not is not particularly well defined. For example, imagine a trajectory which initially leaves the region around the initial condition and then slowly comes back in. Currently *Sys_View* integrates the system for a user specified time and then checks to see if the trajectory at the end time is closer to the center point; the center point is a user defined point which defines the point to compute the stability characteristics about – the center point is assumed to be an equilibrium point. Thus the above problem may be alleviated if the end time was made large, but computational speed suffers as the end time is increased.

It would be easy to have *Sys_View* allow the user define the criterion for stability by having a *C* routine dynamically linked in to return whether a point is stable or not. Furthermore, the user could be allowed to increase the end time dynamically to alleviate the above problem in a more intelligent way.

3.1 Server-Client Communication

A simple language had to be defined for communication between the server and each client. These include commands to download code, receive a set of points and send back the results. Error and warning commands also had to be incorporated along with another set of commands to handle the computation of surfaces.

Extra effort was made to insure the connections be as robust as possible. Communication takes place asynchronously with a maximum latency response to a message of 1.5 seconds and is carried through at a maximum rate of 1 Mbit/sec. If a client for some reason fails, the server is notified and redistributes the computation while closing the connection to the defunct client. In addition, if a client does not receive a command for some period of time (currently 30 minutes), the client reports to the server and terminates itself. This prevents *rogue* processes from running indefinitely. Furthermore, a client does not use any resources while it is waiting for a new set of computations.

Dynamic load scheduling is also used. This basically distributes the computational load *equally* amongst the clients. Prior to a computation each client is asked to return the time it took to compute a benchmark problem. This benchmark is included in each client as a subroutine and does not have to be sent nor is it linked in dynamically. This computation typically takes a couple of seconds and based on the returned result the server then prorates the number of points to be integrated. Hence if a machine is either inherently slow or is currently heavily loaded then its work will be accordingly scaled back.

Thus, with this setup we gained the ability to do parallel computation and had a simple way to specify the system equations (i.e. the equations could be described in a simple C or FORTRAN subroutine). In the current version there can be up to sixty four computational clients that would receive the subroutine, dynamically link it in and wait for commands to act on and report back to the server. A pictorial description of the setup is given in figure 3

A benchmark problem which required the integration of over 14,000 initial conditions on a system which was stiff and included trigonometric functions in its dynamics was completed in less than a minute on ten Sparc Station 1's as computational clients.

Since computations are essentially done in parallel we get an N times speed up, with hardly any overhead, and we again get the advantage of using dynamic linking to pull in the system description in the form of a C or FORTRAN subroutine.

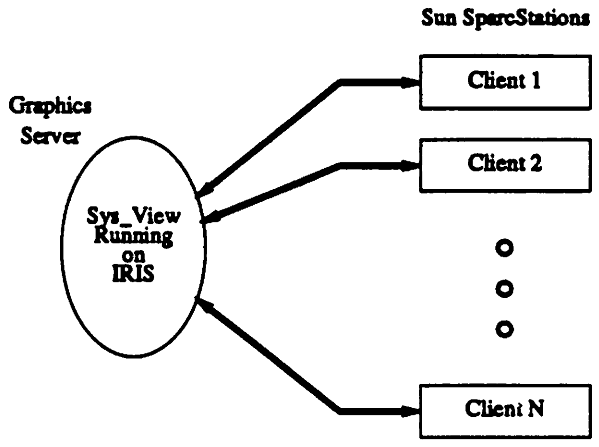

Figure 3: Graphics Server/Computational Client Set Up

4 Viewing the Results

One of the original ideas to view the data was to traverse along some axis and look at 2-D slices of the system (much like CAT scans). Multiple slices could be stored and in conjunction with transparency one could look past the initial slice to get more global information. This type of visualization may still be implemented but we feel that it would not be particularly easy to visually parse these images into a meaningful picture. that the global information lost would make this option not as attractive.

The route taken was to create a portrait cloud of points that was not fully dense (thus allowing us to see *through* things to view the behavior of the system at various points). This was accomplished by defining a random point constrained to be in a subcube of the gridded cube from figure 2. This allows us to see quite a bit of detail in the global aspect of the dynamics while still allowing us to view the local nature. Depth cueing is also available to give some depth perception in the visualization.

The randomness of the state space points was necessary to see the distant points. It should also be noted that it is much easier to view the data in an orthonormal projection then in a perspective projection.

A control panel was created as the main user interface (see figure 4). It allows the user to chose from the plethora of combinations given to view the data, and a complete description is given in the man pages for *sys-view*. A couple of features worth mentioning are the ability to record a sequence

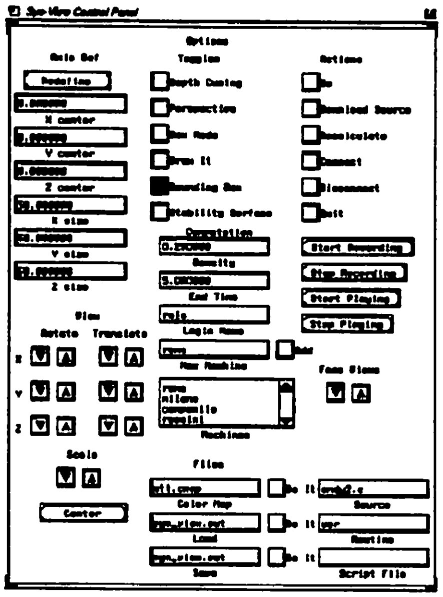

Figure 4: Sys-view Control Panel

of button clicks in the control panel and then play them back for a sort of movie to traverse the escape portrait and secondly the ability to define your own color map for the escape portrait. Along with the control panel a plotting window to view the data is opened (see figure 5 for a black and white version).

The region of attraction for a system is the locus of initial conditions whose trajectories steer towards the origin and is valuable piece of information to compute for a system. The region of attraction may be simply plotted by using a predefined color map which defines the 0^{th} entry to be white and all others to be black. Hence the points that do not leave the region are white while all points that do leave are black, thus giving us the region of attraction. One can play similar games for viewing all unstable points.

4.1 Creating Approximating Volume for the Region of Attraction

While the above solution for viewing the region attraction is quick, it is not the most visually pleasing. We would like to construct a a polyhedron which closely resembles the cloud of points.

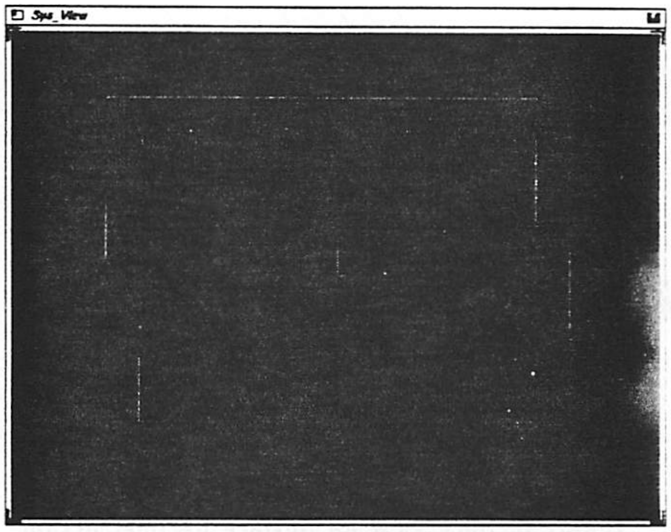

Figure 5: Sys-view Plotting Window

Interestingly enough this problem is very similar to one investigated in the post processing of CAT, MRI, and SPECT scans in the medical information field. The problem facing them is to reconstruct surfaces based on density slices made through the body.

A CAT scan creates a two dimensional slice through the body with each slice containing a grid of numbers representing how *dense* that particular point was. Thus if we wanted to reconstruct a face or the bone structure of an individual we simply focus on the points which have the desired density.

This is virtually the same problem we face with the construction of a polyhedron which represents the region of attraction. We have two dimensional slices which contain data representing stability of a point and we wish to focus in on only the stable points.

The approach taken by [Lorenson and Cline, 1987], which is the algorithm we choose, was the so called marching cubes algorithm. In short one basically looks at two slices of data at a time (see figure 6). We then restrict ourselves to look at eight vertices which form the so called *marching cube* as in figure 7. The three vertices define a face which constructs the local surface for the particular cube. We continue on repeating the process for each cube to construct the whole surface.

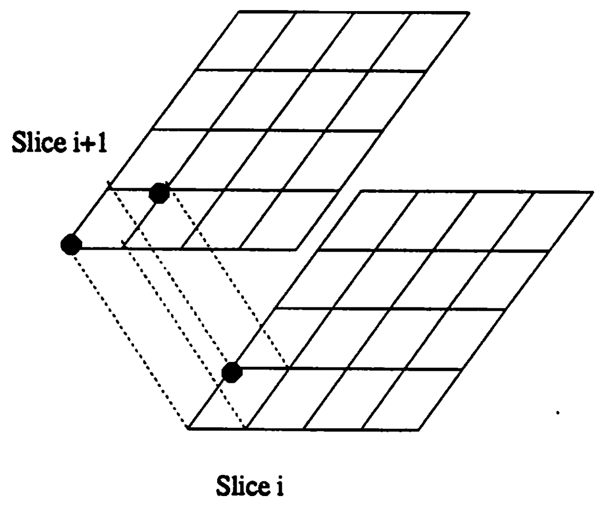

Figure 6: Slice view for Surface Reconstruction

It turns out that if one factors in rotational symmetry and symmetry due to vertex complements (ie. same configuration as case i except where there were points there are not and vice versa) there are only 14 unique configuration of the points within a cube. Thus we examine each cube in the data set and match it with one of the 14 basis cubes modulo rotational and complementary symmetry. Then retrieve the face structure for the basis cube and include it in the surface data structure with the rotation and complement operations applied.

It should be noted that these computations are carried out in a parallel fashion similar to the computation of the stability characteristics. The only drawback with this algorithm for our task at hand is that it may create an overabundance of triangles depending on the density of the point clouds. Alternatively, we do achieve an accurate approximation of the region of attraction.

5 Conclusions

A tool for viewing the dynamics of three dimensional continuous time systems has been developed in an interactive environment. With the addition of computational clients on remote machines the calculations necessary can be carried through relatively quickly. Possible additions for the future would be to include the CAT scan like slices mentioned above as another possible

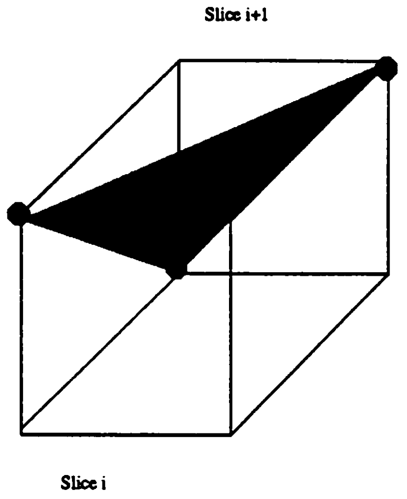

Figure 7: Marching Cube

viewing scheme. *Sys_View* combined with *AP_LIN* (see [Kadiyala, 1992]) provides a powerful set of utilities for controlling and viewing nonlinear dynamics.

References

[Hindmarsh, 1983] A. C. Hindmarsh. Odepack, a systematized collection of ode solvers. In R. S. Stepleman et al., editor, *Scientific Computing*, pages 55–64. North-Holland, Amsterdam, 1983.

[Kadiyala, 1992] R. R. Kadiyala. Ap_lin: A tool box for approximate linearization of nonlinear systems. In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design*, 1992.

[Lorenson and Cline, 1987] W. E. Lorenson and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Association of Computing Machinery, Computer Graphics*, 21, no. 4:163–169, 1987.

[Petzold, 1983] L. R. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *Siam Journal of Scientific Statistical Computing*, 4:136–148, 1983.

A Manual Pages for *Sys_View*

NAME

sys_view – Interactively view a three dimensional nonlinear dynamic system

SYNOPSIS

sys_view [-b *box mode*] [-c *color mapping*] [-d *cloud density*] [-h *hush mode*] [-i -j -k *position of bounding box*] [-l *login name to use on client*] [-m *remote machine*] [-o *output file*] [-p *use perspective projection*] [-q *turn on depth cueing*] [-r *routine*] [-s *source_file*] [-t *end time*] [-x -y -z *dimensions of bounding box*]

DESCRIPTION

sys_view creates a three dimensional stability portrait of a dynamical system described by the subroutine in *source_file*, which may be either C or FORTRAN and is called by the name in *routine*. **sys_view** creates connections up to the machines specified through repetitive uses of the -m option (i.e. sys_view -m mach1 -m mach2 ...) and then downloads the file to the clients. The clients then dynamically link in this routine and compute equal portions of the trajectories. The results are then reported back to the server machine and a plot is created which the user may interactively view. Up to 64 machines may be used as clients.

OPTIONS

–b go into box mode which will draw cubes instead of pixels at the grid point. This is useful if a low density is used.

–c color mapping; Use the file contained in *color mapping* to define the colors to use in the escape portrait. This should be an ASCII file with each line containing four integers ranging from 0 to 255 specifying the index and the standard red, green, and blue (rgb) color values. (i.e. 0 12 87 39 would specify entry 0 to have a red value of 12, a green value of 87, and a blue value of 39). Typically entry 0 will be black and entry 255 will be white.

–d density; Use floating point value *density* as the parameter defining how dense of a pixel cloud to use when generating the escape portrait; valid range is 0 to 1, the default is 0.25 (anything larger than 0.6 will take more than a few minutes to run).

–h go into hush mode which limits the number of messages sent to the window which sys_view was initiated in.

–i –j –k center of bounding box; Use these three numbers to define the origin of the bounding box. This is useful for studying behavior away from the origin; the default is zero.

–l login name; If the account name on the remote machine is different than on the local machine then this option must be set accordingly. The remote machine should allow entry of the local account through an entry in .rhosts.

–m machine; Specifies which machines to run the calculations of the trajectories on. For a low density it is not necessary to specify too many machines. In fact this may hurt you since the overhead involved in the communication may slow things down. For large densities it will be well worth the effort to spread the computing across as many machines as possible. A maximum of 64 machines may be specified in the following fashion: sys_view -m mach -m mach2 ...

–o output file; Use *output file* as the file to save to; the default is sys_view.out.

–p perspective projection; This allows the user to specify a perspective projection be used as opposed to the default orthonormal projection.

–q turn on depth cueing; This will allow the user to gain some depth perception as points further away are darker than closer points. This will, however, slow down the redrawing of the plot in interactive mode.

–r routine; Use *routine* to tell sys_view the name of the routine to call to execute the system equations.

–s source_file; Use *source_file* as the file to dynamically link in to get the system equations.

–t end time; Use the floating point value *end time* as the length to numerically integrate the system before determining the stability characteristics; the default is 2.0 seconds.

–x –y –z bounding box dimensions; Use these three numbers to define the boundary of the plot. If not specified sys_view tries to set them automatically.

USER INTERFACE

The user interface may be divided into two sections. The first being the control panel and the second being the interactive or plot window. The control panel is partitioned into seven groups. The characteristics of the bounding box may be defined in the *Axis Def* group, while various attributes may be toggled on and off in the *Toggles* group. These attributes include depth cueing, perspective projection, box/cube mode, draw it, turn on and off the bounding box, and create a surface approximation for the region of attraction. This last toggle will create another graphics window with the surface displayed. The third subgroup is the *Actions* group which allows the user to start calculations, download a new source file, disconnect, reconnect to the clients and to quit. Note that this is the only way to quit sys_view. The *Computation* group allows the user to specify the density and end time and also allows the addition of a new machine to the client list which is also displayed. It is suggested that a disconnect occur before the addition of a new client followed by a connect. The next subgroup is the *Scripting* group which allows the user to record a series of button clicks thus creating a movie to be played back. Typically one records the actions from the *View* group which allows the user to rotate, translate, and scale the portrait in a precise manner. The *Face Views* section is a simple pair of up-down buttons which allows the user to view all six of the viewing cube's faces in an easy manner. The final group is the *Files* group which allows the user to specify the load/save files and the source and routine names.

The plot window allows for interactive viewing of the portrait much like the *View* group above, but in a less precise but faster manner. The left mouse button controls scaling, while the middle mouse button controls rotations and the right mouse button controls translations. All operations are made with the given mouse button down and moving the mouse on the pad in the appropriate direction. Z axis rotation and translation may be obtained by holding the shift key down and performing the normal rotation or translation. Pressing the c key will recenter the portrait.

Example C Program

The following is an example C program which will show the format necessary to be linked in and run by the remote computational server. The routine takes five arguments with the first being *init* which is true for the initialization call (ie. if init = 1 then the routine should do any initialization it needs to do). The second argument is the state variable *x* which is of length *neq* (currently neq is three). The third argument is *xd* which is the return information for the system (ie. the user sets xd to the proper dynamics for the differential equation). The variable *t* represents the current simulation time.

```
#include <math.h>

usr(init, x, xd, t, neq)
int init, neq;
double *x, *xd;
{

        if (init)
         {
          Initialize code here
```

```
                    ]

                   Compute dynamics here
              }
```

AUTHOR

Raja R. Kadiyala, Dept. of EECS U.C. Berkeley. *email: raja@robotics.berkeley.edu*

BUGS

There is no error checking on the validity of the subroutine specified in source_file.