4-30-2003

Automated Annotation of Heegaard Diagrams

Dmitry Mogilevsky '03
Illinois Wesleyan University

Automated Annotation of Heegaard Diagrams

Dmitry Mogilevsky
Dept. Of Computer Science
Illinois Wesleyan University

April 30, 2003

1 Abstract

An important tool for dealing with the geometric and algebraic problems in low-dimensional topology involves the decomposition of topological spaces of dimensions 2, 3 and 4 into discrete data structures which can be displayed in a two dimensional setting.

This data may be represented by "Heegaard Diagrams", geometric representations of the essential structure constants associated with the 3 and 4 dimensional spaces under consideration, by using 2 dimensional constructions.

To aid in studies involving spaces with more complicated structures, P.J. Kapitza has developed a computer program (i.e., *Handles*) to allow the construction of arbitrary manifold spaces using the procedures developed for these purposes[1].

Recent research within this area has resulted in diagrams which have a complexity beyond which computer display systems can be expected to reasonably return accurate information. In addition, the labelling of the resulting structure constants has also increased at an exponential rate, which requires a computational solution to properly address.

The goal of this project is to create a driver program (Handles) which will convert the data generated by the client program into a format suitable for viewing and publication.

Design characteristics to be included within the driver include

1. Representation of the generated diagram must be faithful to the original data.

2. The representation should be scalable in nature, i.e., it should be designed with the possibility of altering the original scale of the diagram quickly and with minimal effort.

3. Features such as color coding of objects and a mechanism to automatically add label constructions are to be provided by the driver.

4. The driver is also to be further customizable by means of a command-line interaction to allow the user to select additional options.

[1] *c.f.* J.Singer, *Three dimensional manifolds and the Heegaard diagrams*, Trans.Amer.Math.Soc. 35 (1933)

The project has two stages. The first stage will consist of constructing a translator for the output data which is passed to the routine by the parent process.

The second stage draws conclusions based on the first stage input. It provides automated path-labeling for every segment of every path in the diagram. One difficulty in this process is to determine suitable space for the labels so that the final diagram contains usable information. Algorithms to identify the available space for labels and font size are explored, in an effort to minimize overlap.

2 Introduction

A significant function of computers is to solve problems that require a large amount of simple and repetitive calculations and actions, making them too time consuming to be solved manually. This is particularly true in creation of Heegaard Diagrams, which are used to represent mathematical objects called (closed) 3-manifolds. One challenge is creating presentation-quality Heegaard diagrams that are scalable to media of different sizes (Kapitza). A larger challenge, however, is to correctly label the various elements of the diagram. This is an extremelly time consuming task when done by hand, as the diagram first has to be drawn in a vector-based graphical format[2], following which every element has to be labelled. However, since placing labels is a relatively easy task, it can be automated. For the purpose of obtaining the structure constants of the manifold, these paths are critical. Furthermore, they do not proceed in a linear order.

3 Preliminary Concepts

3.1 About Manifolds

Heegaard diagrams, the desired goal of this project are used as two dimentional representation of manifolds. It is therefore important to understand what manifolds are. In simple terms, manifolds are topological objects which

[2]The need for vector-based format is dictated by our desire for scalability. Vector-based formats are by their nature more scalable than bitmap or other non-vector graphical formats

closely resemble a line, a plane or generally, a space which is locally homeomorphic to a copy of Euclidean space. The term n-manifold means a manifold which locally resembles an n-dimensional space. It does not necessarily have to lie in the n-dimensional space. An example of this is a line twisted into a knot. The local area around any point of the line can be described using only one dimension - hence the object is a 1-manifold. However, the knot itself is in three dimensions. As a general rule, the most dimensions n-manifold can have is $2 * n + 1$. Simple objects like a sphere or a torus are all 2-manifolds, because any point on the surface can be locally described in 2 dimensional space, even though the object itself is 3-dimensional. But manifold also have more complex applications. For example, in the theory of General Relativity the physical space is thought to be a manifold (More specifically, that the spacetime continuum is a 4-manifold) (UCDavis). One outstanding conjecture in the area of manifold topology, the Poincare Conjecture states that a 3-manifold is the three dimensional sphere s^3 if certain properties are met. Should this prove to be false, it would affect the way we fundamentally think about the universe (Craggs 1).

Some examples of simple manifolds can be seen in figure 1, which shows a simple manifold with three handles, as well as figure 2 which shows the Mazur manifold which willl be used in future study (Kapitza).

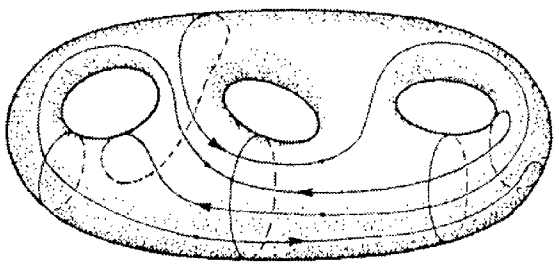

Figure 1: Simple Manifold with 3 handles

3.2 About Heegaard Diagrams

The 3-manifolds described previously can be studied thoroughly using diagrams of curves on surfaces called Heegaard Diagrams (Craggs 1). There is a correlation between Heegaard diagrams and manifolds that they represent. A

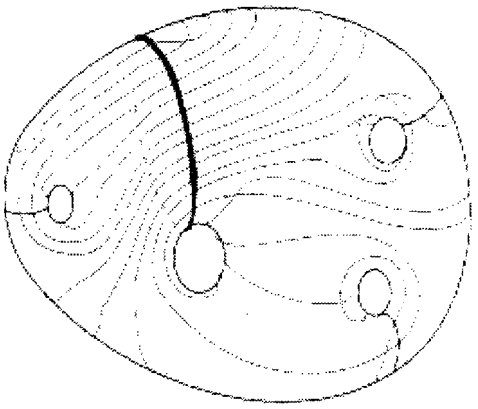

Figure 2: The Mazur Manifold

Heegaard diagram represents a unique manifold, however an infinite number of equivalent diagrams can be constructed from each manifold.

Heegaard diagrams are made up of a surface S, and several collections of curves $\{C_i\}$ and $\{D_i\}$. The curves in each collection are disjoint from each other, but they may cross back and forth over the curves from other collection. Examining these crossings reveal information about the manifolds represented by the diagrams.

In this project, the Heegaard diagrams will consist of two basic types of objects placed on a 2-dimensional plane. These objects are the "Paths" and the "Handles". Handles are represented on Heegaard Diagrams as pairs of triangles labeled "Positive (+)" and "Negative (-)". Handles can be of different sizes, but each handle must be of the same size as its pair. These handles represent the cut points of the manifold handles after they've been pushed into the plane (Kapitza).

Paths, which represent the curves on the manifold, are closed, noncurving lines on the 2-dimensional diagram. Paths intersect handles, creating path segments. A path that enters a handle will exit another handle in a manner prescribed by the original diagram construction. Because the paths are closed, the last segment of the path must end at the starting point of

the first segment. Lines are color coded into several color groups, each color group representing one collection of curves on the manifold. Figure 3 shows the Heegaard diagram of the Mazur manifold. Generating such diagrams is the purpose of this project.

3.2.1 Labelling Conventions

Every object on the Heegaard Diagram is identified with a serial number using the following conventions:

- Each pair of handles is identified by an integer starting with 1 and increasing by 1. Each handle in the pair is additionally labelled with either "+" or "-", to denote the direction traversed across the handle.
- Each path segment is identified by the string $k.l$ where k is the serial number of the path and l is the sequential number of the segment on the path.

3.3 The Goal

This project creates a command-line driven program that accepts a coordinate-encoding of a Heegaard Diagram and outputs the corresponding diagram in a scalable graphical format, fully labelled. Specific achievements of the program are enumerated below:

1. The Output accurately reflects the input information for arder and direction.
2. The Output is labelled to identify sequential segments in a path.
3. Output is created in a format that is easily scalable for display and publishing purposes.
4. The user can easily control the finished size of the resulting object.
5. The output is standard in that it can easily be integrated into papers and publications.

The following questions were considered:

1. Is the problem solvable in principle?

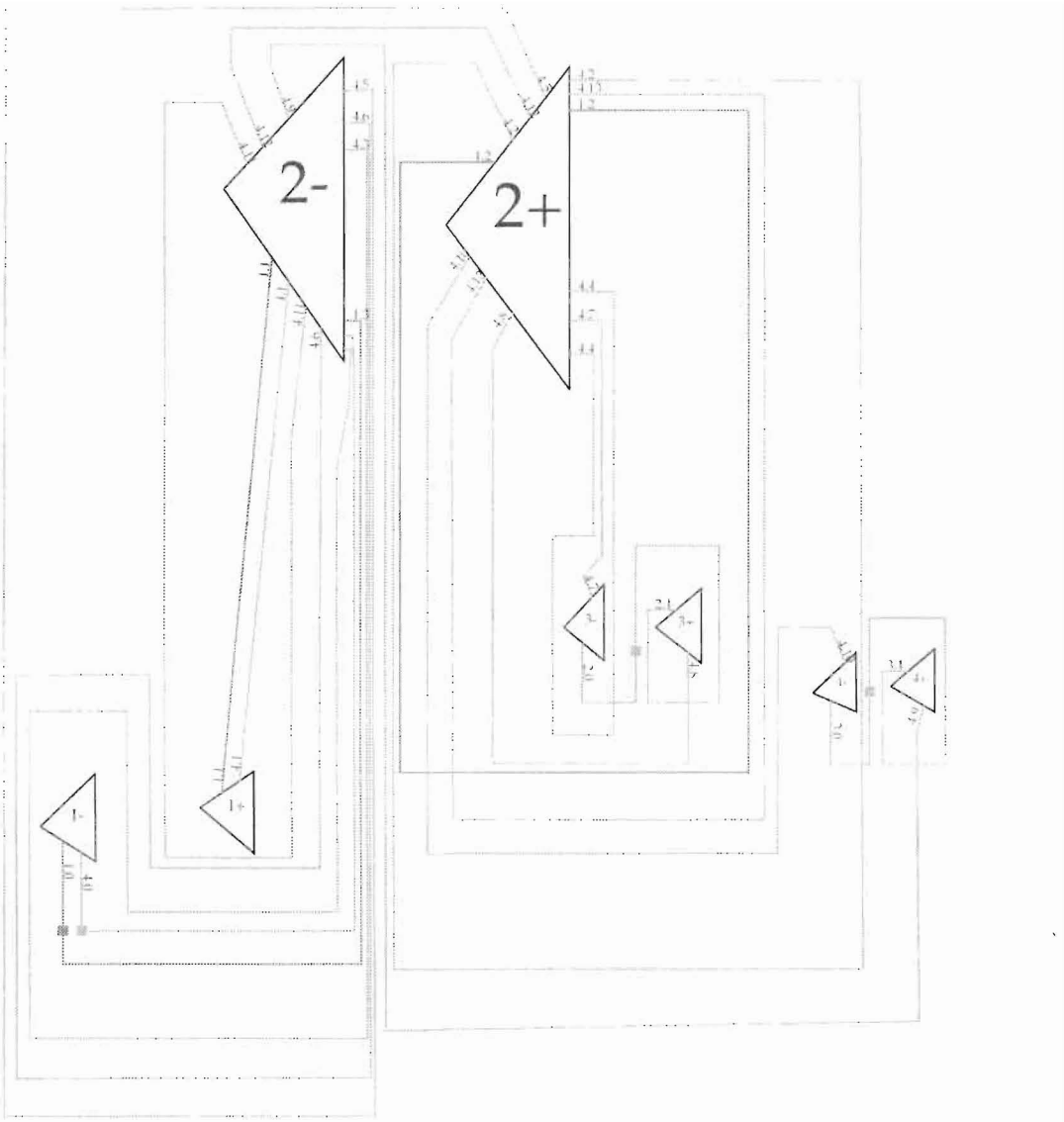

Figure 3: Heegaard Diagram of Mazur Manifold

2. What is the format of the input?

3. How will the user interact with the program?

4. What output format would be best suited for the purposes of this program?

The following four sections will address these topics.

3.4 Solvability

In order for it to be possible to create the required program, the problem that it attempts to solve must be solvable in polynomial time. The problem can be outlined with the following steps.

1. Read the Input File.

2. Identify various elements of the Heegard Diagram (Collectively, #1 and #2 are known as parsing).

3. Generate labels for the elements.

4. Output elements and labels in a vector-based graphical format.

It is easy to demonstrate that this problem is solvable in polynomial time. It will be proved further in the paper by providing a polynomial-time algorithm that solves the problem for finite input.

3.5 Input

A Heegaard diagram can be completely determined by the following information:

1. The number of Handles.

2. The number of paths.

3. The position of vertices.

4. The position of those vertices which occur on a "cut" handle.

The data that the program is to use is generated with software designed by Dr. Paul Kapitza. This software allows the user to create Heegaard diagrams using a graphical interface. The output file is an ASCII file which contains encoded properties of all the objects, such as their coordinates on the page, path colors and flags indicating whether a point is on a handle or not. It also contains a single integer value which indicates how many handle pairs are in the encoded diagram.

The ASCII encoding treats handles as paths of three segments. Hence the whole file is an encoding of point objects contained within path objects. Point objects represent each point where a path changes direction. A path is an envelope object that unites all points on it.

This ASCII encoding is to be used as input to the program being designed. The below sample is a segment from the file used to generate the Mazur diagram shown earlier in the paper:

```
1:P
2:16777215
  FALSE
  TRUE
3:X
4:   6.57031249999818E-0001
5:   7.36328125000000E-0001
6:FALSE
  0
  FALSE
  X
    6.82031250000364E-0001
    7.50000000000000E-0001
  FALSE
```

The symbol P on the line numbered 1 indicates the start of a new path. The integer value on line 2 is the RGB value of the color of the new path. Symbol X on line 3 indicates new point on the path. Lines 4 and 5 contain the coordinates of the point on the plane. Coordinates are given as floating point values in the interval [0.1], and indicate the relative position of the point relative to top-left point of the diagram. The boolean value on line 6 indicates whether the point is on a handle or not. Boolean values not marked

are not used by the driver. Line numbers are included here for clarification purposes only: they are not a part of the input file.

3.6 User Interaction

To allow the user to customize the resulting diagrams, the user is allowed to specify several options. Manipulation of these options will allow for generation of diagrams most suitable for their intended purpose. The user can manipulate the following options:

1. The size of the bounding box of the diagram in pixels (X and Y dimensions).
2. Range of label the text size.
3. Name of the output file.
4. Distance from the labels to handles.

The last item is used to fine tune performance on complex diagrams with a large number of paths.

3.7 Output

In choosing a format for the output of the program, three key factors were considered, namely:

1. The output should be in a graphical format that allows for easy scalability.
2. The output should be standard in that it can be readily inserted into papers and manuscripts.
3. The output format should allow easy encoding.

Based on these criteria, PostScript[3] was selected as the format of choice for the output of the program.

[3]Level 2 PostScript was used, even though the resulting code is compatible with older PostScript interpreters as well.

3.7.1 About Postscript

PostScript, developed by Adobe Systems, is a powerful graphics language. At its core is a graphical page description language, backed by a stack-based, interpreted programming language. Graphics created in PostScript are vector-based, and thus completely scalable. Furthermore, PostScript images are written using high-level, simple code, thus making it easy to generate PostScript graphics. PostScript is also highly portable, being compatible with a variety of platforms, including printers directly, most of which now come preequiped with a PostScript interpreter. Of particular usefullness to this project is a subset of PostScript entitled Encapsulated PostScript. Encapsulated PostScript works like regular PostScript, but it forces the image to be contained on one page. Because of its properties, Encapsulated PostScript images are easy to include in a body of text (McGilton).

4 Solution

The basic design of the program consists of reading the input file into a RAM-stored data structure which holds the diagram objects. Several calculations are then performed using the data to determine the optimal label size. The program then writes the output into a new file, simultaneously performing calculations to determine placement of the labels.

4.1 Data Structure

If handles are thought of as triangular paths, Heegaard diagrams are constructed entirely from paths. Each path, in turn, can be thought of as a collection of points connected by lines. This intuitive way of representing Heegaard diagrams is suggested by the structure of the input itself. A natural way to represent this relation is with a 2-dimensional array, where each "row" is a different path, and each element of the row is a point on that path. An alternative solution is to store the information in a linked list of linked lists. However, since neither the number of paths, nor their size changes after being read from the input file, the dynamic memory management capabilities of the linked list are not necessary. Thus it was decided to use the easier to implement array structure.

To easily differentiate between paths that are real paths and paths that represent handles, the data is stored in two separate arrays. The size of

HandleArray, which stores the handle paths is set as $2k \times 3$, where k is the number of handle pairs in the diagram. The size of PointArray, which stores the "regular" paths is $l \times m$, where l is the total number of paths in the diagram and m is the number of points in the largest path. The procedure to determine the values of l and m are described in the next section. Additionally, the integer array PointArraySize of size l stores the actual number of points in every path.

The type of these two arrays are yet to be mentioned. They are of type Point. Point is an object that stores a single point of the Heegaard diagram. Point object stores the coordinates of the point, as well as some flags pertaining to whether a particular point is located on a handle. It also contains methods to determine distance and angle between two points which are used by the program to perform label-placement calculations.

Another data organization possibility was to create a Path object that would initialize to contain the exact amount of points needed in that path, then store the Path objects in a one-dimensional array. While that solution would have been more memory efficient, it was decided against to keep the data structure simple.

4.2 Basic Algorithm Outline

The basic outline of the algorithm will proceed as follows:

- The output from the client program (in this case Handles) is read into the memory and stored in a two dimensional array.
- The path array is traversed to find most suitable text size for labels.
- Coordinates and PostScript code of the manifold curves are written to the output file.
- The path array is traversed again to determine where labels need to be placed.
- The angles at which the labels need to be rotated to be displayed correctly is determined.
- PostScript code for label display is written to the output file.

Each stage of the program employs at most 4 nested iterative loops. However, because the loops are used to scan a two dimensional array, it actually takes two nested loops to go through the entire set of points. Also, the algorithm does not use any recursive functions. Thus, if n is the number of the points in a processed diagram, no part of program is executed more than n^2 times. Therefore, the program runs in $O(n^2)$ polynomial time (Sipser).

4.3 Algorithm Specifics

4.3.1 Loading of Data

Scan of file is performed to determine what size array is necessary to store the information. Coordinates, path colors and boolean values are read from the input file and stored in a two dimensional array of Point objects. Each row of the array is a single path. Each instance of Point object is an edge point on the curve.

4.3.2 Font Size Calculations

The path array is traversed point by point. For each point, the program searches for other points on the same side of the same handle and determines the distance between the points. The minimum distance found is used to approximate the most appropriate font size for label display. Precedence is given to user specified size, if such exists.

4.3.3 Handle Drawing

The program traverses through the array of handles and determines absolute coordinates by multiplying the relative coordinates by the size of the diagram. The Coordinates of handle edges and PostScript code is written to the output file. The coordinates of the center of each handle is determined, and a handle label is placed there.

4.3.4 Path Drawing

The program traverses through every row of the path array. For every path, the path color is determined from the color array, and written to the output file. Every point on the path is traversed and its global coordinates are written to the output file along with PostScript code. Since every path is

closed, a final segment is then drawn between the last and the first point of the path.

4.3.5 Labeling

The path array is traversed again to examine every point. If a point is on a handle, availability of free space around it is checked by examining distances between intersection points on handles. If free space is available, a label is placed near that point. If the length of a segment between two points is larger than a certain constant value, and neither of its end points lie on a handle, a label is placed on its midpoint.

5 Results

5.1 Code

The code is attached as an appendix.

6 Conclusion

HandleDriver allows for efficient and accurate reproduction of Heegaard Diagrams in a form that is easily readable. It also significantly simplifies the process of labelling the paths of the diagram. The process of creating and labeling Heegaard diagrams manually takes a significantly longer time as compared to generating them with HandleDriver, without significant improvement in accuracy. For example, a complex Heegaard diagram which can be encoded and generated in a matter of several hours using this method, would take more than a month to be drawn by hand (Kapitza). Because the output files are created in PostScript graphical format, they are fully scalable.

7 Appendix - Code

7.1 Point.h

```
class Point{
 public:
  Point(); //Default Constructor
```

```
  Point(double x, double y, int b); //Secondary constructor.
   //x is the x-coordinate of the point
   //y is the y-coordinate of the point
   //b is a flag whether the point is on the handle.
  double getX(); //Returns x coordinate of the point.
  double getY(); //Returns y coordinate of the point.
  int isOnHandle(); //Returns 1 if point is on Handle, 0 if not.
  int whichHandle(); //Returns the number of the handle point is on.
  int whichSide();  //Returns which side of the handle the point is on.
  void setHandle(int handleArg, int sideArg); //Sets the Handle and Side
     //variables of the point
  void setX(double x); //Sets the x coordinate of the point
  void setY(double y); //Sets the y coordinate of the point
  void setIsOnHandle(int b); //Sets whether point is on the handle.
  double distanceToPoint(Point& P); // Returns distance to point P
  double angleToPoint(Point& P); //Returns angle of rotation to point P

 private:
  double coordx; //Stores x coordinate of the point.
  double coordy; //Stores y coordinate of the point.
  int onHandle; //Stores 1 if point is on handle, 0 if not.
  int handle; //Stores which handle the point is on.
  int side; //Stores which handle side the point is on.

};
```

7.2 Point.cc

```
#include "Point.h"
#include <iostream.h>
#include <math.h>

Point::Point():coordx(0),coordy(0)
{
}

Point::Point(double x, double y, int b):coordx(x),coordy(y),onHandle(b)
```

```
{
}

double Point::getX()
{
  return coordx;
}

double Point::getY()
{
  return coordy;
}

int Point::isOnHandle()
{
  return onHandle;
}

int Point::whichHandle()
{
  return handle;
}

int Point::whichSide()
{
  return side;
}

void Point::setHandle(int handleArg, int sideArg)
{
  handle = handleArg;
  side = sideArg;
}

void Point::setX(double x)
{
  coordx = x;
}
```

```
void Point::setY(double y)
{
  coordy = y;
}

void Point::setIsOnHandle(int b)
{
  onHandle = b;
}

double Point::distanceToPoint(Point& P)
{
  return sqrt((coordx-P.getX())*(coordx-P.getX())+(coordy-P.getY())
     *(coordy-P.getY()));
}

double Point::angleToPoint(Point& P)
{
  double angle=0;
  const double PI = 3.14159265358979323846264338327 95;
  if ((P.getY()==coordy)&&(P.getX()<coordx))
   return 0;
  if ((P.getY()==coordy)&&(P.getX()>coordx))
   return 0;
  if ((P.getY()-coordy)>=0&&(P.getX()-coordx)>0)
    angle = 0-atan((P.getY()-coordy)/(P.getX()-coordx));
  if ((P.getY()-coordy)>0&&(P.getX()-coordx)<=0)
    angle =(0-atan((P.getY()-coordy)/(P.getX()-coordx)));
  if((P.getY()-coordy)<=0&&(P.getX()-coordx)<0)
    angle = 0-atan((P.getY()-coordy)/(P.getX()-coordx));
  if ((P.getY()-coordy)<0&&(P.getX()-coordx)>=0)
    angle = (0-atan((P.getY()-coordy)/(P.getX()-coordx)));
  return angle/PI*180;
}
```

7.3 handle.cc

```
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "Point.cc"

//Default values for user variables
#define xdef 600
#define ydef 800
#define FALSE 0
#define TRUE 1
#define DEFAULT_MIN_TEXT 5
#define DEFAULT_MAX_TEXT 10


void getMax(ifstream& infile,int& pts, int& mxpoint);
//Returns the maximum value of points on a path - used to set up
//Path array

int isOnSpecificHandle(Point test, Point A, Point B);
//Tests whether a point is on a specific handle by testing
//whether test is between points A and B

int between(double t, double a, double b);
//helper function that tests if the number is between a and b.

void writeRGBColor(int color, double& red, double& green, double& blue);
//determines RGB color from 8 digit encoding

int H2D(char hex);
//Returns decimal value of a hexadecimal number

int H2D(char hex){
 switch (hex){
  case '0':
```

```
  return 0;
  break;
case '1':
  return 1;
  break;
case '2':
  return 2;
  break;
case '3':
  return 3;
  break;
case '4':
  return 4;
  break;
case '5':
  return 5;
  break;
case '6':
  return 6;
  break;
case '7':
  return 7;
  break;
case '8':
  return 8;
  break;
case '9':
  return 9;
  break;
case 'A':
  return 10;
  break;
case 'B':
  return 11;
  break;
case 'C':
  return 12;
  break;
```

```
  case 'D':
    return 13;
    break;
  case 'E':
    return 14;
    break;
  case 'F':
    return 15;
    break;
  }
}
void writeRGBColor(int color, double& red, double& green, double& blue)
{
  char hexstring[6];
  int charcounter=5;
  int sequentialcounter=5;
  int hexvalue;
  int hextest=1;
  int temp=color;
  while (sequentialcounter>=0){
    hexstring[sequentialcounter]='0';
    sequentialcounter--;
  }
  sequentialcounter = 5;
  while (temp!=0)
    {
      hextest = 1;
      charcounter = 5;
      while (hextest<temp){
hextest*=16;
charcounter--;
      }
      hextest/=16;
      charcounter++;
      while (sequentialcounter>charcounter){
hexstring[sequentialcounter]='0';
sequentialcounter--;
      }
```

```
      hexvalue = temp/hextest;
      temp = temp % (hextest*hexvalue);
      switch (hexvalue){
      case 0:
hexstring[charcounter]='0';
break;
      case 1:
hexstring[charcounter]='1';
break;
      case 2:
hexstring[charcounter]='2';
break;
      case 3:
hexstring[charcounter]='3';
break;
      case 4:
hexstring[charcounter]='4';
break;
      case 5:
hexstring[charcounter]='5';
break;
      case 6:
hexstring[charcounter]='6';
break;
      case 7:
hexstring[charcounter]='7';
break;
      case 8:
hexstring[charcounter]='8';
break;
      case 9:
hexstring[charcounter]='9';
break;
      case 10:
hexstring[charcounter]='A';
break;
      case 11:
```

```
hexstring[charcounter]='B';
break;
      case 12:
hexstring[charcounter]='C';
break;
      case 13:
hexstring[charcounter]='D';
break;
      case 14:
hexstring[charcounter]='E';
break;
      case 15:
hexstring[charcounter]='F';
break;
      }
 }
  red = (H2D(hexstring[0])*16+H2D(hexstring[1]))/256.0;
  green = (H2D(hexstring[2])*16+H2D(hexstring[3]))/256.0;
  blue = (H2D(hexstring[4])*16+H2D(hexstring[5]))/256.0;

}

int between(double t, double a, double b)
{
  if (a > b)
    return (t<=a&&t>=b);
  else if (a < b)
    return (t>=a&&t<=b);
  else
    return (t==a);
}

void getMax(ifstream& infile, int& pts, int& mxpoint){
  char temp;
  int count=0;
  infile.get(temp);
    cout <<"GetMax Entered\n";
  while (temp!='*'){
```

```
    switch (temp){
    case 'P':
      pts++;
      if (count>mxpoint)
mxpoint = count;
      cout << mxpoint;
      count=0;
      break;
    case 'X':
      count++;
      break;
    }
    if (count>mxpoint)
      mxpoint=count;
    infile.get(temp);
  }
}

int isOnSpecificHandle(Point test, Point A, Point B){
  double x = A.getX()-B.getX();
  double y = A.getY()-B.getY();
  double slope =y/x;
  double testy = slope*(test.getX()-B.getX())+B.getY();
  if ((testy-test.getY()<0.0001) && (testy-test.getY()>-0.0001))
    return (between(test.getY(), A.getY(), B.getY())&&
     between(test.getX(),A.getX(), B.getX()));
  else
    return FALSE;
}

int main(int argc, char *argv[]) {
 \\Initialization of variables
 int xbox = xdef; \\x-dimension of diagram
 int ybox = ydef; \\y-dimension of diagram
 int mintext = DEFAULT_MIN_TEXT; \\minimum text size
 int maxtext = DEFAULT_MAX_TEXT; \\maximum text size
 int textdistance = 5; \\default distance of text from handle
 int argcount = 1; \\argument counter
```

```
 int OutSet = 0;  \\Output file flag
 ofstream output; \\output stream

 \\Read command line flags and parameters
 while (argcount < (argc-1)){
  if (strcmp(argv[argcount],"-X")==0){ \\X size is set
   argcount++;
   xbox = atoi(argv[argcount]);
   argcount++;
  }
  else if (strcmp(argv[argcount],"-Y")==0){ \\Y size is set
   argcount++;
   ybox = atoi(argv[argcount]);
   argcount++;
  }
  else if (strcmp(argv[argcount],"-m")==0){ \\minimum text size set
   argcount++;
   mintext = atoi(argv[argcount]);
   argcount++;
  }
else if (strcmp(argv[argcount],"-M")==0){ \\maximum text size set
 argcount++;
 maxtext = atoi(argv[argcount]);
 argcount++;
}
else if (strcmp(argv[argcount],"-d")==0){ \\distance to handles set
 argcount++;
 textdistance = atoi(argv[argcount]);
 argcount++;
}
  else if (strcmp(argv[argcount],"-O")==0){ \\output file name set
   argcount++;
   output.open(argv[argcount]);
   OutSet = 1;
   argcount++;
  }
  else{
   cout << argv[argcount]<<endl;
```

```
 }
}
if (!OutSet)
 output.open("output.eps");
ifstream input(argv[argc - 1]); \\input file name read
int paths=0;
int max_points=0;
getMax(input,paths,max_points); \\ Determine array size
input.close();
input.open(argv[argc - 1]);
\\Write EPS file header
output << "%!PS-Adobe-3.0 EPSF-3.0\n";
output << "%%BoundingBox: 0 0 "<< xbox << " " <<  ybox << "\n";
output << "%%EndComments\n";
output << "/makex {"<<xbox<<"  mul} bind def\n";
output << "/makey {"<<0-ybox<<"  mul} bind def\n";
output << "/ln {2 1 roll makex 2 1 roll makey lineto} bind def\n";
output << "/mv {2 1 roll makex 2 1 roll makey moveto} bind def\n";
output << "/st {newpath var1 mv 3 3 rmoveto 0 -6
       rlineto -6 0 rlineto 0 6 rlineto closepath fill} bind def\n";
output << "%EndProlog\n";
output << "0 "<<ybox<<"  translate\n";
char circles,inchar;
int iterations=0;
input >> iterations; \\ Read number of handle pair
iterations *= 2; \\Total number of handle objects
paths -= iterations; \\total number of paths without handles.
\\Array declaration
Point pointGrid[paths][max_points]; \\Stores paths
int gridSize[paths]; \\Stores lengths of paths
int pathColors[paths]; \\Stores colors of paths
Point handleGrid[iterations][3]; \\Stores handles


int counter=0;
int handleCounter = 0; //Counts which handle is currently read
int pointOnHandle = 0; //Counts specific points on handle
int path =0;
```

```
int FLAG = FALSE; //Generally used to flag if a point is on a handle.
int curpath=-1; //Counts which path is currently read
int curpt=0; //Counts which point on path is currently read

\\Reads entire input file and stores it in arrays.
while(inchar!='*')
{
 input.get(inchar);
 switch(inchar){
  case 'X': \\Point found in file
  input.get(inchar);
  double cx;
double cy;
  input >> cx; //Read x coordinate
  input >> cy; //Read y coordinate

  if (handleCounter < iterations){ //Point is on a handle
   handleGrid[handleCounter][pointOnHandle].setX(cx);
   handleGrid[handleCounter][pointOnHandle].setY(cy);

   pointOnHandle++;
   if (pointOnHandle > 2){
     pointOnHandle = 0;
     handleCounter++;
   }//if
  }//if
  else{ //Point is on a path
   pointGrid[curpath][curpt].setX(cx);
   pointGrid[curpath][curpt].setY(cy);
   while (inchar!='T'&&inchar!='F') //Read if point is on handle
     input.get(inchar);
   if (inchar == 'T')
     pointGrid[curpath][curpt].setIsOnHandle(1);
   else
     pointGrid[curpath][curpt].setIsOnHandle(0);
   curpt++;
  }//else
  break;
```

```
  case 'P': \\Path object is found
   int color;
   if (handleCounter >= iterations){ \\Path is being read
    input >> color;
   pathColors[curpath+1]=color; \\Store color code in color array
   cout << color;
   if (curpath>=0)
     gridSize[curpath]=curpt; \\Store previous path length
   curpath++;
   curpt=0;
  }//if
 }//switch
}//while

gridSize[curpath]=curpt;
curpath++;



 //Traverses the array to determine on which handle and
 //handle side objects are and set the appropriate attributes
 //of the object
 for (int l=0; l < paths; l++){
  for (int m = 0; m < gridSize[l]; m++){
   if (pointGrid[l][m].isOnHandle()){
    //Determine which handles points are on
    for (int o = 0; o < iterations; o++){
     for (int p = 0; p < 3; p++){
       int nextValue = p+1;
       if (nextValue==3)
        nextValue = 0;
       if (isOnSpecificHandle(pointGrid[l][m],handleGrid[o][p],
         handleGrid[o][nextValue]))
       pointGrid[l][m].setHandle(o,p);
     }//for
    }//for
```

```
  }//if
 }//for
}//for

// Traverses the path array to
// determine minimum between-point distance
// to set the minimum text size
double min_length = 1000;
for (int i = 0; i < paths; i++){
 for (int k = 0; k < gridSize[i];k++){
  if (pointGrid[i][k].isOnHandle()){
   for (int l = 0; l < paths; l++){
    for (int m = 0; m < gridSize[l];m++){
     if (pointGrid[l][m].isOnHandle()){
      if((pointGrid[l][m].whichHandle()==pointGrid[i][k].whichHandle())&&
        (pointGrid[l][m].whichSide()==pointGrid[i][k].whichSide())&&
        (l!=i)&&(m!=k)){
          double dist = pointGrid[i][k].distanceToPoint(pointGrid[l][m]);
          if (dist < min_length)
          min_length = dist;
        }//if
       }//if
      }//for
     }//for
    }//if
   }//for
  }//for

int min_txt_size=(int)(min_length*ybox);
//Checks against user input values
if (min_txt_size<mintext)
  min_txt_size = mintext;
if (min_txt_size>maxtext)
  min_txt_size = maxtext;


//draw handles
char sign = '-';
```

```
for (int k=0;k<iterations; k++){
  output << " newpath\n";
  //Next four variables are used to calculate
  //the center point of each handle
  //and the text size to be used for it
  double accumx=0;
  double accumy=0;
  double min=handleGrid[k][1].getY();
  double max=handleGrid[k][1].getY();
  for (int l=0; l<3; l++){
    double cx = handleGrid[k][l].getX();
    accumx +=cx;
    double cy = handleGrid[k][l].getY();
    if (min > cy)
  min = cy;
  if (max < cy)
    max = cy;
    accumy +=cy;
    if (l == 0){
     //PostScript code is written to output file
     //for the first point of the handle
     output<<"\n/var1 {";
     output << cx;
     output << " ";
     output << cy;
     output << "} bind def\n";
     output << cx;
     output << " ";
     output << cy;
     output << " mv\n";
    }//if
    else {
     //PostScript code for other points
 output << cx;
 output << " ";
 output << cy;
 output << " ln\n";
    }//else
```

```
   }//for

   //PostScript code for the handle label is determined
   //and written to a file.
   output << "var1 ln stroke \n";
   output<< "/Palatino-Roman findfont \n "<< (max-min)/5*ybox<<
       " scalefont\n setfont\n";
   output<< accumx/3<<" "<<accumy/3<<" mv\n 0 (" <<k/2 + 1<<sign<<
       ")stringwidth pop 2 div  sub 0 rmoveto\n";
   output << "(" <<k/2 + 1<<sign<<") show\n";
   if (sign == '+')
    sign = '-';
   else
    sign = '+';
 }//for

output << "%% END HANDLE DRAWING \n";

//These three variables are used to store color information
double red,green,blue;

//Path drawing begins
for (int i=0;i<paths;i++)
{
 if (i > 0)
  output << "var1 ln stroke newpath st\n";
 else
  output << "newpath\n";
  //Determines path color and writes the color setting
  //PostScript commands to the file
  writeRGBColor(pathColors[i],red,green,blue);
  output << red << " " << green << " " << blue << " setrgbcolor\n";

  for (int j=0;j<gridSize[i];j++){
   double cx = pointGrid[i][j].getX();
   double cy = pointGrid[i][j].getY();
   //Writes PostScript code for the first point of the path.
   if (j == 0){
```

```
 output<<"\n/var1 {";
 output << cx;
 output << " ";
 output << cy;
 output << "} bind def\n";
 output << cx;
 output << " ";
 output << cy;
 output << " mv\n";
}//if
//Writes PostScript code for all other points on the path
else {
 output << cx;
 output << " ";
 output << cy;
 //If previous point does not enter a handle,
 //checks if current point enters a handle
 if (FLAG != TRUE){
  if (pointGrid[i][j].isOnHandle() == TRUE){
  FLAG = TRUE;
  output << " ln\n";
  }//if
  else{
   FLAG = FALSE;
   output << " ln\n";
  }//else
 }//if
 else{
 //Previous point does enter a handle
  int desPoint;
  if (j!= gridSize[i]-1)
   desPoint = j+1;
  else
   desPoint = 0;
  output << " mv\n";
  FLAG = FALSE;
 }//else
}//else
```

```
 }//for
}//for


//Label drawing segment begins
//Closes the last path and resets color back to black
output << " var1 ln stroke st \n";
output << "0 0 0 setrgbcolor \n";

//these two variables count the path and the path segment
//to be written as label output
int pathlabel = 1;
int segmentlabel = 0;

for (int i = 0; i < paths; i++){
 FLAG = FALSE;
 segmentlabel = 0;
 for (int j = 0; j < gridSize[i]; j++){
  //If a point is on the handle...
  if (pointGrid[i][j].isOnHandle() == TRUE){
   if (FLAG == FALSE){\\Previous point did not enter a handle.
    //PostScript cursor is moved to that point
    output << pointGrid[i][j].getX() << " "
           << pointGrid[i][j].getY() << " mv\n";
    output << "gsave\n";
    //Angle is determined
  double angle = pointGrid[i][j].angleToPoint(pointGrid[i][j-1]);
  //Canvas is rotated by that angle
  output << angle << " rotate\n";
  //Cursor is moved to the precise location the label should
  //be at.
if (pointGrid[i][j].getX() < pointGrid[i][j-1].getX())
     output << textdistance ;
    else{
    if ((pointGrid[i][j].getX()==pointGrid[i][j-1].getX())&&
       (pointGrid[i][j].getY()<pointGrid[i][j-1].getY()))
       output << 2.5*textdistance << " ";
    else
```

```
       output << 0-2.5*textdistance << "  ";
      }//else
      output << " 1 rmoveto\n";
      double min_dis=ybox;
      //Neighbourhood is checked to see whether there aren't any
      //other points on the same handle side near by.
      for (int y = 0; y < paths; y++){
       for (int z = 0; z < gridSize[y]; z++){
        if (!((y==i)&&(z==j))){
         if ((pointGrid[y][z].isOnHandle()==1)&&
          (pointGrid[y][z].whichHandle()==pointGrid[i][j].whichHandle())&&
          (pointGrid[y][z].whichSide()==pointGrid[i][j].whichSide())){
          if ((pointGrid[i][j].distanceToPoint(pointGrid[y][z])*ybox)<min_dis)
           min_dis = pointGrid[i][j].distanceToPoint(pointGrid[y][z])*ybox;

         }//if
        }//if
       }//for
      }//for

      //If there's enough space the label is drawn
      if (min_dis > min_txt_size)
       output << "/Palatino-Roman findfont \n"<<min_txt_size
         <<"  scalefont\n setfont\n  (" <<pathlabel<<"."<<segmentlabel
         <<") show\n";
      output << "grestore\n";
      segmentlabel++;
      FLAG = TRUE;
     }//if
//Previous point did enter a handle
//Same process is repeated, but the next point is checked
//To see whether its not the starting point.
     else{
      int desPoint;
      if (j!= gridSize[i]-1)
       desPoint = j+1;
      else
       desPoint = 0;
```

```
 output << pointGrid[i][j].getX() << " "
  << pointGrid[i][j].getY() <<" mv\n";
 output << "gsave \n";
 double angle = pointGrid[i][j].angleToPoint(pointGrid[i][desPoint]);
 output << angle << " rotate\n";
 if (pointGrid[i][j].getX() < pointGrid[i][desPoint].getX())
  output << textdistance ;
 else{
 if ((pointGrid[i][j].getX()==pointGrid[i][desPoint].getX())&&
    (pointGrid[i][j].getY()<pointGrid[i][desPoint].getY()))
  output << 2.5*textdistance << " ";
 else
  output << 0-2.5*textdistance << " ";
 }\\else
 output << " 1 rmoveto\n";
 double min_dis=ybox;
 for (int y = 0; y < paths; y++){
  for (int z = 0; z < gridSize[y]; z++){
   if (!((y==i)&&(z==j))){
    if ((pointGrid[y][z].isOnHandle()==1)&&
     (pointGrid[y][z].whichHandle()==pointGrid[i][j].whichHandle())&&
     (pointGrid[y][z].whichSide()==pointGrid[i][j].whichSide())){
     if ((pointGrid[i][j].distanceToPoint(pointGrid[y][z])*ybox)<min_dis)
      min_dis = pointGrid[i][j].distanceToPoint(pointGrid[y][z])*ybox;
    }//if
   }//for
  }//for
 }//else
 if (min_dis > min_txt_size)
  output<< "/Palatino-Roman findfont \n"<<min_txt_size
   <<"  scalefont\n setfont\n  (" <<pathlabel<<"."<<segmentlabel
   <<") show\n";
 output << "grestore\n";
 FLAG = FALSE;
 }
}//if
//If a point is not on a handle
//Check whether the line between two points is over a certain length
```

```
    else{
     int desPoint;
     if (j!= gridSize[i]-1)
      desPoint = j+1;
     else
      desPoint = 0;
     //Check that the next point is not on the handle as well
     if (pointGrid[i][desPoint].isOnHandle()==FALSE){
      //Check that the line is sufficiently long
      if (pointGrid[i][j].distanceToPoint(pointGrid[i][desPoint])>0.25){
       //if it is, place a label in the middle of it.
       output << "gsave \n";
       //Middle point is determined
       output << (pointGrid[i][j].getX()+pointGrid[i][desPoint].getX())/2
              << " ";
       output << (pointGrid[i][j].getY()+
            pointGrid[i][desPoint].getY())/2 <<" mv\n";
       //angle is determined
       double angle = pointGrid[i][j].angleToPoint(pointGrid[i][desPoint]);
       output << angle << " rotate\n";
       output << "0 1 rmoveto";
       //label output is written
       output<< "/Palatino-Roman findfont \n"<<min_txt_size
        <<"  scalefont\n setfont\n  (" <<pathlabel<<"."<<segmentlabel
        <<") show\n";
       output << "grestore\n";
      }//if
     }//if
    }
   }
   pathlabel++;
  }
  //end of label drawing
   return 0;
}
```

8 Sample Heegaard Diagrams

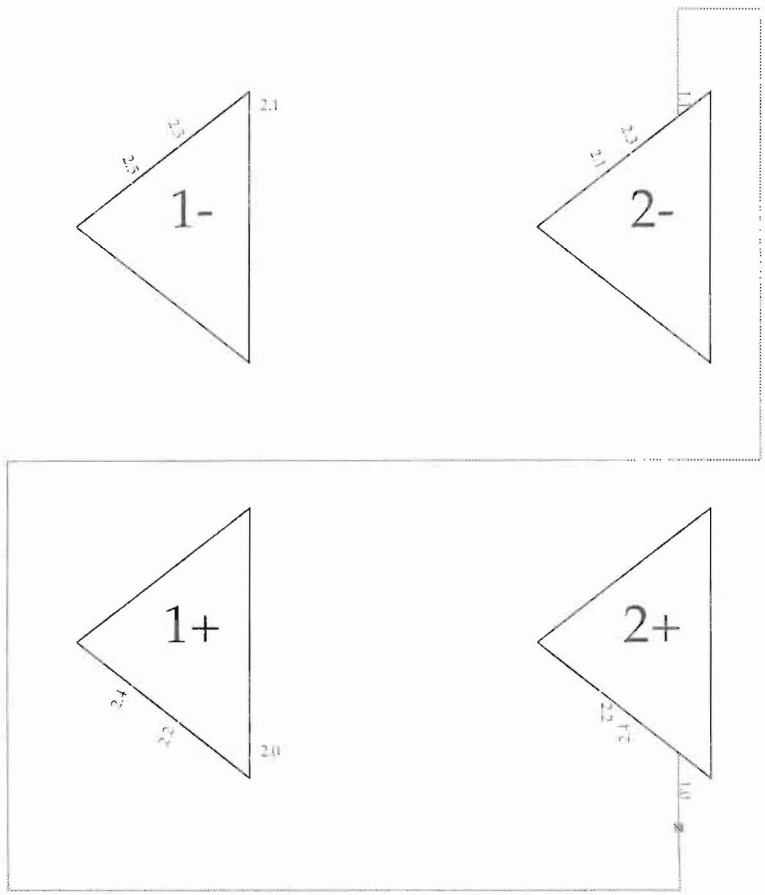

Figure 1: An example taken from a paper by Michael Boileau and Heiner Zieschang studying whether certain relations can be realized in a manifold space.

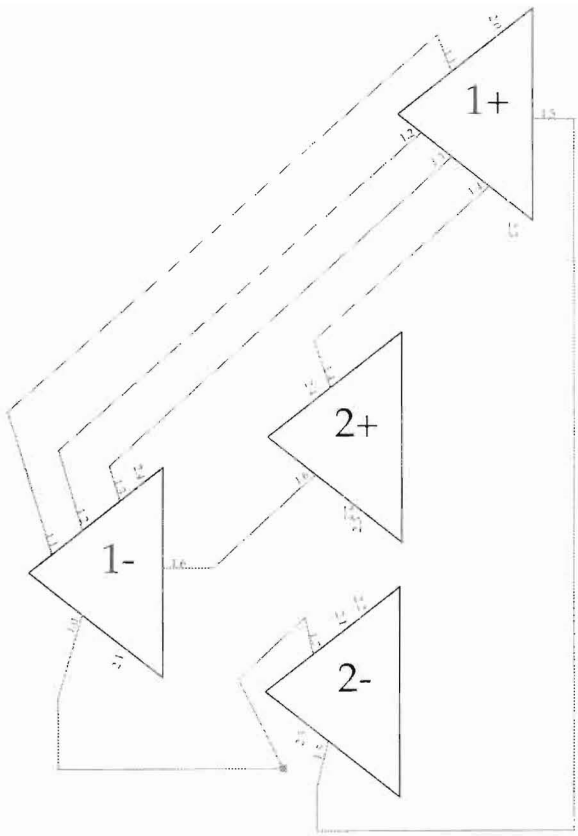

Figure 5: This has some historical significance in that it was produced by Henri Poincare as a counterexample to his earlier conjecture on how to recognize the 3-sphere. It is a manifold space whose fundamental group is called the binary icosahedral group, having 120 elements. It's significance is that it satisfies the conditions of Poincare's first attempt to recognize the 3-sphere (namely, it's homology groups are trivial), yet because of the fundamental group, cannot be the 3-sphere (it is known that the 3-sphere has only the trivial fundamental group). This particular manifold representation appeared in the same paper as the new conjecture Poincare made about recognizing the 3-sphere (THE Poincare conjecture). It is safe to say that this conjecture is one of THE fundamental questions to answer in topology.

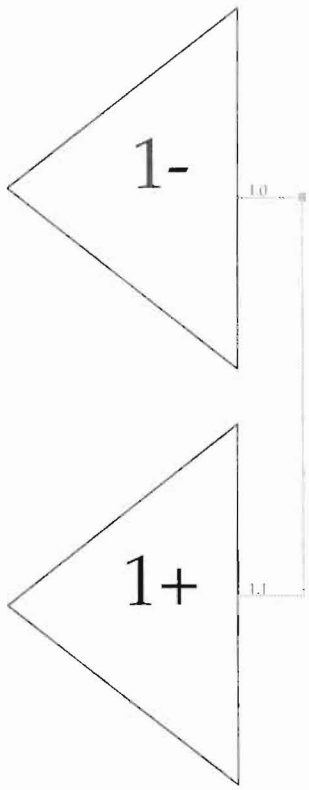

Figure 6: An example of the simplest Heegaard diagram one can build, that of the 3-sphere with 1 handle / 1 relator.

9 References

1. Craggs, Robert. "Visual Manipulation of Curve Diagrams on Surfaces". Research Proposal to AT&T. University of Illinois. Urbana. IL

2. Craggs. Robert. "On Doubled 3-Manifolds and Minimal Handle Presentations for 4-Manifolds". Paper. January 18th, 2000

3. Singer. James. "Three-Dimensional Manifolds and Their Heegaard Diagrams"'. Presented to AMS, October 29th. 1932.

4. McGilton. Henry. Campione, Mary. *PostScript By Example*. Addison-Wesley, Reading Massachusetts, 1992.

5. UC-Davis Dept. Of Mathematics. *Glossary*. online. http://www.math.ucdavis.edu/glossary.html

6. Sipser. Michael. *Introduction to Theory Of Computation*. PWS Publishing Company. Boston. MA. 1997

7. Kapitza. Paul. Emails and Conversations.