

Chapter 1

Nonlinear and Chaotic Maps

1.1 One-Dimensional Maps

In this section we consider nonlinear and chaotic one-dimensional maps

$$f : S \rightarrow S, \quad S \subset \mathbf{R}.$$

In most cases the set S will be $S = [0, 1]$ or $S = [-1, 1]$. The one-dimensional map can also be written as a difference equation

$$x_{t+1} = f(x_t), \quad t = 0, 1, 2, \dots \quad x_0 \in S.$$

Starting from an initial value $x_0 \in S$ we obtain, by iterating the map, the sequence

$$x_0, \quad x_1, \quad x_2, \quad \dots$$

or

$$x_0, \quad f(x_0), \quad f(f(x_0)), \quad f(f(f(x_0))), \quad \dots$$

For any $x_0 \in S$ the sequence of points x_0, x_1, x_2, \dots is called the forward *orbit* (or forward *trajectory*) generated by x_0 . The goal of a dynamical system is to understand the nature of all orbits, and to identify the set of orbits which are periodic, eventually periodic, asymptotic, etc. Thus we want to understand what happens if $t \rightarrow \infty$. In some cases the long-time behaviour is quite simple.

Example. Consider the map $f : \mathbf{R}^+ \rightarrow \mathbf{R}^+$ with $f(x) = \sqrt{x}$. For all $x_0 \in \mathbf{R}^+$ the forward trajectory tends to 1. ♣

Example. Consider the map $f : [0, 1] \rightarrow [0, 1]$, $f(x) = x^2$. If $x_0 = 0$, then $f(x_0) = 0$. Analogously, if $x_0 = 1$, then $f(x_0) = 1$. The points 0 and 1 we call fixed points for this map. For $x \in (0, 1)$, the forward trajectory tends to 0. ♣

In most cases the behaviour of a map is much more complex.

Next we introduce some basic definitions for dynamical systems (Devaney [25], Holmgren [53]).

Definition. A point $x^* \in S$ is called a *fixed point* of the map f if $f(x^*) = x^*$.

Example. Consider the map $f : [0, 1] \rightarrow [0, 1]$ with $f(x) = 4x(1 - x)$. Then $x^* = 3/4$ and $x^* = 0$ are fixed points. ♣

Definition. A point $x^* \in S$ is called a *periodic point* of period n if

$$f^{(n)}(x^*) = x^*$$

where $f^{(n)}$ denotes the n -th iterate of f . The least positive integer n for which $f^{(n)}(x^*) = x^*$ is called the *prime period* of x^* . The set of all iterates of a periodic point form a periodic orbit.

Example. Consider the map $f : \mathbf{R} \rightarrow \mathbf{R}$ and $f(x) = x^2 - 1$. Then the points 0 and -1 lie on a periodic orbit of period 2 since $f(0) = -1$ and $f(-1) = 0$. ♣

Definition. A point x^* is *eventually periodic* of period n if x^* is not periodic but there exists $m > 0$ such that

$$f^{(n+i)}(x^*) = f^{(i)}(x^*)$$

for all $i \geq m$. That is, $f^{(i)}(x^*)$ is periodic for $i \geq m$.

Example. Consider the map $f : \mathbf{R} \rightarrow \mathbf{R}$ with $f(x) = x^2 - 1$. Then with $x_0 = \sqrt{2}$ we have the orbit $x_1 = 1$, $x_2 = 0$, $x_3 = -1$, $x_4 = 0$, i.e., the orbit is eventually periodic. ♣

Definition. Let x^* be a periodic point of prime period n . The point x^* is *hyperbolic* if

$$|(f^{(n)})'(x^*)| \neq 1$$

where $'$ denotes the derivative of $f^{(n)}$ with respect to x .

Example. Consider the map $f_c : \mathbf{R} \rightarrow \mathbf{R}$ with $f_c(x) = x^2 + c$ and $c \in \mathbf{R}$. Then the fixed points are $x_{\pm}^* = 1/2 \pm \sqrt{1/4 - c}$. We have $f'_c \equiv df_c/dx = 2x$ and $df'_c(x_{\pm}^*)/dx = 1 + \sqrt{1 - 4c}$. With $c = 1/4$ we have $|f'_c(x_{\pm}^*)| = 1$ and the fixed point is non-hyperbolic. ♣

Theorem. Let x^* be a hyperbolic fixed point with $|f'(x^*)| < 1$. Then there is an open interval U about x^* such that if $x \in U$, then

$$\lim_{n \rightarrow \infty} f^{(n)}(x) = x^*.$$

In the following sections we also introduce the following concepts important in the study of nonlinear and chaotic maps. The concepts are

- 1) Fixed points
- 2) Liapunov exponent
- 3) Invariant density
- 4) Autocorrelation functions
- 5) Moments
- 6) Fourier transform
- 7) Bifurcation diagrams
- 8) Feigenbaum number
- 9) Symbolic dynamics
- 10) Chaotic repeller

The one-dimensional maps we study in our examples are the logistic map, the tent map, the Bernoulli map, the Gauss map, a bungalow-tent map and the circle map

1.1.1 Exact and Numerical Trajectories

In this section we calculate trajectories for one-dimensional maps. In our first example we consider the map $f : \mathbf{N} \rightarrow \mathbf{N}$ defined by

$$f(x) := \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3x + 1 & \text{if } x \text{ is odd} \end{cases}$$

where \mathbf{N} denotes the natural numbers. For this map it is conjectured that for all initial values the trajectory finally tends to the period orbit $\dots 4 \ 2 \ 1 \ 4 \ 2 \ 1 \ \dots$. The data type `unsigned long` (4 bytes) in C++ is restricted to the range

`0...4294967295`

and the data type `long` (4 bytes) in C++ is restricted to the range

`-2147483648...+2147483647`

To check the conjecture for larger initial values we use the abstract data type `Verylong` in `SymbolicC++`. In this class all arithmetic operators are overloaded. The overloaded operators are

`+, -, *, /, %, +=, -=, *=, /=` .

For the initial value 28 we find the sequence

14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, ...

Thus the orbit is eventually periodic. Two different initial values are considered in the program `trajec1.cpp`, namely 28 and 998123456789.

```

// trajectory1.cpp

#include <iostream>    // for cout
#include "verylong.h" // for data type Verylong of SymbolicC++
using namespace std;

int main(void)
{
    unsigned long y = 28; // initial value
    unsigned long T = 20; // number of iterations
    unsigned long t;
    for(t=0;t<T;t++)
    {
        if((y%2)==0) y = y/2;
        else y = 3*y+1;
        cout << y << endl;
    }
    Verylong x("998123456789"); // initial value
    Verylong zero("0"), one("1"), two("2"), three("3");
    T = 350;
    for(t=0;t<T;t++)
    {
        if((x%two)==zero) x = x/two;
        else x = three*x + one;
        cout << x << endl;
    }
    return 0;
}

```

Java provides a class `BigInteger`. Since operators such as `+`, `-`, `*`, `/`, `%` cannot be overloaded in Java, Java uses methods to do the arithmetic operations. The methods are

`add()`, `subtract()`, `multiply()`, `divide()`, `mod()` .

The constructor `BigInteger(String val)` translates the decimal `String` representation of a `BigInteger` into a `BigInteger`. The class `BigInteger` also provides the data fields `BigInteger.ONE` and `BigInteger.ZERO`.

```

// Trajectory1.java

import java.math.*;

public class Trajectory1
{
    public static void main(String[] args)
    {
        BigInteger X = new BigInteger("998123456789");
        BigInteger TWO = new BigInteger("2");
    }
}

```

```

BigInteger THREE = new BigInteger("3");
int T = 350;
for(int t=0;t<T;t++)
{
if((X.mod(TWO)).equals(BigInteger.ZERO))
X = X.divide(TWO);
else { X = X.multiply(THREE); X = X.add(BigInteger.ONE); }
System.out.println("X = " + X);
}
}
}

```

In our second example we consider the trajectories for the logistic map. The *logistic map* $f : [0, 1] \rightarrow [0, 1]$ is given by $f(x) = 4x(1 - x)$. The logistic map can also be written as the difference equation

$$x_{t+1} = 4x_t(1 - x_t)$$

where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1]$. It follows that $x_t \in [0, 1]$ for all $t \in \mathbf{N}$. Let $x_0 = 1/3$ be the initial value. Then we find that

$$x_1 = \frac{8}{9}, \quad x_2 = \frac{32}{81}, \quad x_3 = \frac{6272}{6561}, \quad x_4 = \frac{7250432}{43046721}, \quad \dots$$

The exact solution of the logistic map is given by

$$x_t = \frac{1}{2} - \frac{1}{2} \cos(2^t \arccos(1 - 2x_0)).$$

In the C++ program `trajectory2.cpp` we evaluate the exact trajectory up to $t = 10$ using the abstract data type `Verylong` of `SymbolicC++`. For $t = 7$ we find

$$x_7 = \frac{3383826162019367796397224108032}{3433683820292512484657849089281}.$$

```

// trajectory2.cpp

#include <iostream>
#include "verylong.h" // for data type Verylong
#include "rational.h" // for data type Rational
using namespace std;

inline void map(Rational<Verylong>& x)
{
    Rational<Verylong> one("1"); // number 1
    Rational<Verylong> four("4"); // number 4
    Rational<Verylong> x1 = four*x*(one-x);
    x = x1;
}

```

```

int main(void)
{
    Rational<Verylong> x0("1/3"); // initial value 1/3
    unsigned long T = 10;        // number of iterations
    Rational<Verylong> x = x0;
    cout << "x[0] = " << x << endl;
    for(unsigned long t=0;t<T;t++)
    {
        map(x);
        cout << "x[" << t+1 << "] = " << x << endl;
    }
    return 0;
}

```

In the C++ program `trajectory3.cpp` we evaluate the numerical trajectory using the basic data type `double`. We find that the difference between the exact value and the numerical value for $t = 40$ is

$$x_{40exact} - x_{40approx} = 0.055008 - 0.055015 = -0.000007.$$

```

// trajectory3.cpp

#include <iostream>
using namespace std;

inline void map(double& x)
{
    double x1 = 4.0*x*(1.0-x);
    x = x1;
}

int main(void)
{
    double x0 = 1.0/3.0; // initial value
    unsigned long T = 10; // number of iterations
    double x = x0;
    cout << "x[0] = " << x << endl;
    for(unsigned long t=0;t<T;t++)
    {
        map(x);
        cout << "x[" << t+1 << "] = " << x << endl;
    }
    return 0;
}

```

As a third example we consider the *Bernoulli map*. Let $f : [0, 1) \rightarrow [0, 1)$. It is defined by

$$f(x) := 2x \bmod 1 \equiv \text{frac}(2x).$$

The map can be written as the difference equation

$$x_{t+1} = \begin{cases} 2x_t & \text{for } 0 \leq x_t < 1/2 \\ (2x_t - 1) & \text{for } 1/2 \leq x_t < 1 \end{cases}$$

where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1)$. The map admits only one fixed point $x^* = 0$. The fixed point is unstable. Let $x_0 = 1/17$. Then we find the periodic orbit

$$\frac{2}{17}, \frac{4}{17}, \frac{8}{17}, \frac{16}{17}, \frac{15}{17}, \frac{13}{17}, \frac{9}{17}, \frac{1}{17}, \frac{2}{17}, \dots$$

If x_0 is a rational number in the interval $[0, 1)$, then x_t is either periodic or tends to the fixed point $x^* = 0$. The solution of the Bernoulli map is given by

$$x_t = 2^t x_0 \pmod{1}$$

where x_0 is the initial value. For almost all initial values the Liapunov exponent is given by $\ln 2$. Every $x_0 \in [0, 1)$ can be written (uniquely) in the *binary representation*

$$x_0 = \sum_{k=1}^{\infty} a_k 2^{-k}, \quad a_k \in \{0, 1\}.$$

One now defines

$$(a_1, a_2, a_3, \dots) := \sum_{k=1}^{\infty} a_k 2^{-k}$$

and considers the infinite sequence (a_1, a_2, a_3, \dots) . For example, $x_0 = 3/8$ can be represented by the sequence $(0, 1, 1, 0, 0, \dots)$. Instead of investigating the Bernoulli map we can use the map τ defined by

$$\tau(a_1, a_2, a_3, \dots) := (a_2, a_3, a_4, \dots).$$

This map is called the *Bernoulli shift*. In the C++ program `trajectory4.cpp` we find the trajectory of the Bernoulli map using the data type `Rational` and `Verylong` of `SymbolicC++`. The initial value is $1/17$.

```
// trajectory4.cpp

#include <iostream>
#include "verylong.h" // for data type Verylong
#include "rational.h" // for data type Rational
using namespace std;

inline void map(Rational<Verylong>& x)
{
    Rational<Verylong> one("1"); // number 1
    Rational<Verylong> two("2"); // number 2
    Rational<Verylong> half("1/2"); // number 1/2
    Rational<Verylong> x1;
    if(x < half) x1 = two*x;
```

```

    else x1 = two*x-one;
    x = x1;
}

int main(void)
{
    Rational<Verylong> x0("1/17"); // initial value 1/17
    unsigned long T = 10;         // number of iterations
    Rational<Verylong> x = x0;
    cout << "x[0] = " << x << endl;
    for(unsigned long t=0;t<T;t++)
    {
        map(x);
        cout << "x[" << t+1 << "] = " << x << endl;
    }
    return 0;
}

```

As a fourth example we consider the *tent map*. The tent map $f : [0, 1] \rightarrow [0, 1]$ is defined as

$$f(x) := \begin{cases} 2x & \text{if } x \in [0, 1/2) \\ 2(1-x) & \text{if } x \in [1/2, 1] \end{cases}.$$

The map can also be written as the difference equation

$$x_{t+1} = \begin{cases} 2x_t & \text{if } x_t \in [0, 1/2) \\ 2(1-x_t) & \text{if } x_t \in [1/2, 1] \end{cases}$$

where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1]$. Let $x_0 = 1/17$ be the initial value. Then the exact orbit is given by

$$x_0 = \frac{1}{17}, \quad x_1 = \frac{2}{17}, \quad x_2 = \frac{4}{17}, \quad x_3 = \frac{8}{17}, \quad x_4 = \frac{16}{17}, \quad x_5 = \frac{2}{17}, \dots$$

This is an example of an eventually periodic orbit. If the initial value is a rational number then the orbit is eventually periodic, periodic or tends to a fixed point. For example the initial value $1/16$ tends to the fixed point 1. To find chaotic orbits the initial value must be an irrational number, for example $x_0 = 1/\pi$. The fixed points of the map are given by $x^* = 0$, $x^* = 2/3$. These fixed points are unstable. The map shows fully developed chaotic behaviour. The invariant density is given by $\rho(y) = 1$. For almost all initial values the Liapunov exponent is given by $\lambda = \ln 2$. For the autocorrelation function we find

$$C_{xx}(\tau) = \begin{cases} \frac{1}{12} & \text{for } \tau = 0 \\ 0 & \text{for } \tau \geq 1 \end{cases}.$$

The tent map $f : [0, 1] \rightarrow [0, 1]$ given above and the logistic map $g : [0, 1] \rightarrow [0, 1]$, $g(x) = 4x(1-x)$ are *topologically conjugate*, i.e. $f = h \circ g \circ h^{-1}$, where the homeomorphism $h : [0, 1] \rightarrow [0, 1]$ is given by

$$h(x) = \frac{2}{\pi} \arcsin(\sqrt{x}).$$

In the C++ program `trajectory5.cpp` we find the trajectory of the tent map with the initial value $1/17$.

```
// trajectory5.cpp

#include <iostream>
#include "verylong.h"
#include "rational.h"
using namespace std;

inline void map(Rational<Verylong>& x)
{
    Rational<Verylong> one("1"), two("2");
    Rational<Verylong> half("1/2"); // number 1/2
    Rational<Verylong> x1;
    if(x < half) x1 = two*x;
    else x1 = two*(one-x);
    x = x1;
}

int main(void)
{
    Rational<Verylong> x0("1/17"); // initial value 1/17
    unsigned long T = 10;          // number of iterations
    Rational<Verylong> x = x0;
    cout << "x[0] = " << x << endl;
    for(unsigned long t=0;t<T;t++)
    {
        map(x);
        cout << "x[" << t+1 << "] = " << x << endl;
    }
    return 0;
}
```

As a fifth example we consider a *bungalow-tent map*. Our bungalow-tent map $f_r : [0, 1] \rightarrow [0, 1]$ is defined by

$$f_r(x) := \begin{cases} \frac{1-r}{r}x & \text{for } x \in [0, r) \\ \frac{2r}{1-2r}x + \frac{1-3r}{1-2r} & \text{for } x \in [r, 1/2) \\ \frac{2r}{1-2r}(1-x) + \frac{1-3r}{1-2r} & \text{for } x \in [1/2, 1-r) \\ \frac{1-r}{r}(1-x) & \text{for } x \in [1-r, 1] \end{cases}$$

where $r \in (0, 1/2)$ is the control parameter. The map is continuous, but not differentiable at the points r , $1-r$ ($r \neq 1/3$) and $x = 1/2$. The map is piecewise linear. The fixed points are 0 and $1-r$. For $r = 1/3$ we obtain the tent map. The map f_r

is a special bungalow-tent map. The intersection point P of the line in the interval $[1/2, 1 - r)$ and the line in the interval $[1 - r, 1]$ lies on the diagonal $y = x$. The invariant density is given by

$$\rho_r(x) = \frac{1}{2 - 3r} \chi_{[0, 1-r]}(x) + \frac{1 - 2r}{r(2 - 3r)} \chi_{(1-r, 1]}(x)$$

where χ is the *indicator function*, i.e. $\chi_A(x) = 1$ if $x \in A$ and $\chi_A(x) = 0$ if $x \notin A$. Thus the invariant density is constant in the interval $[0, 1 - r)$. At $1 - r$ the invariant density jumps to another constant value. The Liapunov exponent is given by

$$\lambda(r) = \frac{1 - r}{2 - 3r} \ln \left(\frac{1 - r}{r} \right) + \frac{1 - 2r}{2 - 3r} \ln \left(\frac{2r}{1 - 2r} \right).$$

For $r = 1/3$ we obviously obtain $\lambda(1/3) = \ln 2$. This is the Liapunov exponent for the tent map. For $r \rightarrow 0$ we obtain $\lambda(r \rightarrow 0) = \frac{1}{2} \ln 2$. For $r \rightarrow 1/2$ we obtain $\lambda(r \rightarrow 1/2) = 0$. $\lambda(r)$ has a maximum for $r = 1/3$ (tent map). Furthermore $\lambda(r)$ is a convex function in the interval $(0, 1/2)$. Thus we have

$$\lambda(r) \leq \ln 2.$$

The C++ program `trajectory6.cpp` finds the trajectory of the bungalow-tent map for the control parameter $r = 1/7$ and the initial value $x_0 = 1/17$. We find $x_1 = 6/17$, $x_2 = 16/17$, $x_3 = 6/17$. Thus the orbit is eventually periodic.

```
// trajectory6.cpp

#include <iostream>
#include "verylong.h"
#include "rational.h"
using namespace std;

inline void map(Rational<Verylong>& x, Rational<Verylong>& r)
{
    Rational<Verylong> one("1"), two("2"), three("3");
    Rational<Verylong> half("1/2"); // number 1/2
    Rational<Verylong> x1;
    if(x < r) x1 = (one-r)*x/r;
    else if((x >= r) && (x < half))
        x1 = two*r*x/(one-two*r) + (one-three*r)/(one-two*r);
    else if((x >= half) && (x < one-r))
        x1 = two*r*(one-x)/(one-two*r)+(one-three*r)/(one-two*r);
    else if((x <= one) && (x > one-r))
        x1 = (one-r)*(one-x)/r;
    x = x1;
}

int main(void)
{
```

```

Rational<Verylong> x0("1/17"); // initial value 1/17
Rational<Verylong> r("1/7");   // control parameter 1/7
unsigned long T = 10;          // number of iterations
Rational<Verylong> x = x0;
cout << "x[0] = " << x << endl;
for(unsigned long t=0;t<T;t++)
{
  map(x,r);
  cout << "x[" << t+1 << "] = " << x << endl;
}
return 0;
}

```

As another example we consider the *Gauss map*. The Gauss map $f : [0, 1] \rightarrow [0, 1]$ is defined as

$$f(x) := \begin{cases} 0 & \text{if } x = 0 \\ [1/x] & \text{if } x \neq 0 \end{cases}$$

where $[y]$ denotes the *fractional part* of y . For example

$$[3.2] = 0.2, \quad [17/3] = \frac{2}{3}.$$

Owing to the definition $x^* = 0$ is a fixed point. Let $x_0 = 23/101$ be the initial value. Then the orbit is given by

$$x_1 = \frac{9}{23}, \quad x_2 = \frac{5}{9}, \quad x_3 = \frac{4}{5}, \quad x_4 = \frac{1}{4}, \quad x_5 = 0$$

where $x_5 = 0$ is a fixed point. The Gauss map possesses an infinite number of discontinuities and is not injective since each $x_0 \in [0, 1]$ has countable infinite images. The map admits an infinite number of unstable fixed points and shows chaotic behaviour. For example $x^* = (\sqrt{5} - 1)/2$ is a fixed point, since $x^* = f(x^*)$. The Gauss map preserves the Gauss measure on $[0, 1]$ which is given by

$$m(A) := \frac{1}{\ln 2} \int_A \frac{1}{1+x} dx.$$

The periodic points of the Gauss map are the reciprocal of the reduced quadratic irrationals. These numbers are dense in $[0, 1]$.

```

// trajectory7.cpp

#include <iostream>
#include "verylong.h"
#include "rational.h"
using namespace std;

inline void map(Rational<Verylong>& x)
{

```

```

Rational<Verylong> zero("0"), one("1");
Rational<Verylong> x1;
if(x==zero) return;
x1 = one/x;
while(x1 >= one) x1 = x1-one;
x = x1;
}

int main(void)
{
    Rational<Verylong> x0("23/101"); // initial value
    unsigned long T = 10;           // number of iterations
    Rational<Verylong> x = x0;
    cout << "x[0] = " << x << endl;
    for(unsigned long t=0;t<T;t++)
    {
        map(x);
        cout << "x[" << t+1 << "] = " << x << endl;
    }
    return 0;
}

```

The next program `datagnu.cpp` shows how to write the output data from an iteration to a file. We consider the logistic map as an example. The output is stored in a file called `timeev.dat`. We use the C++ style for the file manipulation.

```

// datagnu.cpp

#include <fstream> // for ofstream, close
using namespace std;

int main(void)
{
    ofstream data("timeev.dat");
    unsigned long T = 100; // number of iterations
    double x0 = 0.618;     // initial value
    double x1;
    for(unsigned long t=0;t<T;t++)
    {
        x1 = 4.0*x0*(1.0-x0);
        data << t << " " << x0 << "\n";
        x0 = x1;
    }
    data.close();
    return 0;
}

```

The data files can now be used to draw a graph of the time evolution using *GNU-plot*. After we entered GNU-plot using the command `gnuplot` the plot command is as follows:

```
plot [0:10] 'timeev.dat'
```

This command plots the first eleven points of the time evolution. Furthermore we can create a *postscript file* using the commands:

```
set term postscript default
set output "timeev.ps"
plot 'timeev.dat'
```

The next program shows how to write data to a file and read data from a file using JAVA. As an example we consider the logistic map. The data are stored in a file called "timeev.dat". In the second part of the program we read the data back. In JAVA the filename and the class name which includes the

```
public static void main(String args[])
```

method must coincide (case sensitive). We output data using a `DataOutputStream` that is connected to a `FileOutputStream` via a technique called chaining of stream objects. When the `DataOutputStream` object output is created, its constructor is supplied a `FileOutputStream` object as an argument. The statement creates a `DataOutputStream` object named output associated with the file `timeev.dat`. The argument "timeev.dat" is passed to the `FileOutputStream` constructor which opens the file.

Class `DataOutputStream`. A data output stream lets an application write primitive (basic) Java data types to an output stream in a portable way.

Class `FileOutputStream`. A file output stream is an output stream for writing data to File or a FileDescriptor.

The method `void writeDouble(double v)` converts the double argument to a long using the `doubleToLongBits` method in class `Double`, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.

The method `double readDouble()` reads eight input bytes and returns a double value.

```
// FileManipulation.java

import java.io.*;
import java.lang.Exception;

public class FileManipulation
{
    public static void main(String args[])
    {
        DataOutputStream output;
```

```

try
{
output = new DataOutputStream(new FileOutputStream("timeev.dat"));
int T = 10;
double x0 = 0.618; double x1;
output.writeDouble(x0);
System.out.println("The output is " + x0);
for(int t=0;t<T;t++)
{
x1 = 4.0*x0*(1.0-x0);
System.out.println("The output is " + x1);
output.writeDouble(x1);
x0 = x1;
}
try { output.flush(); output.close(); }
catch(IOException e)
{
System.err.println("File not closed properly\n" + e.toString());
System.exit(1);
}
}
catch(IOException e)
{
System.err.println("File not opened properly\n" + e.toString());
System.exit(1);
}
System.out.println("\nReading file:");
try
{
FileInputStream fin = new FileInputStream("timeev.dat");
DataInputStream in = new DataInputStream(fin);
while(true) System.out.print(in.readDouble() + " ");
}
catch(Exception e) { }
}
}

```

1.1.2 Fixed Points and Stability

Consider a map $f : [0, 1] \rightarrow [0, 1]$. The *fixed points* are defined as the solutions of the equation

$$f(x^*) = x^*.$$

Let us assume that the map f is differentiable. Then the *variational equation* of $x_{t+1} = f(x_t)$ is defined as

$$y_{t+1} = \left. \frac{d}{d\epsilon} f(x + \epsilon y) \right|_{\epsilon=0, x=x_t, y=y_t} = \frac{df}{dx}(x = x_t) y_t.$$

A fixed point is called *stable* if

$$\left| \frac{df}{dx}(x = x^*) \right| < 1.$$

When we consider the logistic map $f : [0, 1] \rightarrow [0, 1]$, $f(x) = 4x(1 - x)$ we have to solve the quadratic equation

$$4x^*(1 - x^*) = x^*$$

to find the fixed points. Therefore the fixed points are given by $x_1^* = 0$, $x_2^* = 3/4$. In the C++ program we consider the stability of the fixed points for the logistic map $x_{t+1} = 4x_t(1 - x_t)$. We evaluate the variational equation of the logistic equation and determine the stability of the fixed points. For the logistic map we find

$$\frac{df}{dx} = 4 - 8x.$$

and therefore the fixed points $x_1^* = 0$ and $x_2^* = 3/4$ are unstable.

In the C++ program `fixpointlog.cpp` we test whether the fixed points of the logistic map $f(x) = 4x(1 - x)$ are unstable. We use the header file `derive.h` from `SymbolicC++` to do the differentiation.

```
// fixpointlog.cpp

#include <iostream>
#include <cmath>      // for fabs
#include "verylong.h"
#include "rational.h"
#include "derive.h"
using namespace std;

int main(void)
{
    double x1 = 0.0;
    double x2 = 3.0/4.0;
    Derive<double> C1(1.0); // constant 1.0
    Derive<double> C4(4.0); // constant 4.0
    Derive<double> X1, X2;
    X1.set(x1);
    Derive<double> R1 = C4*X1*(C1-X1);
    double result1 = df(R1);
    cout << "result1 = " << result1 << endl;
    if(fabs(result1) > 1) cout << "fixpoint x1 unstable " << endl;
    X2.set(x2);
    Derive<double> R2 = C4*X2*(C1-X2);
    double result2 = df(R2);
    cout << "result2 = " << result2 << endl;
    if(fabs(result2) > 1) cout << "fixpoint x2 unstable ";
    return 0;
}
```

1.1.3 Invariant Density

Consider a one-hump fully developed chaotic map $f : [0, 1] \rightarrow [0, 1]$

$$x_{t+1} = f(x_t)$$

where $t = 0, 1, 2, \dots$. We define the *invariant density* (also called *probability density*) of the iterates, starting from an initial point x_0 , by

$$\rho(x) := \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} \delta(x - f^{(t)}(x_0))$$

where $f^{(0)}(x_0) = x_0$ and

$$f^{(1)}(x_0) = f(x_0) = x_1, \dots, \quad f^{(t)}(x_0) = f^{(t-1)}(f(x_0)) = f(f^{(t-1)}(x_0)) = x_t$$

with $t > 1$. Here δ denotes the delta function. Not all starting points $x_0 \in [0, 1]$ are allowed in the definition. Those belonging to an unstable cycle must be excluded since we are only interested in the stable chaotic trajectory. For any arbitrary (but integrable in the Lebesgue sense) function g on the unit interval $[0, 1]$ the *mean value* of that function along the chaotic trajectory is

$$\langle g(x) \rangle := \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} g(x_t) = \int_0^1 \rho(x) g(x) dx.$$

Choosing $g(x) = 1$ we obtain the normalization condition

$$\int_0^1 \rho(x) dx = 1.$$

Since the probability density is independent of the starting point x_0 , the expression for ρ can also be written as

$$\rho(x) = \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} \delta(x - x_{t+k}), \quad k = 0, 1, 2, \dots$$

An integral equation for ρ can be derived as follows: Let σ be defined as

$$\sigma(y) := \int_0^1 \delta(y - f^{(k)}(x)) \rho(x) dx.$$

Let g be an arbitrary (but integrable in the sense of Lebesgue) function on $[0, 1]$. Then

$$\int_0^1 \sigma(y) g(y) dy = \int_0^1 \int_0^1 \delta(y - f^{(k)}(x)) \rho(x) g(y) dy dx.$$

Therefore we obtain

$$\int_0^1 \sigma(y) g(y) dy = \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} \int_0^1 \delta(x - f^{(t)}(x_0)) g(f^{(t)}(x_0)) dx.$$

Using the properties of the delta function we arrive at

$$\int_0^1 \sigma(y)g(y)dy = \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} g(f^{(t+k)}(x_0)).$$

Hence

$$\int_0^1 \sigma(y)g(y)dy = \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} \int_0^1 \delta(y - f^{(t+k)}(x_0))g(y)dy = \int_0^1 \rho(y)g(y)dy.$$

Since the function g is arbitrarily chosen, we have to set $\sigma(y) = \rho(y)$. Thus the probability density ρ obeys the integral equation

$$\rho(y) = \int_0^1 dx \delta(y - f(x))\rho(x).$$

This equation is called the *Frobenius-Perron integral equation*. This equation has many solutions (Kluiving et al [60]). Among these are the solutions associated with the unstable periodic orbits. If these unstable solutions are left out of consideration and the map is one-hump fully developed chaotic, then there is only one stable chaotic trajectory exploring the unit interval $[0, 1]$ and the Frobenius-Perron equation has a unique solution associated with the chaotic orbit.

In the C++ program we determine numerically the invariant density for the logistic map $x_{t+1} = 4x_t(1 - x_t)$. For the stable chaotic trajectory exploring the unit interval $[0, 1]$ the Frobenius-Perron integral equation has the unique solution

$$\rho(x) = \frac{1}{2\pi\sqrt{x(1-x)}}$$

where

$$\int_0^1 \rho(x) dx = 1$$

and $\rho(x) > 0$ for $x \in [0, 1]$. We see that $\rho(x) \rightarrow \infty$ for $x \rightarrow 0$ and $x \rightarrow 1$, respectively. The solution can be found by iteration of

$$\rho_{t+1}(y) = \int_0^1 dx \delta(y - f(x))\rho_t(x)$$

with the initial density $\rho_0(x) = 1$.

In the C++ program `invdensity.cpp` we find the histogram for the logistic map. We divide the unit interval $[0, 1]$ into 20 bins with bin size 0.05 each. We calculate how many points exist in the intervals $[0.05 \cdot i, 0.05 \cdot (i+1))$, where $i = 0, 1, 2, \dots, 19$. This gives an approximation for the invariant density defined above. For example, the number of points in the intervals $[0, 0.05)$ and $[0.95, 1.0]$ is much higher than in the other intervals (bins).

```

// invdensity.cpp

#include <iostream>
#include <cmath>    // for floor, sqrt
using namespace std;

void histogram(double* x,int* hist,double T,double xmax,
              double xmin,int n_bins)
{
    double grad = n_bins/(xmax-xmin);
    for(int t=0;t<T;t++) ++hist[((int) floor(grad*(x[t]-xmin)))]];
}

int main(void)
{
    int T = 10000;    // number of iterations
    double xmax = 1.0; // length of interval xmax-xmin
    double xmin = 0.0;
    double bin_width = 0.05;
    double* x = new double[T];
    int n_bins = (int)(xmax-xmin)/bin_width;
    cout << "number of bins = " << n_bins << endl;

    // generating the data for the histogram
    x[0] = (sqrt(5.0)-1.0)/2.0; // initial value
    for(int t=0;t<(T-1);t++) x[t+1] = 4.0*x[t]*(1.0-x[t]);

    int* hist = new int[n_bins];
    // setting hist[i] to zero
    for(int i=0;i<n_bins;i++) hist[i] = 0;
    histogram(x,hist,T,xmax,xmin,n_bins);

    for(int i=0;i<n_bins;i++)
    cout << "hist[" << i << "] = " << hist[i] << endl;
    delete[] x; delete[] hist;
    return 0;
}

```

As a second example we consider the *sine map*. The sine map $f : [0, 1] \rightarrow [0, 1]$ is defined by

$$f(x) := \sin(\pi x).$$

The map can also be written as a difference equation

$$x_{t+1} = \sin(\pi x_t)$$

where $x_0 \in [0, 1]$. The fixed points are determined by the solution of the equation

$$x^* = \sin(\pi x^*).$$

The map admits two fixed points. One fixed point is given by $x_1^* = 0$. The other fixed point is determined from $x_2^* = \sin(\pi x_2^*)$ and $x_2^* > 0$. We find $x_2^* = 0.73648\dots$. The variational equation of the sine map takes the form

$$y_{t+1} = \pi \cos(\pi x_t) y_t.$$

Both fixed points are unstable. This can be seen by inserting x_1^* and x_2^* into the variational equation.

To find the invariant density for the sine-map we replace in program `invdensity.cpp` the line

```
x[t+1] = 4.0*x[t]*(1.0-x[t]);
```

with

```
x[t+1] = sin(pi*x[t]);
```

and add `const double pi = 3.14159;` in front of this statement. The numerical result suggests that the density for the sine map is quite similar to that of the logistic map.

Next we find the invariant density for the *bungalow-tent map* $f_r : [0, 1] \rightarrow [0, 1]$

$$f_r(x) := \begin{cases} \frac{1-r}{r}x & \text{for } x \in [0, r) \\ \frac{2r}{1-2r}x + \frac{1-3r}{1-2r} & \text{for } x \in [r, 1/2) \\ \frac{2r}{1-2r}(1-x) + \frac{1-3r}{1-2r} & \text{for } x \in [1/2, 1-r) \\ \frac{1-r}{r}(1-x) & \text{for } x \in [1-r, 1] \end{cases}$$

where $r \in (0, 1/2)$. To find the invariant density exactly we solve the Frobenius-Perron integral equation. The Frobenius-Perron integral equation is given by

$$\rho_r(x) = \int_0^1 \rho_r(y) \delta(x - f_r(y)) dy.$$

We apply the identities for the *delta function*

$$\delta(cy) \equiv \frac{1}{|c|} \delta(y), \quad \delta(g(y)) \equiv \sum_n \frac{1}{|g'(y_n)|} \delta(y - y_n)$$

where the sum runs over all zeros with multiplicity 1 and $g'(y_n)$ denotes the derivative of g taken at y_n . Taking these identities into account and differentiating in the sense of generalized functions we obtain the invariant density

$$\rho_r(x) = \frac{1}{2-3r} \chi_{[0, 1-r]}(x) + \frac{1-2r}{r(2-3r)} \chi_{(1-r, 1]}(x)$$

where χ is the *indicator function*, i.e.,

$$\chi_A(x) := \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}.$$

Thus the invariant density is constant in the interval $[0, 1-r)$. At $1-r$ the invariant density jumps to another constant value. In the calculations we have to consider two domains for x , $[0, 1-r)$ and $[1-r, 1]$. The Liapunov exponent is calculated using

$$\lambda(r) = \int_0^1 \rho_r(x) \ln \left| \frac{df_r}{dx} \right| dx$$

where we differentiated in the sense of generalized functions. Thus we find that the Liapunov exponent as a smooth function of the control parameter r is given by

$$\lambda(r) = \frac{1-r}{2-3r} \ln \left(\frac{1-r}{r} \right) + \frac{1-2r}{2-3r} \ln \left(\frac{2r}{1-2r} \right).$$

For $r = 1/3$ we obviously obtain $\lambda(1/3) = \ln 2$. This is the Liapunov exponent for the tent map. For $r \rightarrow 0$ we obtain

$$\lambda(r \rightarrow 0) = \frac{1}{2} \ln 2.$$

For $r \rightarrow 1/2$ we obtain $\lambda(r \rightarrow 1/2) = 0$. The Liapunov exponent $\lambda(r)$ has a maximum for $r = 1/3$ (tent map). Furthermore $\lambda(r)$ is a convex function in the interval $(0, 1/2)$. We have $\lambda(r) \leq \ln 2$. The numerical simulation confirms the result for the invariant density, i.e. constant in the interval $[0, 1-r)$ and another constant in the interval $[1-r, 1]$. In our numerical simulation we have to set one of the bins boundary points to $1-r$.

1.1.4 Liapunov Exponent

Here we calculate the *Liapunov exponent* λ for one-dimensional chaotic maps. Consider the one-dimensional map

$$x_{t+1} = f(x_t)$$

where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1]$. The *variational equation* (also called the *linearized equation*) of this map takes the form

$$y_{t+1} = \frac{df}{dx}(x_t) y_t$$

with $y_0 \neq 0$. We assumed that f is differentiable. The Liapunov exponent λ is defined as

$$\lambda(x_0, y_0) := \lim_{T \rightarrow \infty} \frac{1}{T} \ln \left| \frac{y_T}{y_0} \right|.$$

Here we calculate the Liapunov exponent for the logistic map $x_{t+1} = 4x_t(1-x_t)$. Thus $f(x) = 4x(1-x)$ and

$$\frac{df}{dx} = 4 - 8x.$$

Consequently we obtain the variational equation

$$y_{t+1} = (4 - 8x_t)y_t$$

with $y_0 \neq 0$. The exact solution of the logistic map is given by

$$x_t = \frac{1}{2} - \frac{1}{2} \cos(2^t \arccos(1 - 2x_0)).$$

For almost all initial values we find that the Liapunov exponent is given by $\lambda = \ln 2$. In the C++ program we evaluate the Liapunov exponent by using the variational equation. Overflow occurs if T is made too large. In an alternative method we use nearby trajectories and reset the distance between the two trajectories after each time step. Thus we avoid overflow for large T .

```
// liapunov1.cpp

#include <iostream>
#include <cmath>          // for fabs, log
using namespace std;

int main(void)
{
    unsigned long T = 200; // number of iterations
    double x = 0.3;       // initial value for logistic map
    double y = 1.0;       // initial value for variational map
    double x1, y1;
    for(unsigned long t=0;t<T;t++)
    {
        x1 = x;  y1 = y;
        x = 4.0*x1*(1.0-x1); // logistic map
        y = (4.0-8.0*x1)*y1; // variational map
    }
    // notice that y becomes large very quickly
    double lambda = log(fabs(y))/((double) T); // Liapunov exponent
    cout << "lambda = " << lambda << endl;

    // alternative method
    double eps = 0.001;
    double xeps, xeps1;
    x = 0.3;  xeps = x-eps;
    // x and xeps are nearby points
    double sum = 0.0;
    T = 1000;
    double distance;
    for(unsigned long t=0;t<T;t++)
    {
        x1 = x;  xeps1 = xeps;
        x = 4.0*x1*(1.0-x1);
```

```

    xeps = 4.0*xeps1*(1.0-xeps1);
    double distance = fabs(x-xeps);
    sum += log(distance/eps);
    xeps = x-eps;
}
lambda = sum/((double) T);
cout << "lambda = " << lambda << endl;
return 0;
}

```

In the following program we use the Rational, Verylong and Derive class of SymbolicC++ to find an approximation of the Liapunov exponent. The Derive class provides the derivative. Thus the variational equation is obtained via exact differentiation

```

// Liapunov2.cpp
// Iteration of logistic equation and variational equation

#include <iostream>
#include <cmath>          // for fabs, log
#include "verylong.h"
#include "rational.h"
#include "derive.h"
using namespace std;

int main(void)
{
    int T = 100;          // number of iterations
    double x = 1.0/3.0;  // initial value
    double x1;
    double y = 1.0;
    Derive<double> C1(1.0); // constant 1.0
    Derive<double> C4(4.0); // constant 4.0
    Derive<double> X;
    cout << "t = 0   x = " << x << "   " << "y = " << y << endl;
    for(int t=1;t<=T;t++)
    {
        x1 = x; x = 4.0*x1*(1.0-x1);
        X.set(x1);
        Derive<double> Y = C4*X*(C1-X);
        y = df(Y)*y;
        cout << "t = " << t << "   " << "x = " << x << "   "
             << "y = " << y << endl;
    }
    double lambda = log(fabs(y))/((double) T);
    cout << "approximate value for lambda = " << lambda << endl;
    cout << endl;
    int M = 9;
    Rational<Verylong> u1;

```

```

Rational<Verylong> u("1/3"), v("1");
Rational<Verylong> K1("1"), K2("4");
Derive<Rational<Verylong> > D1(K1); // constant 1
Derive<Rational<Verylong> > D4(K2); // constant 4
Derive<Rational<Verylong> > U;
cout << "j = 0    u = " << u << "    " << "v = " << v << endl;
for(int j=1;j<=M;j++)
{
u1 = u; u = K2*u1*(K1-u1);
U.set(Rational<Verylong>(u1));
Derive<Rational<Verylong> > V = D4*U*(D1-U);
v = df(V)*v;
cout << "j = " << j << "    "
      << "u = " << u << "    " << "v = " << v << endl;
}
lambda = log(fabs(double(v)))/((double) M);
cout << "approximate value for lambda = " << lambda << endl;
return 0;
}

```

As a second example we consider the sine map. The *sine map* $f : [0, 1] \rightarrow [0, 1]$ is defined by

$$f(x) := \sin(\pi x).$$

The map can be written as the difference equation

$$x_{t+1} = \sin(\pi x_t)$$

where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1]$. The variational equation of the sine equation is given by

$$y_{t+1} = \frac{df}{dx}(x = x_t)y_t = \pi \cos(\pi x_t)y_t.$$

To find the Liapunov exponent for the sine-map we replace in program `liap.cpp` the line

```
x = 4.0*x1*(1.0-x1);  xeps = 4.0*xeps1*(1.0-xeps1);
```

by

```
x = sin(pi*x1);  xeps = sin(pi*xeps1);
```

and add `const double pi = 3.14159;` in front of this statement. For $T = 5000$ we find $\lambda = 0.689$. Thus there is numerical evidence that the sine-map shows chaotic behaviour.

1.1.5 Autocorrelation Function

Consider a one-dimensional difference equation $f : [0, 1] \rightarrow [0, 1]$

$$x_{t+1} = f(x_t)$$

where $t = 0, 1, 2, \dots$. The *time average* is defined as

$$\langle x_t \rangle := \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} x_t.$$

Obviously, $\langle x_t \rangle$ depends on the initial value x_0 . The *autocorrelation function* is defined as

$$C_{xx}(\tau) := \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^{T-1} (x_t - \langle x_t \rangle)(x_{t+\tau} - \langle x_t \rangle)$$

where $\tau = 0, 1, 2, \dots$. The autocorrelation function depends on the initial value x_0 .

For the logistic map $f : [0, 1] \rightarrow [0, 1]$, $f(x) = 4x(1 - x)$ we find that the time average for almost all initial conditions is given by

$$\langle x_t \rangle = \frac{1}{2}.$$

The autocorrelation function is given by

$$C_{xx}(\tau) = \begin{cases} \frac{1}{8} & \text{for } \tau = 0 \\ 0 & \text{otherwise} \end{cases}$$

for almost all initial conditions. The C++ program `autocorrelation.cpp` calculates the time average and autocorrelation function for the logistic map.

```
// autocorrelation.cpp

#include <iostream>
using namespace std;

double average(double* x,int T)
{
    double sum = 0.0;
    for(int t=0;t<T;t++) { sum += x[t]; }
    double av = sum/((double) T);
    return av;
}

void autocorr(double* x,double* CXX,int T,int length,double av)
{
    for(int tau=0;tau<length;tau++)
    {
        double C = 0.0;
        double diff = (double) (T-length);
        for(int t=0;t<diff;t++) { C += (x[t]-av)*(x[t+tau]-av); }
        CXX[tau] = C/(diff+1.0);
    } // end for loop tau
}
```

```

int main(void)
{
    const int T = 4096;
    double* x = new double[T];
    x[0] = 1.0/3.0;           // initial value
    for(int t=0;t<(T-1);t++) { x[t+1] = 4.0*x[t]*(1.0-x[t]); }
    double av = average(x,T);
    cout << "average value = " << av << endl;
    int length = 11;
    double* CXX = new double[length];
    autocorr(x,CXX,T,length,av);
    delete[] x;
    for(int tau=0;tau<length;tau++)
    cout << "CXX[" << tau << "] = " << CXX[tau] << endl;
    delete[] CXX;
    return 0;
}

```

The output is (exact solution is 0.5, CXX[0]=1/8, CXX[1]=0, CXX[2]=0 etc.)

```

average value = 0.497383
CXX[0] = 0.125707
CXX[1] = 0.00134996
CXX[2] = -0.000105384
CXX[3] = -0.000289099
CXX[4] = 0.00477107
CXX[5] = -0.00186259
CXX[6] = 0.00383531
CXX[7] = -0.00425356
CXX[8] = -0.00288615
CXX[9] = -0.00110183
CXX[10] = -0.00148765

```

1.1.6 Discrete Fourier Transform

The *discrete Fourier transform* is an approximation of the continuous Fourier transform. The discrete transform is used when a set of function sample values, $x(t)$, are available at equally spaced time intervals numbered $t = 0, 1, \dots, T - 1$. The discrete Fourier transform maps the given set of function values into a set of uniformly spaced sine waves whose frequencies are numbered $k = 0, 1, \dots, T - 1$, and whose amplitudes are given by

$$\hat{x}(k) = \frac{1}{T} \sum_{t=0}^{T-1} x(t) \exp\left(-i2\pi k \frac{t}{T}\right).$$

This equation can be written as

$$\hat{x}(k) = \frac{1}{T} \sum_{t=0}^{T-1} x(t) \cos\left(2\pi k \frac{t}{T}\right) - \frac{i}{T} \sum_{t=0}^{T-1} x(t) \sin\left(2\pi k \frac{t}{T}\right).$$

The inverse discrete Fourier transformation is given by

$$x(t) = \sum_{k=0}^{T-1} \hat{x}(k) \exp\left(i2\pi t \frac{k}{T}\right).$$

To find the inverse Fourier transformation we use the fact that

$$\sum_{k=0}^{T-1} \exp\left(i2\pi k \frac{(n-m)}{T}\right) = T\delta_{nm}$$

where δ_{nm} denotes the Kronecker symbol.

In our first C++ program (`Fourier.cpp`) we consider the time series

$$x(t) = \cos(2\pi t/T)$$

where $T = 8$ and $t = 0, 1, 2, \dots, T-1$. We find the discrete Fourier transform $\hat{x}(k)$ ($k = 0, 1, 2, \dots, T-1$). We have

$$\hat{x}(k) = \frac{1}{8} \sum_{t=0}^7 \cos\left(\frac{2\pi t}{8}\right) e^{-i2\pi kt/8}.$$

Using the identity

$$\cos(2\pi t/8) \equiv \frac{e^{i2\pi t/8} + e^{-i2\pi t/8}}{2}$$

we find

$$\hat{x}(k) = \frac{1}{16} \sum_{t=0}^7 (e^{i2\pi t(1-k)/8} + e^{-i2\pi t(1+k)/8}).$$

Consequently,

$$\hat{x}(k) = \begin{cases} \frac{1}{2} & \text{for } k = 1 \\ \frac{1}{2} & \text{for } k = 7 \\ 0 & \text{otherwise} \end{cases}.$$

In our second program (`fourierlog.cpp`) we consider the logistic map $x_{t+1} = 4x_t(1-x_t)$, where $t = 0, 1, 2, \dots$ and $x_0 \in [0, 1]$. We assume that we have a set of T samples from the logistic map, i.e., $x_0, x_1, x_2, \dots, x_{T-1}$.

```
// fourier.cpp

#include <iostream>
#include <cmath> // for cos, sin
using namespace std;
```

```

int main(void)
{
    const double pi = 3.14159;
    int T = 8;
    double* x = new double[T];
    for(int t=0;t<T;t++) x[t] = cos(2.0*pi*t/((double) T));
    double* rex = new double[T]; double* imx = new double[T];
    for(int k=0;k<T;k++)
    {
        double cossum = 0.0, sinsum = 0.0;
        for(int t=0;t<T;t++)
        {
            cossum += x[t]*cos(2.0*pi*k*t/((double) T));
            sinsum += x[t]*sin(2.0*pi*k*t/((double) T));
        }
        rex[k] = cossum/((double) T);
        imx[k] = -sinsum/((double) T);
    }
    // display the output
    for(int k=0;k<T;k++)
    {
        cout << "rex[" << k << "]" = " << rex[k] << " ";
        cout << "imx[" << k << "]" = " << imx[k] << endl;
    }
    delete[] x; delete[] rex; delete[] imx;
    return 0;
}

```

// fourierlog.cpp

```

#include <iostream>
#include <cmath> // for cos, sin
using namespace std;

int main(void)
{
    const double pi = 3.14159;
    int T = 256;
    double* x = new double[T];
    x[0] = 0.5;
    for(int t=0;t<(T-1);t++) x[t+1] = 4.0*x[t]*(1.0-x[t]);
    double* rex = new double[T]; double* imx = new double[T];

    for(int k=0;k<T;k++)
    {
        double cossum = 0.0, sinsum = 0.0;
        for(int t=0;t<T;t++)
        {

```

```

cossum += x[t]*cos(2.0*pi*k*t/((double) T));
sinsum += x[t]*sin(2.0*pi*k*t/((double) T));
}
rex[k] = cossum/((double) T);  imx[k] = -sinsum/((double) T);
}

// display the output
for(int k=0;k<T;k++)
{
cout << "rex[" << k << "]" = " << rex[k] << "    ";
cout << "imx[" << k << "]" = " << imx[k] << endl;
}
delete[] x;  delete[] rex;  delete[] imx;
return 0;
}

```

1.1.7 Fast Fourier Transform

Let $n \geq 1$. The discrete Fourier transform transforms an n -vector with real components into a complex n -vector. Methods that compute the discrete Fourier transform in $O(N \log N)$ complex floating-point operations are referred to as *fast Fourier transforms*, FFT for short. Based on the odd-even decomposition of a trigonometric polynomial, a problem of size $n = 2^k$ is reduced to two problems of size 2^{k-1} . Subsequently, two problems of size 2^{k-1} are reduced to two problems of size 2^{k-2} . Ultimately, $n = 2^k$ problems of size 1 are obtained, each of which is solved trivially. Let ω be a *primitive n th root* of 1, i.e.

$$\omega = \exp(2\pi i/n).$$

The matrix F_n denotes the $n \times n$ matrix with entries

$$f_{jk} := \omega^{jk} \equiv e^{2\pi ijk/n}$$

where $0 \leq j, k \leq n-1$. The discrete Fourier transform of the n -vector

$$P^T = (p_0, p_1, \dots, p_{n-1})$$

is the product $F_n P$. The components of $F_n P$ are

$$\begin{aligned}
(F_n P)_0 &= \omega^0 p_0 + \omega^0 p_1 + \dots + \omega^0 p_{n-2} + \omega^0 p_{n-1} \\
(F_n P)_1 &= \omega^0 p_0 + \omega p_1 + \dots + \omega^{n-2} p_{n-2} + \omega^{n-1} p_{n-1} \\
&\vdots \\
(F_n P)_i &= \omega^0 p_0 + \omega^i p_1 + \dots + \omega^{i(n-2)} p_{n-2} + \omega^{i(n-1)} p_{n-1} \\
&\vdots \\
(F_n P)_{n-1} &= \omega^0 p_0 + \omega^{n-1} p_1 + \dots + \omega^{(n-1)(n-2)} p_{n-2} + \omega^{(n-1)(n-1)} p_{n-1}.
\end{aligned}$$

Rewritten in a slightly different form, the i th component is

$$p_{n-1}(\omega^i)^{n-1} + p_{n-2}(\omega^i)^{n-2} + \cdots + p_1\omega^i + p_0.$$

Thus if we interpret the components of P as coefficients of the polynomial

$$p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \cdots + p_1x + p_0$$

then the i th component is $p(\omega^i)$ and computing the discrete Fourier transform of P means evaluating the polynomial $p(x)$ at $\omega^0, \omega, \omega^2, \dots, \omega^{n-1}$, i.e., at each of the n th roots of 1. We describe a *Divide and Conquer algorithm* first and then examine it closely to remove the recursion. We assume that $n = 2^k$ for some $k \geq 0$. The strategy of Divide and Conquer is to divide the problem into smaller instances, solve those, and use the solutions to get the solution for the current instance. Here, to evaluate p at n points, we evaluate two smaller polynomials at a subset of the points and then combine the results appropriately. Since $\omega^{n/2} = -1$ we have for $0 \leq j \leq n/2 - 1$,

$$\omega^{(n/2)+j} = -\omega^j.$$

We group the terms of $p(x)$ with even powers and the terms with odd powers as follows

$$p(x) = \sum_{i=0}^{n-1} p_i x^i \equiv \sum_{i=0}^{n/2-1} p_{2i} x^{2i} + x \sum_{i=0}^{n/2-1} p_{2i+1} x^{2i}.$$

We define

$$p_{\text{even}}(x) := \sum_{i=0}^{n/2-1} p_{2i} x^i, \quad p_{\text{odd}}(x) := \sum_{i=0}^{n/2-1} p_{2i+1} x^i.$$

Then

$$p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2), \quad p(-x) = p_{\text{even}}(x^2) - x \cdot p_{\text{odd}}(x^2).$$

To evaluate p at

$$1, \omega, \dots, \omega^{(n/2)-1}, -1, -\omega, \dots, -\omega^{(n/2)-1}$$

it suffices to evaluate p_{even} and p_{odd} at

$$1, \omega^2, \dots, (\omega^{(n/2)-1})^2$$

and then do $n/2$ multiplications (for $x \cdot p_{\text{odd}}(x^2)$) and n additions and subtractions. The polynomials p_{even} and p_{odd} can be evaluated recursively by the same scheme. That is, they are polynomials of degree $n/2 - 1$ and will be evaluated at the $n/2$ th roots of unity

$$1, \omega^2, \dots, (\omega^{(n/2)-1})^2.$$

```
// fft1.cpp
```

```
#include <iostream>
```

```

#include <cmath> // for cos, sin
using namespace std;

void p(double wre,double wim,double *re,double *im,
       double &fftre,double &fftim,const int M,int step,int init)
{
    double pre, pim, w2re, w2im;
    if(step==(1 << M))
    {
        fftre = re[init]*wre-im[init]*wim;
        fftim = im[init]*wre+re[init]*wim;
        return;
    }
    w2re = wre*wre-wim*wim; w2im = 2.0*wre*wim;
    p(w2re,w2im,re,im,pre,pim,M,step<<1,init); // peven
    fftre = pre; fftim = pim;
    p(w2re,w2im,re,im,pre,pim,M,step<<1,init+step); // podd
    fftre += wre*pre-wim*pim;
    fftim += wre*pim+wim*pre;
}

void fft(double *re,double *im,double *ftre,double *ftim,const int M)
{
    const double pi = 3.1415927;
    int N = 1 << M;
    double fftre, fftim, wre, wim, w2re, w2im;
    for(int i=0;i<(N>>1);i++)
    {
        wre = cos(i*2.0*pi/N); wim = sin(i*2.0*pi/N);
        w2re = wre*wre-wim*wim; w2im = 2.0*wre*wim;
        p(w2re,w2im,re,im,fftre,fftim,M,2,0); // peven
        ftre[i] = ftre[i+(N>>1)] = fftre;
        ftim[i] = ftim[i+(N>>1)] = fftim;
        p(w2re,w2im,re,im,fftre,fftim,M,2,1); // podd
        ftre[i] += wre*fftre-wim*fftim;
        ftre[i+(N>>1)] -= wre*fftre-wim*fftim;
        ftim[i] += wre*fftim+wim*fftre;
        ftim[i+(N>>1)] -= wre*fftim+wim*fftre;
    }
}

int main(void)
{
    const double pi = 3.1415927;
    const int M = 3;
    int T = 1 << M;
    double* re = new double[T];
    double* im = new double[T];

```

```

double* fftre = new double[T];
double* fftim = new double[T];
for(int i=0;i<T;i++) { re[i] = cos(2.0*i*pi/T); }
for(int k=0;k<T;k++) { im[k] = 0.0; }
fft(re,im,fftre,fftim,M);
for(int k=0;k<T;k++)
cout << "fftre[" << k << "]" << " = " << fftre[k]/T << endl;
cout << endl;
for(int k=0;k<T;k++)
cout << "fftim[" << k << "]" << " = " << fftim[k]/T << endl;
delete[] re; delete[] im;
delete[] fftre; delete[] fftim;
return 0;
}

```

A nonrecursive version is given below. We use in place substitution.

```

// FFT2.cpp

#include <iostream>
#include <cmath> // for sqrt, cos
using namespace std;

// dir = 1 gives the FFT tranform
// dir = -1 gives the inverse FFT transform
// n = 2^m is the length of the time series
// x[] is the real part of the signal
// y[] is the imaginary part of the signal

void FFT(int dir,unsigned long m,double* x,double* y)
{
    unsigned long n, i, i1, j, k, i2, l, l1, l2;
    double c1, c2, tx, ty, t1, t2, u1, u2, z;
    // number of points n = 2^m
    n = 1;
    for(i=0;i<m;i++) n *= 2;
    // bit reversal
    i2 = n >> 1;
    j = 0;
    for(i=0;i<n-1;i++)
    {
        if(i < j)
        {
            tx = x[i]; ty = y[i];
            x[i] = x[j]; y[i] = y[j]; x[j] = tx; y[j] = ty;
        }
        k = i2;
        while(k <= j) { j -= k; k >>= 1; }
        j += k;
    }
}

```

```

} // end for loop

// compute the FFT
c1 = -1.0; c2 = 0.0;
l2 = 1;
for(l=0;l<m;l++)
{
l1 = l2;
l2 <<= 1;
u1 = 1.0; u2 = 0.0;
for(j=0;j<l1;j++)
{
for(i=j;i<n;i+=l2)
{
i1 = i + l1;
t1 = u1*x[i1]-u2*y[i1]; t2 = u1*y[i1]+u2*x[i1];
x[i1] = x[i]-t1; y[i1] = y[i]-t2;
x[i] += t1; y[i] += t2;
}
z = u1*c1-u2*c2;
u2 = u1*c2+u2*c1;
u1 = z;
}
c2 = sqrt((1.0-c1)/2.0);
if(dir == 1) c2 = -c2;
c1 = sqrt((1.0+c1)/2.0);
}
if(dir==1)
{
for(i=0;i<n;i++) { x[i] /= n; y[i] /= n; }
}
} // end function FFT

unsigned long power(unsigned long m)
{
unsigned long r = 1;
for(unsigned long i=0;i<m;i++) r *= 2;
return r;
}

int main(void)
{
unsigned long m = 3;
const double pi = 3.14159;
unsigned long n = power(m);
double* x = new double[n]; double* y = new double[n];

unsigned long k;

```

```

for(k=0;k<n;k++) { x[k] = cos(2.0*pi*k/n); y[k] = 0.0; }
// call FFT
FFT(1,m,x,y);
for(k=0;k<n;k++) { cout << x[k] << "    " << y[k] << endl; }

cout << "calling inverse FFT" << endl;
// call inverse FFT
FFT(-1,m,x,y);
for(k=0;k<n;k++) { cout << x[k] << "    " << y[k] << endl; }
return 0;
}

```

1.1.8 Logistic Map and Liapunov Exponent for $r \in [3, 4]$

We consider the logistic map

$$x_{t+1} = rx_t(1 - x_t)$$

where $t = 0, 1, 2, \dots$, $x_0 \in [0, 1]$ and $r \in [3, 4]$. Here r is the bifurcation parameter. Thus the Liapunov exponent depends on r . We evaluate the Liapunov exponent for $r \in [3, 4]$. The variational equation is given by

$$y_{t+1} = r(1 - 2x_t)y_t.$$

The *Liapunov exponent* is defined as

$$\lambda(x_0, y_0) := \lim_{T \rightarrow \infty} \frac{1}{T} \ln \left| \frac{y_T}{y_0} \right|.$$

The point $r = 3$ is a bifurcation point. The Liapunov exponent is given by $\lambda = 0$. In the range $3 < r < 3.5699\dots$ we find periodic solutions. The Liapunov exponent is negative. We also find period doubling. In the region

$$3.5699\dots < r < 4$$

we find chaotic behaviour (positive Liapunov exponent) but also periodic windows. For example in the region

$$3.828\dots < r < 3.842\dots$$

we have a trajectory with period 3. The Liapunov exponent can be evaluated exactly only for $r = 4$. One finds $\lambda(r = 4) = \ln 2$ for almost all initial values. In the program `lambdaf.cpp` the Liapunov exponent is evaluated for the interval $r \in [3.0, 4.0]$ with step size 0.001.

```

// lambdaf.cpp

#include <fstream>
#include <cmath> // for fabs, log
using namespace std;

```

```

int main(void)
{
    ofstream data("lambda.dat");
    int T = 10000;          // number of iterations
    double x = 0.618;     // initial value
    double x1;
    double eps = 0.0005;
    double xeps = x-eps;
    double xeps1;
    double r = 3.0;
    double sum = 0.0;
    while(r <= 4.0)
    {
        for(int t=0;t<T;t++)
        {
            x1 = x; x = r*x1*(1.0-x1);
            xeps1 = xeps; xeps = r*xeps1*(1.0-xeps1);
            double distance = fabs(x-xeps1);
            sum += log(distance/eps);
            xeps = x-eps;
        }
        double lambda = sum/((double) T);
        data << r << " " << lambda << "\n";
        sum = 0.0;
        r += 0.001;
    } // end while
    data.close();
    return 0;
}

```

1.1.9 Logistic Map and Bifurcation Diagram

We consider the logistic map

$$x_{t+1} = rx_t(1 - x_t)$$

where $r \in [2, 4]$ and $x_0 \in [0, 1]$. Here r is a bifurcation parameter. We now study the bifurcation diagram. For $r \in [2, 3)$ the fixed point $x^* = 1 - 1/r$ is stable. The fixed point $x^* = 0$ is unstable in the range $(2, 4]$. For $r = 3$ (bifurcation point) the stable fixed point $x^* = 1 - 1/r$ becomes unstable. We find a stable orbit of period 2. With increasing r we find a period doubling process with repeated bifurcation from

$$2, 4, 8, \dots, 2^n, \dots$$

There is a threshold value

$$r_\infty = 3.5699\dots$$

for the parameter r where the limit 2^n , $n \rightarrow \infty$ of the periodicity is reached. For $r = 4$ the logistic map and all its iterates are ergodic and mixing. Within the interval $(r_\infty, 4)$ period triplings $p3^n$ and quadruplings $p4^n$ etc. also occur (so-called *periodic windows*)

In the Java program `Bifurcationlo.java` we display the bifurcation diagram for the interval $r \in [2.0, 4.0]$.

```
// Bifurcationlo.java

import java.awt.*;
import java.awt.Frame;
import java.awt.event.*;
import java.awt.Graphics;

public class Bifurcationlo extends Frame
{
    public Bifurcationlo()
    {
        setSize(600,500);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent event)
        { System.exit(0); }}); }

    public void paint(Graphics g)
    {
        int xmax = 600; int ymax = 400;
        int j, k, m, n;
        double x, xplot, yplot;
        double r = 2.0; // bifurcation parameter
        while(r <= 4.0)
        {
            xplot = xmax*(r-2.0)/2.0;
            x = 0.5;
            for(j=0;j<400;j++) { x = r*x*(1.0-x); }
            for(k=0;k<400;k++)
            {
                x = r*x*(1.0-x);
                yplot = ymax*(1.0-x);
                m = (int) Math.round(xplot);
                n = 50 + (int) Math.round(yplot);
                g.drawLine(m,n,m,n);
            }
            r += 0.0005;
        } // end while
    }

    public static void main(String[] args)
```

```

{
  Frame f = new Bifurcationlo(); f.setVisible(true);
}
}

```

1.1.10 Random Number Map and Invariant Density

We consider methods for generating a sequence of random fractions, i.e., random real numbers u_t , uniformly distributed between zero and one. Since a computer can represent a real number with only finite accuracy, we shall actually be generating integers x_t between zero and some number m . The fraction

$$u_t = x_t/m, \quad t = 0, 1, 2, \dots$$

will then lie between zero and one. Usually m is the word size of the computer, so x_t may be regarded as the integer contents of a computer word with the radix point assumed at the extreme right, and u_t may be regarded as the contents of the same word with the radix point assumed at the extreme left. The most popular random number generators are special cases of the following scheme. We select four numbers

$$\begin{aligned}
& m, \text{ the modulus; } & m > 0 \\
& a, \text{ the multiplier; } & 0 \leq a < m \\
& c, \text{ the increment; } & 0 \leq c < m \\
& x_0, \text{ the initial value; } & 0 \leq x_0 < m.
\end{aligned}$$


The desired sequence of pseudo-random numbers x_0, x_1, x_2, \dots is then obtained by the one-dimensional difference equation

$$x_{t+1} = (ax_t + c) \pmod{m}, \quad t = 0, 1, 2, \dots$$

This is also called a *linear congruential sequence*. Taking the remainder mod m is somewhat like determining where a ball will land in a spinning roulette wheel.

Example. The sequence obtained when $m = 10$ and $x_0 = a = c = 7$ is

$$7, \quad 6, \quad 9, \quad 0, \quad 7, \quad 6, \quad 9, \quad 0, \quad \dots$$

This example shows that the sequence is not always “random” for all choices of m , a , c , and x_0 . 

The example also illustrates the fact that congruential sequences always “get into a loop”; i.e., there is ultimately a cycle of numbers which is repeated endlessly. The repeating cycle is called the period. The sequence given above has a period of length 4. A useful sequence will of course have a relatively long period. In our first C++ program we implement a linear congruential sequence. In our second C++ program we consider the sequence

$$x_{t+1} = (\pi + x_t)^5 \pmod{1} \equiv \text{frac}(\pi + x_t)^5$$

and ask whether the sequence is uniformly distributed.

```
// modulus.cpp

#include <iostream>
using namespace std;

int main(void)
{
    unsigned long a = 7, c = 7;
    unsigned long m = 10; // modulus
    unsigned long x0 = 7; // initial value
    int T = 10;           // number of iterations
    unsigned long x1;
    for(int t=0;t<T;t++)
    {
        x1 = a*x0 + c;
        while(x1 >= m) x1 = x1-m;
        x0 = x1;
        cout << "x[" << t << "] = " << x0 << endl;
    }
    a = 3125; c = 47;
    m = 2048; // modulus
    x0 = 3; // initial value
    T = 12; // number of iterations
    for(int t=0;t<T;t++)
    {
        x1 = a*x0 + c;
        while(x1 >= m) x1 = x1-m;
        x0 = x1;
        cout << "x[" << t << "] = " << x0 << endl;
    }
    return 0;
}

// random1.cpp

#include <iostream>
#include <cmath> // for sqrt, fmod
using namespace std;

int main(void)
{
    const double pi = 3.14159;
    int T = 6000; // number of iterations
    double* x = new double[T];
    x[0] = (sqrt(5.0)-1.0)/2.0; // initial value
    for(int t=0;t<(T-1);t++)
    { double r = x[t]+pi; x[t+1] = fmod(r*r*r*r*r,1); }
    const int N = 10;
```

```

double hist[N];
for(int j=0;j<N;j++) hist[j] = 0.0;

for(int k=0;k<T;k++)
hist[(int) floor(N*x[k])] = hist[(int) floor(N*x[k])+1];
for(int l=0;l<N;l++)
cout << "hist[" << l << "] = " << hist[l] << endl;
return 0;
}

```

1.1.11 Random Number Map and Random Integration

We describe the *Monte Carlo method* for the calculation of integrals. We demonstrate the technique on one-dimensional integrals. Let $f : [0, 1] \rightarrow [0, 1]$ be a continuous function. Consider the integral

$$I = \int_0^1 f(x)dx.$$

We choose N number pairs (x_j, y_j) with uniform distribution and define z_i by

$$z_i := \begin{cases} 0 & \text{if } y_j > f(x_j) \\ 1 & \text{if } y_j \leq f(x_j). \end{cases}$$

Putting

$$n = \sum_j z_j$$

we have $n/N \simeq I$. More precisely, we find

$$I = n/N + O(N^{-1/2}).$$

The accuracy here is not very good. The traditional formulas, such as Simpson's formula, are much better. However, in higher dimensions the Monte Carlo technique is extremely favourable, at least if the number of dimensions is ≥ 6 . We consider the integral as the mean value of $f(\xi)$ where ξ is uniform. An estimate of the mean value is

$$I \simeq \frac{1}{N} \sum_{j=1}^N f(\xi_j).$$

This formula can easily be generalized to higher dimensions. In the C++ program we use the map

$$f(x) = (x + \pi)^5 \pmod{1}$$

as random number generator and evaluate

$$\int_0^1 \sin(x)dx = 0.459697694132.$$

```
// randint.cpp

#include <iostream>
#include <cmath>
using namespace std;

void randval(double* x,double pi)
{ *x = fmod((*x+pi)*(*x+pi)*(*x+pi)*(*x+pi)*(*x+pi),1); }

int main(void)
{
    const double pi = 3.14159;
    unsigned long T = 20000; // number of iterations
    double x = 0.5; // initial value
    double sum = 0.0;
    for(int t=0;t<T;t++) { randval(&x,pi); sum += sin(x); }
    cout << "The integral is = " << sum/((double) T);
    return 0;
}
```

In the Java program we use the same map as in the C++ program for the random number generator.

```
// Random1.java

class WrappedDouble
{
    WrappedDouble(final double value) { this.value = value; }
    public double value() { return value; }
    public void value(final double newValue) { value = newValue; }
    private double value;
}

class MathUtils
{
    public static void randval(WrappedDouble x)
    {
        double y = Math.pow(x.value()+Math.PI,5);
        x.value(y-Math.floor(y));
    }
}

class Random1
{
    public static void main(String[] args)
    {
        int n = 20000;
        double sum = 0.0;
        WrappedDouble x = new WrappedDouble(0.5);
    }
}
```

```

for(int i=0;i<n;++i)
{ MathUtils.randval(x); sum += Math.sin(x.value()); }
System.out.println("The integral is " + sum/n);
}
}

```

1.1.12 Circle Map and Rotation Number

The *circle map* is given by

$$x_{t+1} = f(x_t) \equiv x_t + \Omega - \frac{r}{2\pi} \sin(2\pi x_t)$$

which may be regarded as a transformation of the phase of one oscillator through a period of the second one. The map depends on two bifurcation parameters: Ω describes the ratio of undisturbed frequencies while the bifurcation parameter r governs the strength of the nonlinear interaction. The subcritical ($r < 1$) mappings are diffeomorphisms (and thus invertible) whereas the supercritical ones ($r > 1$) are non-invertible and may exhibit chaotic behaviour. The borderline between these two cases consists of the critical circle mappings - homeomorphisms with one (usually cubic) inflection point. This corresponds to $r = 1$ in the family of this map. The dynamics of the map may be characterized by the *rotation number* (also called winding number)

$$\rho := \lim_{T \rightarrow \infty} \frac{1}{T} (f^{(T)}(x) - x).$$

When f is invertible, the rotation number is well defined and independent of x . f^{-1} does not exist for $r > 1$. For subcritical and critical maps this number does not depend on the initial point x . The dependence $\rho(\Omega)$ is the so-called *devil's staircase*, in which each rational $\rho = p/q$ is represented by an interval of Ω values (which is named the p/q -locking interval). The set of all these intervals has a full measure in the critical case. The locked motion in subcritical and critical cases is represented by a stable periodic orbit of period q . The rotation number is the mean number of rotations per iteration, i.e., the frequency of the underlying dynamical system. If $r = 0$ we obviously find $\rho = \Omega$. Under iteration the variable x_i may converge to a series which is either periodic,

$$x_{i+Q} = x_i + P$$

with rational rotation number $\rho = P/Q$; quasiperiodic, with irrational rotation number $\rho = q$; or chaotic where the sequence behaves irregularly.

```

// circle.cpp

#include <fstream>
#include <cmath>      // for sin
using namespace std;

int main(void)

```

```

{
  ofstream data("circle.dat");
  const double pi = 3.14159;
  int T = 8000;    // number of iterations
  double r = 1.0; // parameter of map
  double Omega = 0.0;

  while(Omega <= 1.0)
  {
    double x = 0.3;    // initial value
    double x0 = x;
    double x1;
    for(int t=0;t<T;t++)
    {
      x1 = x;
      x = x1+Omega-r*sin(2.0*pi*x1)/(2.0*pi); // circle map
    }
    double rho = (x-x0)/((double) T);
    data << Omega << " " << rho << "\n";
    Omega += 0.005;
  } // end while
  data.close();
  return 0;
}

```

1.1.13 Newton Method

Consider the equation $f(x) = 0$ where it is assumed that $f : \mathbf{R} \rightarrow \mathbf{R}$ is at least twice differentiable. Let I be some interval containing a root of f . A root is a point \tilde{x} such that $f(\tilde{x}) = 0$. We assume that the root is simple (also called multiplicity one). The *Newton method* can be derived by taking the tangent line to the curve $y = f(x)$ at the point $(x_t, f(x_t))$ corresponding to the current estimate, x_t of the root. The intersection of this line with the x -axis gives the next estimate to the root, x_{t+1} . The gradient of the curve $y = f(x)$ at the point $(x_t, f(x_t))$ is $f'(x_t)$. The tangent line at this point has the form $y = f'(x)x + b$. Since this passes through $(x_t, f(x_t))$ we see that $b = f(x_t) - x_t f'(x_t)$. Therefore the tangent line is

$$y = f'(x_t)x + f(x_t) - x_t f'(x_t).$$

To determine where this line cuts the x -axis we set $y = 0$. Taking this point of intersection as the next estimate, x_{t+1} , to the root we have

$$0 = f'(x_t)x_{t+1} + f(x_t) - x_t f'(x_t).$$

We obtain the first order difference equation

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}, \quad t = 0, 1, 2, \dots$$

This is the Newton-Raphson method. This scheme has the form 'next estimate = current estimate + correction term'. The correction term is $-f(x_t)/f'(x_t)$ and this must be small when x_t is close to the root if convergence is to be achieved. This will depend on the behaviour of $f'(x)$ near the root and, in particular, difficulty will be encountered when $f'(x)$ and $f(x)$ have roots close together. The Newton-Raphson method is of the form $x_{t+1} = g(x_t)$ with

$$g(x) := x - \frac{f(x)}{f'(x)}.$$

The order of the method can be examined. Differentiating this equation leads to

$$g'(x) = \frac{f(x)f''(x)}{(f'(x))^2}.$$

For convergence we require that

$$\left| \frac{f(x)f''(x)}{(f'(x))^2} \right| < 1$$

for all x in some interval I containing the root. Since $f(\tilde{x}) = 0$, the above condition is satisfied at the root $x = \tilde{x}$ provided that $f'(\tilde{x}) \neq 0$. Then provided that $g(x)$ is continuous, an interval I must exist in the neighbourhood of the root and over which the condition above is satisfied. Difficulty is sometimes encountered when the interval I is small, because the initial guess must be taken from this interval. This usually arises when $f(x)$ and $f'(x)$ have roots close together, since the correction term is inversely proportional to $f'(x)$.

In the C++ program `Newton.cpp` we consider the function

$$f(x) = x - \sin(\pi x)$$

in the interval $[0.5, 1]$. This means we find the fixed point of the sine-map

$$x_{t+1} = \sin(\pi x_t)$$

in the interval $[0.5, 1]$. We have

$$f'(x) = 1 - \pi \cos(\pi x), \quad f''(x) = \pi^2 \sin(\pi x).$$

Thus the condition for convergence is satisfied.

```
// Newton.cpp

#include <iostream>
#include <cmath>    // for sin, cos, fabs
using namespace std;

double f(double x)
```

```

{ const double pi = 3.14159; return x-sin(pi*x); }

double fder(double x) // derivative of f
{ const double pi = 3.14159; return 1.0-pi*cos(pi*x); }

double newtonmeth(double initial,double eps)
{
  double x0, x1;
  x1 = initial;
  do
  { x0 = x1; x1 = x0-f(x0)/fder(x0); }
  while(fabs(x1-x0) > eps);
  return x0;
}

int main(void)
{
  double initial = 0.5;
  double eps = 0.0001;
  double result = newtonmeth(initial,eps);
  cout << "result = " << result << endl;
  return 0;
}

```

1.1.14 Feigenbaum's Constant

In a number of mappings which depend on a bifurcation parameter r we find a period doubling cascade. We consider the bifurcation parameter values where period-doubling events occur. The limit of the ratio of distances between consecutive doubling values is *Feigenbaum's constant*. It has the value

$$4.669201609102990671853\dots$$

Mappings which show this transition are

$$x_{t+1} = rx_t(1 - x_t), \quad x_{t+1} = 1 - rx_t^2, \quad x_{t+1} = x_t^2 + r.$$

The C++ program `feigenbaum1.cpp` finds the Feigenbaum constant using the equation $x_{t+1} = x_t^2 + r$. The program shows the constant computed for two doubling cascades. The first one starts with the period 1 cardioid and the second starts with the period 3 cardioid. Newton's method is used to find the root of $x = x^2 + r$ iterated n times.

```

// feigenbaum1.cpp

#include <stdio.h>
#include <stdlib.h>
#include <math.h> // for fabs

```

```

double newton(long n,double c)
{
    double x, x1;
    double nc = c;
    double absx = 1.0;
    long i, j;
    j = 0;
    while((j < 7) && (absx > 1E-13))
    {
        ++j;
        x = 0.0; x1 = 0.0;
        for(i=0;i<n;i++) { x1 = 2.0*x1*x+1.0; x = x*x+nc; }
        nc -= x/x1; absx = fabs(x);
    }
    return nc;
}

void go(long n0,double a,double b)
{
    double f = 4.0;
    double tmp = a;
    double newc = a+(a-b)/f;
    double oldc = b;
    long n = 2*n0;
    for(int i=0;i<10;++i)
    {
        newc = newton(n,newc);
        f = (tmp-oldc)/(newc-tmp);
        printf("%.16lf %.16lf %.16lf %.16lf\n",oldc,tmp,newc,f);
        oldc = tmp; tmp = newc;
        newc += (newc-oldc)/f;
        n *= 2;
    }
}

int main(void)
{
    double a, b;
    long n;
    printf("c1          c2          c3");
    printf("          f: (c2-c1)/(c3-c2)");
    printf("\n");
    b = 0.0; a = -1.0; n = 2;
    go(n,a,b);
    printf("\n");
    a = -1.7728929033816238; b = -1.75487766624669276;
    n = 6;
}

```

```

go(n,a,b);
return 0;
}

```

1.1.15 Symbolic Dynamics

Consider a one-dimensional nonlinear map $f(x, r)$, which maps points from an interval I into the same interval

$$x_{t+1} = f(x_t, r), \quad x_t \in I$$

where r is a control parameter. The function f may have several monotone branches, divided by turning points, denoted symbolically by C_i (called also *critical points*). For smooth maps the derivative df/dx vanishes at $x = C_i$. The turning points C_i and the end points of the interval I divide I into subintervals I_i . We label each of I_i by a symbol S_i . By iterating the map, we obtain a numerical sequence

$$x_0, \quad x_1 = f(x_0), \quad x_2 = f(x_1), \quad \dots, \quad x_{t-1} = f(x_{t-2}), \quad x_t = f(x_{t-1}) \dots$$

We juxtapose the numerical sequence with a symbolic sequence and call it by the number x_0 which has originated the numerical sequence

$$x_0 = \sigma_0 \sigma_1 \sigma_2 \dots \sigma_{n-1} \sigma_n \dots$$

where σ_i stands for one of the symbols S_j or C_j , depending on whether x_t belongs to the corresponding subinterval or coincides with a turning point. If we want to reverse the numerical sequence, expressing x_0 through x_t , we must indicate which of the monotone branches of f has been used at each iteration. To do so, we attach a subscript σ to f . The symbol σ is chosen by the argument of f

$$x_0, \quad x_1 = f_{\sigma_0}(x_0), \quad x_2 = f_{\sigma_1}(x_1), \quad \dots, \quad x_t = f_{\sigma_{t-1}}(x_{t-1}), \quad \dots$$

Now we are in a position to reverse this sequence. We obtain

$$x_0 = f_{\sigma_0}^{-1} \circ f_{\sigma_1}^{-1} \circ \dots \circ f_{\sigma_{t-1}}^{-1}(x_t).$$

To simplify the notation, we denote each inverse monotone branch f_{σ}^{-1} by its subscript, i.e. we define

$$\sigma(y) \equiv f_{\sigma}^{-1}(y)$$

where σ is one of the symbols S_i . For example, the logistic map

$$x_{t+1} = 1 - rx_t^2, \quad x_t \in [-1, 1], \quad r \in (0, 2)$$

has two inverse branches

$$R(y) = \sqrt{(1-y)/r}, \quad L(y) = -\sqrt{(1-y)/r}.$$

Now we find

$$x_0 = \sigma_0 \circ \sigma_1 \circ \dots \circ \sigma_{t-1}(x_t).$$

Thus we found the number-symbol-inverse function correspondence.

The logistic map $f(x, r) = 1 - rx^2$ is a unimodal map. Since $df/dx = -2rx$ we find that $C = 0$ is the only critical point. The iterate of the critical point $C = 0$ leads to the rightmost point $f(C)$ on the interval that one can ever reach by iterating the map from any point on the interval. The point $f(C)$ thus starts a special symbolic sequence, called the *kneading sequence*. The kneading sequence is named $f(C)$ and sometimes denoted by K (for kneading). For instance, at the parameter value $r = 1.85$ the logistic map has a kneading sequence

$$K \equiv f(C) = RLLRLRLRRL \dots$$

The second iterate of C i.e. $f^{(2)}(C)$, gives the leftmost point that one can ever reach by iterating the map twice starting from any point on the interval. All the interesting dynamics takes place on the subinterval

$$[f^{(2)}(C), f(C)]$$

of I . Once a point is in this subinterval, its iterates can never get out. Therefore, this subinterval defines an *invariant dynamical range*. In principle, one can choose an initial point outside this subinterval, but after a trivial transient (in fact, a few iterations), it will fall into the invariant dynamical range. For maps with multiple critical points each C_i leads to a kneading sequence; one collects the dynamical range of all turning points and finds the overall range of interest dynamics. In general one concentrates on the invariant dynamical range only, neglecting trivial transients. Each kneading sequence is represented by a number. This number may be taken as the parameter for the map. This happens to be very convenient for maps with multiple critical points. In other words, one can parameterize a map by its independently changing kneading sequences. In the C++ program we find the kneading sequence for the logistic map with $r = 37/20$.

```
// kneading.cpp

#include <iostream>
#include <string>
#include "rational.h"
#include "verylong.h"
using namespace std;

int main(void)
{
    int n = 12;
    string s = "";
    Rational<Verylong> one("1"), zero("0");
    Rational<Verylong> r("37/20"); // control parameter
    Rational<Verylong> x = zero; // initial value 0

    if(x < zero) s = s + "L";
```

```

    if(x == zero) s = s + "C";
    if(x > zero) s = s + "R";
    for(int i=1;i<(n-1);i++)
    {
        x = one-r*x*x;
        if(x < zero) s += "L";
        if(x == zero) s += "C";
        if(x > zero) s += "R";
    }
    cout << "symbolic sequence = " << s;
    return 0;
}

```

The output is given by `symbolic sequence = CRLRLRLRLRL`.

1.1.16 Chaotic Repeller

Consider the logistic map

$$x_{t+1} = f(x_t) = rx_t(1 - x_t)$$

where $r > 4.0$, for example $r = 4.1$. Thus, for example, if $x_0 = 0.5$, then $f(x_0) = 1.025 > 1.0$ and $f(f(x_0)) = -0.1050625$. We find that $f^{(n)}(x_0)$ tends to $-\infty$ if $n \rightarrow \infty$. Letting $s = r/4 - 1$ the map has a gap of size $\sqrt{s/(1+s)}$. In this gap $f(x) > 1.0$. Initial conditions chosen from this gap maps out of the unit interval $[0, 1]$ in one iteration and goes to $-\infty$. Almost all initial conditions in the unit interval eventually escape from it except for a set of Lebesgue measure zero. This set, by construction, is a fractal Cantor set. A *chaotic repeller* is a set of points on the attractor that never visit the gap. The chaotic repeller can be used for encoding digital information (Lai [64]).

```

// Repeller.cpp

#include <iostream>
using namespace std;

int main(void)
{
    int count = 0;
    double x0 = 1.0/3.0;
    double x1;
    do
    {
        x1 = 4.1*x0*(1.0-x0);
        count++;
        x0 = x1;
    } while(x0 <= 1.0);
    cout << "x0 = " << x0 << endl;
}

```

```

cout << "count = " << count << endl;
return 0;
}

```

1.1.17 Chaos and Encoding

One-dimensional chaotic maps can be used for the communication of information. We consider two techniques of encoding bit strings using chaos: reverse interval mapping and variable bit length encoding (Hardy and Sabatta [47]).

We consider iteration of one-dimensional maps $f : [0, 1] \rightarrow [0, 1]$ of the form,

$$x_{t+1} = rf(x_t), \quad t = 0, 1, \dots$$

where $r > 1$ and f is 1 to 1 and monotone on $[0, 0.5]$ and has the properties

$$f(0) = 0, \quad f(0.5) = 1, \quad f(1) = 0, \quad f(x) = f(1 - x).$$

Examples include the

Bell map:	$f_b(x) = (e^{-(x-0.5)^2} - e^{-0.25}) / (1 - e^{-0.25})$
Entropy map:	$f_e(x) = -x \log_2 x - (1 - x) \log_2 (1 - x)$
Logistic map:	$f_l(x) = 4x(1 - x)$
Tent map:	$f_t(x) = 1 - 2 x - 0.5 $
Sine map:	$f_s(x) = \sin(\pi x)$

The maps f_l and f_s are good candidates for encoding information. Let f^{-1} denote the inverse of f on $[0, 0.5]$. If we consider the map rf with $r > 1$, there exists two intervals

$$\left[0, f^{-1}(1/r)\right], \quad \left[1 - f^{-1}(1/r), 1\right],$$

where $0 \leq rf(x) \leq 1$. If we now consider points which remain on the unit interval under two iterations of the map, the two intervals are divided into four sub-intervals that remain on the unit interval after two successive maps under the map rf . As we continue in this manner, we construct a Cantor set. Any point on this Cantor set remains on the unit interval under successive iterations of the map, and thus any point not on this set will eventually leave the unit interval. We use orbits that are confined to these intervals to encode messages.

With *reverse interval mapping* one considers a method which encodes messages of the form

$$m = m_1 m_2 \cdots m_n \in \Sigma_2^*$$

where $\Sigma_2 := \{0, 1\}$ and

$$\Sigma_2^* := \Sigma_2 \cup (\Sigma_2 \times \Sigma_2) \cup (\Sigma_2 \times \Sigma_2 \times \Sigma_2) \cup \cdots.$$

Values on the interval $[0, 1]$ are associated with Σ_2 by the map $d : [0, 1] \rightarrow \Sigma_2$

$$d(x) = \begin{cases} 0, & 0 \leq x \leq \frac{1}{2} \\ 1, & \frac{1}{2} < x \leq 1 \end{cases}.$$

Initially we begin with any value x_0 outside the interval $[0, 1]$. For example $(1+r)/2$ is a convenient choice. This value marks the beginning of the message and the end of the decoding process. This point has two pre-images under the map rf

$$x_{1,0} = f^{-1}(x_0/r), \quad x_{1,1} = 1 - f^{-1}(x_0/r).$$

Each of these values lie on either side of $x = 1/2$. Thus we select the value of x_1 from $\{x_{1,0}, x_{1,1}\}$ that satisfies the requirement $d(x_1) = m_1$. We proceed in this manner until the maximum precision of the register storing the value x_i has been reached, or when all of the message has been encoded (i.e. after determining x_n). Thus

$$\begin{aligned} x_0 &:= \frac{1+r}{2} \\ x_1 &:= \begin{cases} f^{-1}(x_0/r) & m_1 = 0 \\ 1 - f^{-1}(x_0/r) & m_1 = 1 \end{cases} \\ x_2 &:= \begin{cases} f^{-1}(x_1/r) & m_2 = 0 \\ 1 - f^{-1}(x_1/r) & m_2 = 1 \end{cases} \\ &\vdots \end{aligned}$$

A naive approach is to assume that the register always has adequate precision (i.e. the message must be short enough). This value is then stored and the procedure is repeated. The following C++ function encodes a string of 0s and 1s as a value of type `double`. The function `fi` denotes the inverse f^{-1} of the map f used in the map rf .

```
double encode(string m,double r,double (*fi)(double))
{
    unsigned int i;
    // initial value: 0.5+0.5r > 1, since r > 1
    // consequently m does not lie in [0,1]
    double x = (1.0+r)/2;

    // for each bit 0/1 in the string m="01001..." from left to right
    for(i=0;i<m.length();i++)
    {
        // for a "0" bit choose the pre-image on the left half
        if(m[i]=='0') x = fi(x/r);
        // for a "1" bit choose the pre-image on the right half
        else x = 1.0-fi(x/r);
    }
    // this is the final pre-image f^(-n)(0.5+0.5r)=fi^(n)(0.5+0.5r)
    // for the sequence of pre-image choices given in m
    return x;
}
```

To decode the message stream we iterate the stored point under the map, at each iteration calculating $m_i = d(x_i)$ to resolve the bit value. This is repeated until the desired number of bits have been retrieved. The message is reconstructed in reverse. The C++ function to decode a value of type `double` to a string of 0s and 1s follows.

```
string decode(double x,double r,double (*f)(double))
{
  // in decoding we work from right to left (opposite to encoding)
  // i.e. every "0" or "1" must be appended on the left: m = "0" + m
  string m;

  // once we leave [0,1] there is no more message to decode
  while((0.0 <= x) && (x <= 1.0))
  {
    // the left pre-image corresponded to "0"
    if(x < 0.5) m = "0" + m;
    // the right pre-image corresponded to "1"
    else m = "1" + m;
    // iterate the map
    x = r*f(x);
  }
  return m;
}
```

Consider the logistic map $f_l(x)$. The arguments below can be extended to any of the other maps listed above. Assume we wish to encode a 16-bit data stream in a single orbit under the logistic map with parameter $r = 1.025$. There are $2^{16} = 65536$ disjoint intervals which remain a subset of $[0, 1]$ under the map $(rf_l)^{(16)}(x)$. These can be found by applying $(rf_l)^{-1}(x)$ to $[0, 1]$, which yields two disjoint intervals. We apply $(rf_l)^{-1}(x)$ to each of the resulting 2 disjoint intervals to obtain 4 disjoint intervals and so on. We apply the inverse $(rf_l)^{-1}(x)$ 16 times. The intervals are numbered from 0 to 65535 according to their relative positions in $[0, 1]$.

In *variable bit length encoding* instead of iterating a single number we iterate an interval

$$[a, b] \subset (f^{-1}(1/r), 0.5)$$

or

$$[a, b] \subset (0.5, 1 - f^{-1}(1/r)).$$

Thus we begin with an interval $[a, b]$ which will iterate out of $[0, 1]$ under rf . A convenient choice in this case is

$$w := 0.1, \quad x_0 := f^{-1}(1/r), \quad a = (1 - w)x_0 + \frac{w}{2}, \quad b = wx_0 + \frac{1 - w}{2}$$

where $0 < w < 0.5$ is a weight for the weighted average of $x_0 = f^{-1}(1/r)$ and 0.5 used to determine a and b . Smaller w is preferred, since it yields a larger initial interval, which will subsequently shrink. In the implementation below we chose

$w = 0.1$. One applies the reverse interval mapping for a and b up to a precision limit, given by the interval length $|a - b|$, by determining when the intervals found at each step become smaller in length than $\epsilon > 0$. We need only store a , since b is only used to determine whether we are still outside of the precision limit. If more bits need to be encoded we begin reverse interval mapping with a new initial interval for the remainder of the bits. Repeating this process until all bits are encoded we obtain a sequence of real numbers. Each number should be decoded with reverse interval mapping to obtain that part of the original bit sequence. Note that the number of bits decoded for each real value need not be the same over the sequence, hence the name of the technique. We find successive intervals $[a_j, b_j]$ as follows

$$a_0 = (1 - w)x_0 + \frac{w}{2}, \quad b_0 = wx_0 + \frac{1 - w}{2}$$

$$a_1 = \begin{cases} f^{-1}(a_0/r) & m_1 = 0 \\ 1 - f^{-1}(b_0/r) & m_1 = 1 \end{cases}, \quad b_1 = \begin{cases} f^{-1}(b_0/r) & m_1 = 0 \\ 1 - f^{-1}(a_0/r) & m_1 = 1 \end{cases}$$

$$a_2 = \begin{cases} f^{-1}(a_1/r) & m_2 = 0 \\ 1 - f^{-1}(b_1/r) & m_2 = 1 \end{cases}, \quad b_2 = \begin{cases} f^{-1}(b_1/r) & m_2 = 0 \\ 1 - f^{-1}(a_1/r) & m_2 = 1 \end{cases}$$

and continue until $b_n - a_n < \epsilon$. Note that $m_j = 1$ causes a swap in the roles of a_j and b_j so that the absolute value is always given by $b_j - a_j$, and so that the resulting interval lies in $[0.5, 1]$. Once the criteria $b_n - a_n < \epsilon$ is met, we store a_n at the end of the sequence of numbers representing the data and begin once again by redefining a_n and b_n

$$a_n := (1 - w)x_0 + \frac{w}{2}, \quad b_n := wx_0 + \frac{1 - w}{2}$$

$$a_{n+1} = \begin{cases} f^{-1}(a_n/r) & m_{n+1} = 0 \\ 1 - f^{-1}(b_n/r) & m_{n+1} = 1 \end{cases}, \quad b_{n+1} = \begin{cases} f^{-1}(b_n/r) & m_{n+1} = 0 \\ 1 - f^{-1}(a_n/r) & m_{n+1} = 1 \end{cases}$$

$$a_{n+2} = \begin{cases} f^{-1}(a_{n+1}/r) & m_{n+2} = 0 \\ 1 - f^{-1}(b_{n+1}/r) & m_{n+2} = 1 \end{cases}, \quad b_{n+2} = \begin{cases} f^{-1}(b_{n+1}/r) & m_{n+2} = 0 \\ 1 - f^{-1}(a_{n+1}/r) & m_{n+2} = 1 \end{cases}$$

until $b_{n+k} - a_{n+k} < \epsilon$ and then continue the process until all the m_j values have been used. Thus the encoding now yields multiple real numbers each representing a portion of the message string, where each real number does not necessarily encode the same number of symbols. A C++ function for the encoding is given below.

```
vector<double> vl_encode(string m,double r,double (*fi)(double),
                        double eps)
{
double x0, a = 0.0, b = 0.0, w = 0.1;
vector<double> vd;
x0 = fi(1.0/r);
// for each bit 0/1 in the string m="01001..." from left to right
for(unsigned int i=0;i<m.length();i++)
{
// the interval has become too small we start from
// the initial values again to encode the remaining bits
if((b-a) < eps)
```

```

{
// create the interval [a,b] as a subset of  $(f^{-1}(1/r), 0.5)$ 
a = (1.0-w)*x0+w*0.5;
b = w*x0+(1.0-w)*0.5;
// push back initializes the next number (appends on the right)
// in the sequence to a, i.e. we add a number onto the end
// of the sequence
vd.push_back(a);
}
// vd.back() is the last number in the sequence
// (which we are still computing)
// for a "0" bit choose the pre-image on the left half
if(m[i]=='0') vd.back() = a;
// for a "1" bit choose the pre-image on the right half
else
{
vd.back() = 1.0-a; // since  $a < 0.5, 1-a > 0.5$ 
// after the following two statements we have
// [a,b] -> [1.0-b, 1.0-a]
// i.e. we change halves around 0.5:
// left --> right or right --> left
a = 1.0-b;
b = vd.back();
}
// find the pre-image of [a,b]
a = fi(a/r); b = fi(b/r);
}
return vd;
}

```

To decode we simply apply the decoding technique of reverse interval mapping to each real number found in the encoding process as demonstrated in the following C++ function.

```

string vl_decode(vector<double> vd, double r, double (*f)(double))
{
// in decoding we work from right to left (opposite to encoding)
// i.e. every "0" or "1" must be appended on the left: m = "0" + m
string m;

// vd.size() is the number of elements in the sequence
// that we have not yet decoded
while(vd.size() != 0)
{
// once we leave [0,1] there is no more message to decode
while((0.0 <= vd.back()) && (vd.back() <= 1.0))
{
// the left pre-image corresponded to "0"
if(vd.back() < 0.5) m = "0" + m;

```

```
// the right pre-image corresponded to "1"
else m = "1" + m;
vd.back() = r*f(vd.back());
}
// remove the right most number of the sequence
// still decoding right to left
vd.pop_back();
}
return m;
}
```

1.2 Two-Dimensional Maps

1.2.1 Introduction

Most of the two-dimensional maps (Arrowsmith [2], Steeb [100], [101]) we consider in this section are diffeomorphisms.

Let U be an open subset of \mathbf{R}^n . Then a function $\mathbf{g} : U \rightarrow \mathbf{R}$ is said to be of class C^r if it is r -fold continuously differentiable, $1 \leq r \leq \infty$. Let V be an open subset of \mathbf{R}^m and $\mathbf{g} : U \rightarrow V$. Given coordinates (x_1, \dots, x_n) in U and (y_1, \dots, y_m) in V , \mathbf{g} may be expressed of component functions $g_j : U \rightarrow \mathbf{R}$, where

$$y_j = g_j(x_1, \dots, x_n), \quad j = 1, \dots, m.$$

The map \mathbf{g} is called a C^r map if g_j is C^r for each $j = 1, \dots, m$. The map \mathbf{g} is said to be a *diffeomorphism* if it is a bijection and both \mathbf{g} and \mathbf{g}^{-1} are differentiable mappings. The map \mathbf{g} is called a C^k -diffeomorphism if both \mathbf{g} and \mathbf{g}^{-1} are C^k -maps.

Note that the bijection $\mathbf{g} : U \rightarrow V$ is a diffeomorphism if and only if $m = n$ and the *functional matrix* (also called *Jacobian matrix*) of partial derivatives

$$D\mathbf{g}(x_1, \dots, x_n) := \left(\frac{\partial g_i}{\partial x_j} \right)_{i,j=1}^n.$$

For $n = m = 2$ we have the functional matrix

$$\begin{pmatrix} \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 \\ \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 \end{pmatrix}.$$

If \mathbf{g} satisfies the definition above with \mathbf{g} and \mathbf{g}^{-1} continuous rather than differentiable, then \mathbf{g} is called a *homeomorphism*.

Example. The map $f : \mathbf{R} \rightarrow \mathbf{R}$, $f(x) = \sinh(x)$ is a diffeomorphism. ♣

Example. The map $f : \mathbf{R} \rightarrow \mathbf{R}$, $f(x) = x^3$ is not a diffeomorphism since its derivative vanishes at 0. ♣

Let U be an open subset of \mathbf{R}^n and $\mathbf{f} : U \rightarrow \mathbf{R}^n$ be a nonlinear diffeomorphism with an isolated fixed point at $\mathbf{x}^* \in U$. The *linearization* of \mathbf{f} at \mathbf{x}^* is given by the $n \times n$ matrix

$$D\mathbf{f}(\mathbf{x}^*) := \left[\frac{\partial f_i}{\partial x_j} \right]_{i,j=1}^n \Big|_{\mathbf{x}=\mathbf{x}^*}$$

where x_1, \dots, x_n are coordinates on U .

Definition. A fixed point \mathbf{x}^* of a diffeomorphism \mathbf{f} is said to be hyperbolic if the map $D\mathbf{f}(\mathbf{x}^*)$ is a hyperbolic, linear diffeomorphism.

Definition. A linear diffeomorphism $A : \mathbf{R}^n \rightarrow \mathbf{R}^n$ is said to be *hyperbolic* if it has no eigenvalues with modulus equal to unity.

The following theorems allow us to obtain valuable information from $D\mathbf{f}(\mathbf{x}^*)$.

Theorem. (Hartman-Grobman) Let \mathbf{x}^* be a hyperbolic fixed point of the diffeomorphism $\mathbf{f} : U \rightarrow \mathbf{R}^n$. Then there is a neighbourhood $N \subset U$ of \mathbf{x}^* and a neighbourhood $N' \subseteq \mathbf{R}^n$ containing the origin such that $\mathbf{f}|N$ is topologically conjugate to $D\mathbf{f}(\mathbf{x}^*)|N'$.

It follows that there are $4n$ topological types of hyperbolic fixed point for diffeomorphisms $\mathbf{f} : U \rightarrow \mathbf{R}^n$.

Theorem. (Invariant Manifold) Let $\mathbf{f} : U \rightarrow \mathbf{R}^n$ be a diffeomorphism with a hyperbolic fixed point at $\mathbf{x}^* \in U$. Then on a sufficiently small neighbourhood $N \subseteq U$ of \mathbf{x}^* , there exist local stable and unstable manifolds,

$$W_{loc}^s(\mathbf{x}^*) := \{ \mathbf{x} \in U \mid \mathbf{f}^{(t)}(\mathbf{x}) \rightarrow \mathbf{x}^* \text{ as } t \rightarrow \infty \}$$

$$W_{loc}^u(\mathbf{x}^*) := \{ \mathbf{x} \in U \mid \mathbf{f}^{(t)}(\mathbf{x}) \rightarrow \mathbf{x}^* \text{ as } t \rightarrow -\infty \}$$

of the same dimensions as E^s and E^u for $D\mathbf{f}(\mathbf{x}^*)$ and tangent to them at \mathbf{x}^* .

This theorem allows us to define global stable and unstable manifolds at \mathbf{x}^* by

$$W^s(\mathbf{x}^*) := \bigcup_{m \in \mathbf{Z}^+} \mathbf{f}^{(-m)}(W_{loc}^s(\mathbf{x}^*))$$

$$W^u(\mathbf{x}^*) := \bigcup_{m \in \mathbf{Z}^+} \mathbf{f}^{(m)}(W_{loc}^u(\mathbf{x}^*)).$$

The behaviour of the stable and unstable manifold $W^s(\mathbf{x}^*)$ and $W^u(\mathbf{x}^*)$ reflects in the complexity of the dynamics of the map \mathbf{f} . In particular, if $W^s(\mathbf{x}^*)$ and $W^u(\mathbf{x}^*)$ meet transversely at one point, they must do so infinitely many times and a homoclinic tangle results. The theorems given above have an extension to the periodic point of \mathbf{f} . Let \mathbf{x}^* belong to a q -cycle of \mathbf{f} then it is said to be a *hyperbolic periodic point* of \mathbf{f} if it is a hyperbolic fixed point of the q -th iterated map $\mathbf{f}^{(q)}$. The orbit of \mathbf{x}^* under \mathbf{f} is referred to as a *hyperbolic periodic orbit* and its topological type is determined by that of the corresponding fixed point of $\mathbf{f}^{(q)}$. Moreover, information about stable and unstable manifolds at each point of the q -cycle can be obtained by applying the theorem to $\mathbf{f}^{(q)}$.

The *Hopf bifurcation theorem* for maps in the plane $\mathbf{f}_r : \mathbf{R}^2 \rightarrow \mathbf{R}^2$, where r is the bifurcation parameter, is as follows.

Theorem. (Hopf bifurcation theorem) Let $\mathbf{f}(r, \mathbf{x})$ be a one-parameter family of maps in the plane satisfying:

a) An isolated fixed point $\mathbf{x}^*(r)$ exists.

b) The map \mathbf{f}_r is C^k ($k \geq 3$) in the neighbourhood of $(\mathbf{x}^*(r_0); r_0)$.

c) The Jacobian matrix $D_{\mathbf{x}}\mathbf{f}(\mathbf{x}^*(r); r)$ possesses a pair of complex, simple eigenvalues

$$\lambda(r) = e^{\alpha(r)+i\omega(r)}$$

and $\bar{\lambda}(r)$, such that the critical value $r = r_0$

$$|\lambda(r_0)| = 1, \quad (\lambda(r_0))^3 \neq 1, \quad (\lambda(r_0))^4 \neq 1, \quad \frac{d|\lambda(r)|}{dr}(r = r_0) > 0.$$

(Existence) Then there exists a real number $\epsilon_0 > 0$ and a C^{k-1} function such that

$$r(\epsilon) = r_0 + r_1\epsilon + r_3\epsilon^3 + O(\epsilon^4)$$

such that for each $\epsilon \in (0, \epsilon_0]$ the map \mathbf{f}_r has an invariant manifold $H(r)$, i.e. $\mathbf{f}(H(r); r) = H(r)$. The manifold $H(r)$ is C^r diffeomorphic to a circle and consists of points at a distance $O(|r|^{1/2})$ of $\mathbf{x}^*(r)$, for $r = r(\epsilon)$.

(Uniqueness) Each compact invariant manifold close to $\mathbf{x}^*(r)$ for $r = r(\epsilon)$ is contained in $H(r) \cup \{0\}$.

(Stability) If $r_3 < 0$ (respectively $r_3 > 0$) then for $r > 0$ (respectively $r < 0$), the fixed point $\mathbf{x}^*(r(\epsilon))$ is stable (respectively unstable) and for $r < 0$ (respectively $r > 0$) the fixed point $\mathbf{x}^*(r(\epsilon))$ is unstable (respectively stable) and the surrounding manifold $H(r(\epsilon))$ is attracting (respectively repelling). When $r_3 < 0$ (respectively $r_3 > 0$) the bifurcation at $r = r(\epsilon)$ is said to be *supercritical* (respectively *subcritical*).

Example. Consider the two-dimensional map

$$f_1(x_1, x_2) = rx_1(3x_2 + 1)(1 - x_1), \quad f_2(x_1, x_2) = rx_2(3x_1 + 1)(1 - x_2)$$

and $r \in \mathbf{R}$. There are fixed points on the diagonal, namely $\mathbf{x}_0^* = (0, 0)$ and for $r \neq 0$

$$\mathbf{x}_1^* = ((r - 1)/r, 0), \quad \mathbf{x}_2^* = (0, (r - 1)/r, 0)$$

and on the diagonal ($r \geq 3/4$)

$$\begin{aligned} \mathbf{x}_3^* &= (1/3 - \sqrt{4 - 3/r}/3, 1/3 - \sqrt{4 - 3/r}) \\ \mathbf{x}_4^* &= (1/3 + \sqrt{4 - 3/r}/3, 1/3 + \sqrt{4 - 3/r}). \end{aligned}$$

To study Hopf bifurcation we consider the fixed points on the diagonal. Notice that these fixed points exist only for $r \geq 3/4$. For $r > 3/4$ a stable period-2 orbit exists. This period-2 orbit loses stability via a Hopf bifurcation which occurs at $r = r_0$, where $r_0 = 0.957$ and gives rise to a stable limit cycle for $r \in [r_0, r_0 + \delta)$ for some $\delta > 0$. ♣

We consider the Hénon map, the Lozi map, standard map, the Ikeda laser map and a coupled logistic map.

1.2.2 Phase Portrait

For a two-dimensional map

$$x_{1t+1} = f_1(x_{1t}, x_{2t}), \quad x_{2t+1} = f_2(x_{1t}, x_{2t})$$

where $t = 0, 1, 2, \dots$ we can plot the points (x_{1t}, x_{2t}) for $t = 0, 1, 2, \dots$ in the (x_1, x_2) plane \mathbf{R}^2 . This is called a *phase portrait*. We also use the notation $x = x_1$ and $y = x_2$. As our first example we consider the *Hénon map* $\mathbf{f} : \mathbf{R}^2 \rightarrow \mathbf{R}^2$ which is given by

$$\mathbf{f}(x, y) := (y + 1 - ax^2, bx)$$

where a and b are bifurcation parameters with $b \neq 0$. The Hénon map is the most studied two-dimensional map with chaotic behaviour. The map can also be written as a system of difference equations

$$x_{t+1} = 1 + y_t - ax_t^2, \quad y_{t+1} = bx_t$$

where $t = 0, 1, 2, \dots$. The map is invertible if $b \neq 0$. The inverse map is given by

$$x_t = \frac{1}{b}y_{t+1}, \quad y_t = x_{t+1} - 1 + \frac{a}{b^2}y_{t+1}^2.$$

In order to visualize the action of the map, note that vertical lines are mapped to horizontal lines, while for $a > 0$ horizontal lines map to parabolas opening to the left.

In the C++ program `henon.cpp` we evaluate the phase portrait (x_t, y_t) for $a = 1.4$ and $b = 0.3$. The data are written to a file named `henon.dat`. Then we use GNU plot to display the phase portrait. In the Java program `Henon.java` the graphics is included and the phase portrait is displayed.

```
// henon.cpp

#include <fstream>
using namespace std;

int main(void)
{
    ofstream data("henon.dat");
    const int T = 2000;          // number of iterations
    double x0 = 0.1, y0 = 0.3;  // initial values
    double x1, y1;
    for(int t=0;t<T;t++)
    {
        x1 = 1.0+y0-1.4*x0*x0;  y1 = 0.3*x0;
        data << x1 << " " << y1 << "\n";
        x0 = x1; y0 = y1;
    }
    data.close();
    return 0;
}
```

The data in the file "henon.dat" can now be used to display the phase portrait using GNU-plot. The command is `plot 'henon.dat' with dots`.

```
// Henon.java

import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class Henon extends Frame
{
    public Henon()
    {
        setSize(600,500);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent event)
        { System.exit(0); }}); }

    public void paint(Graphics g)
    {
        double x1, y1;
        double x = 0.0, y = 0.0; // initial values
        int T = 4000; // number of iterations
        for(int t=0;t<T;t++)
        {
            x1 = x; y1 = y;
            x = 1.0+y1-1.4*x1*x1; y = 0.3*x1;
            int mx = (int) Math.floor(200*x+250+0.5);
            int ny = (int) Math.floor(200*y+150+0.5);
            g.drawLine(mx,ny,mx,ny);
        }
    }

    public static void main(String[] args)
    {
        Frame f = new Henon(); f.setVisible(true);
    }
}
```

The *Lozi map* $\mathbf{f} : \mathbf{R}^2 \rightarrow \mathbf{R}^2$ is a piecewise linear map which has been introduced to simplify the Hénon map. It is given by

$$\mathbf{f}(x, y) = (1 + y - a|x|, bx)$$

where $b > 0$. It can be shown that if

$$b \in (0, 1), \quad a > 0, \quad 2a + b < 4, \quad b < \frac{a^2 - 1}{2a + 1}, \quad a\sqrt{2} > b + 2$$

then there is a hyperbolic fixed point H of saddle type given by

$$x^* = \frac{1}{a+1-b}, \quad y^* = bx^*$$

such that the strange attractor is the closure of their unstable manifold.

In the Java program for the Hénon model `Henon.java` we replace the line

```
x = 1.0+y1-1.4*x1*x1;  y = 0.3*x1;
```

by

```
x = 1.0+y1-a*Math.fabs(x1);  y = b*x1;
```

The parameter values for the Lozi map are $a = 1.7$, $b = 0.4$. For these parameter values the system shows a strange attractor. For these values the conditions given above are satisfied.

The following C++ program finds the largest and smallest values in the x and y direction of the Hénon attractor for the parameter values $a = 1.4$ and $b = 0.3$.

```
// maxmin.cpp

#include <iostream>
using namespace std;

void maxminxy(double* x,double* y,double& xmax,double& xmin,
              double& ymax,double& ymin,int T)
{
    xmax = x[0]; xmin = x[0];  ymax = y[0]; ymin = y[0];
    for(int t=1;t<T;t++)
    {
        if(x[t] < xmin) xmin = x[t];
        if(x[t] > xmax) xmax = x[t];
        if(y[t] < ymin) ymin = y[t];
        if(y[t] > ymax) ymax = y[t];
    }
}

int main(void)
{
    unsigned long T = 10000;          // number of iterations
    double* x = new double[T]; double* y = new double[T];
    x[0] = 1.161094; y[0] = -0.09541356; // initial values
    for(int t=0;t<(T-1);t++)
    {
        x[t+1] = 1.0+y[t]-1.4*x[t]*x[t]; y[t+1] = 0.3*x[t];
    }
}
```

```

double xmax, xmin, ymax, ymin;
maxminxy(x,y,xmax,xmin,ymax,ymin,T);
cout << "xmax= " << xmax << endl; cout << "xmin= " << xmin << endl;
cout << "ymax= " << ymax << endl; cout << "ymin= " << ymin << endl;
delete[] x; delete[] y;
return 0;
}

```

The *standard map* is defined as

$$\begin{aligned}
 I_{t+1} &= I_t + k \sin(\theta_t) \\
 \theta_{t+1} &= \theta_t + I_t + k \sin(\theta_t) \equiv \theta_t + I_{t+1}
 \end{aligned}$$

where $0 \leq \theta < 2\pi$. The quantities I , θ are the *action-angle variables*. It can be derived from the Hamilton function

$$H(p_\theta, \theta) = \frac{p_\theta^2}{2} + k \cos \theta \sum_n \delta(t - n)$$

of a one-dimensional periodically *kicked rotor*. The standard map can be considered as a discrete Hamilton system. We have

$$\det \begin{pmatrix} \frac{\partial f_1}{\partial I} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial I} & \frac{\partial f_2}{\partial \theta} \end{pmatrix} = 1.$$

This map displays all three types of orbits: periodic cycles, KAM tori and chaotic orbits. The first two types of orbits (the regular ones) dominate the (I, θ) phase space for small k ($k \ll 1$). KAM tori extending over the entire θ interval ($0 \leq \theta < 2\pi$) divide the (I, θ) phase plane into disconnected regions: orbits in one region cannot cross the bounding KAM tori into the other regions, so that the variations in I are bounded. For

$$k \geq k_c \approx 0.9716$$

the bounding KAM tori break, making it possible for chaotic orbits to be unbounded in the I direction. At k_c the last, most robust KAM torus with the winding number

$$w = 2\pi(\sqrt{5} - 1)/2$$

(and the other equivalent KAM tori of the standard map with winding numbers

$$\pm w + 2\pi n$$

(n integer)) disintegrates. It corresponds to a nonanalytic continuous curve, exhibiting fractal self-similar structure. The Java program `Standard.java` displays the phase portrait for $k = 0.8$.

```

// Standard.java

import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class Standard extends Frame
{
    public Standard()
    {
        setSize(600,500);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent event)
        { System.exit(0); }}); }

    public void paint(Graphics g)
    {
        int T = 10000;           // number of iterations
        double I = 0.5, theta = 0.8; // initial values
        double I1, theta1;
        for(int t=0;t<T;t++)
        {
            I1 = I;  theta1 = theta;
            I = I1+0.8*Math.sin(theta1);
            theta = theta1+I1+0.8*Math.sin(theta1);
            if(theta > 2*Math.PI) theta = theta-2.0*Math.PI;
            if(theta < 0.0) theta = theta+2.0*Math.PI;
            int m = (int) Math.floor(90*I+200+0.5);
            int n = (int) Math.floor(90*theta+10+0.5);
            g.drawLine(n,m,n,m);
        }
    }

    public static void main(String[] args)
    {
        Frame f = new Standard(); f.setVisible(true);
    }
}

```

Optical bistability only represents the simplest amongst the large variety of dynamical behaviours which can occur in passive nonlinear optical cavities. In particular, temporal instabilities leading to various forms of self-oscillations and chaos have been identified. Ikeda predicted the occurrence of period doubling cascades and chaos treating the cavity dynamics by means of a nonlinear mapping of the complex field amplitude. Let z be a complex number. The *Ikeda laser map* $f : \mathbf{C} \rightarrow \mathbf{C}$ is given by

$$f(z) = \rho + c_2 z \exp\left(i\left(c_1 - \frac{c_3}{1 + |z|^2}\right)\right).$$

The real bifurcation parameters are ρ , c_1 , c_2 and c_3 . With $z = x + iy$ and $x, y \in \mathbf{R}$ we can write the map as a system of difference equations

$$x_{t+1} = \rho + c_2(x_t \cos(\tau_t) - y_t \sin(\tau_t)), \quad y_{t+1} = c_2(x_t \sin(\tau_t) + y_t \cos(\tau_t))$$

where

$$\tau_t := c_1 - \frac{c_3}{1 + x_t^2 + y_t^2}$$

and $t = 0, 1, 2, \dots$. In the Java program `Ikeda.java` we evaluate the phase portrait (x_t, y_t) . The parameter values are $c_1 = 0.4$, $c_2 = 0.9$, $c_3 = 9.0$, $\rho = 0.85$.

```
// Ikeda.java

import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class Ikeda extends Frame
{
    public Ikeda()
    {
        setSize(400,300);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent event)
        { System.exit(0); }}); }

    public void paint(Graphics g)
    {
        int T = 20000;           // number of iterations
        double x = 0.5, y = 0.5; // initial values
        double x1, y1;
        double c1 = 0.4, c2 = 0.9, c3 = 9.0;
        double rho = 0.85;
        for(int t=0;t<T;t++)
        {
            x1 = x;  y1 = y;
            double taut = c1-c3/(1.0+x1*x1+y1*y1);
            x = rho+c2*x1*Math.cos(taut)-y1*Math.sin(taut);
            y = c2*(x1*Math.sin(taut)+y1*Math.cos(taut));
            int m = (int) Math.floor(90*x+200+0.5);
            int n = (int) Math.floor(90*y+200+0.5);
            g.drawLine(m,n,m,n);
        }
    }

    public static void main(String[] args)
    {
        Frame f = new Ikeda(); f.setVisible(true);
    }
}
```

}
}

The *coupled logistic map* is given by

$$x_{t+1} = rx_t(1 - x_t) + e(y_t - x_t), \quad y_{t+1} = ry_t(1 - y_t) + e(x_t - y_t)$$

where $t = 0, 1, 2, \dots$ and r and e are bifurcation parameters with $1 \leq r \leq 4$. For $e = 0$ we have two uncoupled logistic equations. The four fixed points are given by

$$(x^*, y^*) = (0, 0), \quad (x^*, y^*) = \left(\frac{r-1}{r}, \frac{r-1}{r} \right)$$

$$(x^*, y^*) = (a_+, a_-), \quad (x^*, y^*) = (a_-, a_+)$$

where

$$a_{\pm} = \frac{1}{2r} \left[(r-1-2e) \pm \sqrt{(r-1-2e)(r-1+2e)} \right].$$

Depending on the initial conditions and the parameter values one can find the following behaviour: (i) orbits tend to a fixed point, (ii) periodic behaviour, (iii) quasiperiodic behaviour, (iv) chaotic behaviour, (v) hyperchaotic behaviour and (vi) x_t and y_t explode, i.e. for a finite time the state variables x_t and (or) y_t tend to infinity. There is numerical evidence of hyperchaos for $r = 3.70$, $e = 0.06$. In the Java program `Couplog.java` we calculate the phase portrait (x_t, y_t) for these parameter values.

// Couplog.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Graphics;

public class Couplog extends Frame
{
    public Couplog()
    {
        setSize(400,300);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent event)
          { System.exit(0); }}); }

    public void paint(Graphics g)
    {
        int T = 60000;           // number of iterations
        double r = 3.7, e = 0.06; // control parameters
        double x = 0.1, y = 0.2; // initial values
        double x1, y1;
        for(int t=0;t<T;t++)
        {
```

```

x1 = x; y1 = y;
x = r*x1*(1.0-x1)+e*(y1-x1);
y = r*y1*(1.0-y1)+e*(x1-y1);
int m = (int) Math.floor(400*x+150+0.5);
int n = (int) Math.floor(400*y+150+0.5);
g.drawLine(m,n,m,n);
}
}

public static void main(String[] args)
{
Frame f = new Couplog(); f.setVisible(true);
}
}

```

1.2.3 Fixed Points and Stability

Given a two-dimensional map

$$x_{1t+1} = f_1(x_{1t}, x_{2t}), \quad x_{2t+1} = f_2(x_{1t}, x_{2t}).$$

The *fixed points* (x_1^*, x_2^*) are defined as the solution of the equations

$$f_1(x_1^*, x_2^*) = x_1^*, \quad f_2(x_1^*, x_2^*) = x_2^*.$$

As an example we consider the *Hénon map*

$$x_{t+1} = 1 + y_t - ax_t^2, \quad y_{t+1} = bx_t$$

where a and b are bifurcation parameters and $t = 0, 1, 2, \dots$. For $a > 0$ and $1 > b > 0$ the map has two fixed points

$$x^* = \frac{(b-1) \pm \sqrt{(1-b)^2 + 4a}}{2a}, \quad y^* = bx^*.$$

These fixed points are real for

$$a > a_0 = \frac{(1-b)^2}{4}.$$

The map has been studied in detail for $a = 1.4$ and $b = 0.3$. For these values the fixed points are unstable. There is numerical evidence that the map shows chaos for these parameter values. In the C++ program we determine the stability of the fixed points for $a = 1.4$ and $b = 0.3$. Since

$$\frac{\partial f_1}{\partial x} = -2ax, \quad \frac{\partial f_1}{\partial y} = 1, \quad \frac{\partial f_2}{\partial x} = b, \quad \frac{\partial f_2}{\partial y} = 0$$

we obtain the characteristic equation

$$\lambda_{1,2} = -ax^* \pm \sqrt{b + a^2x^{*2}}.$$

```

// henonstability.cpp

#include <iostream>
#include <cmath>      // for sqrt
using namespace std;

int main(void)
{
    double a = 1.4, b = 0.3; // parameter values

    // first fixed point
    double xf1 = ((b-1.0)+sqrt((1.0-b)*(1.0-b)+4.0*a))/(2.0*a);
    double yf1 = b*xf1;

    double lambda1 = -a*xf1+sqrt(b+a*a*xf1*xf1);
    double lambda2 = -a*xf1-sqrt(b+a*a*xf1*xf1);
    cout << "lambda1 for fixpoint 1 = " << lambda1 << endl;
    cout << "lambda2 for fixpoint 1 = " << lambda2 << endl;

    // second fixed point
    double xf2 = ((b-1.0)-sqrt((1.0-b)*(1.0-b)+4.0*a))/(2.0*a);
    double yf2 = b*xf2;

    double lambda3 = -a*xf2+sqrt(b+a*a*xf2*xf2);
    double lambda4 = -a*xf2-sqrt(b+a*a*xf2*xf2);
    cout << "lambda3 for fixpoint 2 = " << lambda3 << endl;
    cout << "lambda4 for fixpoint 2 = " << lambda4 << endl;
    return 0;
}

```

1.2.4 Liapunov Exponents

Consider a system of difference equations

$$x_{t+1} = f_1(x_t, y_t), \quad y_{t+1} = f_2(x_t, y_t)$$

where we assume that f_1 and f_2 are smooth functions. Then the variational equation is given by

$$\begin{aligned}
 u_{t+1} &= \frac{\partial f_1}{\partial x}(x = x_t, y = y_t)u_t + \frac{\partial f_1}{\partial y}(x = x_t, y = y_t)v_t \\
 v_{t+1} &= \frac{\partial f_2}{\partial x}(x = x_t, y = y_t)u_t + \frac{\partial f_2}{\partial y}(x = x_t, y = y_t)v_t.
 \end{aligned}$$

The maximal one-dimensional Liapunov exponent is given by

$$\lambda = \lim_{T \rightarrow \infty} \frac{1}{T} \ln(|u_T| + |v_T|).$$

As an example consider the Hénon map described by

$$x_{t+1} = 1 + y_t - ax_t^2, \quad y_{t+1} = bx_t$$

where a and b are bifurcation parameters and $t = 0, 1, 2, \dots$. Since

$$f_1(x, y) = 1 + y - ax^2, \quad f_2(x, y) = bx$$

we obtain the variational equation

$$u_{t+1} = -2ax_t u_t + v_t, \quad v_{t+1} = bu_t.$$

For the parameter value $a = 1.4$ and $b = 0.3$ we find from the numerical analysis that the (maximal) one-dimensional Liapunov exponent is given by $\lambda \approx 0.42$. There is numerical evidence that the map shows chaos for these parameter values.

```
// henonliapunov.cpp
```

```
#include <iostream>
#include <cmath> // for fabs, log
using namespace std;

double f1(double x,double y)
{ double a = 1.4; return 1.0+y-a*x*x; }

double f2(double x,double y)
{ double b = 0.3; return b*x; }

double vf1(double x,double y,double u,double v)
{ double a = 1.4; return -2.0*a*x*u+v; }

double vf2(double x,double y,double u,double v)
{ double b = 0.3; return b*u; }

int main(void)
{
    int T = 1000; // number of iterations
    double x = 0.1, y = 0.2; // initial values
    double u = 0.5, v = 0.5; // initial values
    double x1, y1, u1, v1;
    for(int t=0;t<T;t++)
    {
        x1 = x; y1 = y; u1 = u; v1 = v;
        x = f1(x1,y1); y = f2(x1,y1);
        u = vf1(x1,y1,u1,v1); v = vf2(x1,y1,u1,v1);
    }
    double lambda = log(fabs(u) + fabs(v))/((double) T);
    cout << "lambda = " << lambda << endl;
    return 0;
}
```

1.2.5 Correlation Integral

Dissipative dynamical systems (for example the Hénon map) which exhibit chaotic behaviour often have an attractor in phase space which is strange. Strange attractors are typically characterized by a fractal dimension D which is smaller than the number of degrees of freedom F , $D < F$. Among the fractal dimensions we have the capacity and the Hausdorff dimension. These fractal dimensions have been the most commonly used measure of the strangeness of attractors. Another measure is obtained by considering correlations between points of a long-time series on the attractor. Denote the T points of such a long-time series by

$$\{\mathbf{x}_i\}_{i=1}^T \equiv \{\mathbf{x}(t + i\tau)\}_{i=1}^T$$

where τ is an arbitrary but fixed time increment. The definition of the *correlation integral* is

$$C(r) := \lim_{T \rightarrow \infty} \frac{1}{T^2} \sum_{\substack{i,j=1 \\ i \neq j}}^T H(r - \|\mathbf{x}_i - \mathbf{x}_j\|)$$

where $H(x)$ is the *Heaviside function*, i.e.

$$H(x) := \begin{cases} 1 & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and $\|\dots\|$ denotes the Euclidean norm. The function $C(r)$ behaves as a power of r for small r

$$C(r) \propto r^\nu.$$

The exponent ν is called the *correlation dimension*. Moreover, the exponent ν is closely related to the capacity D .

In the C++ program `henoncorrelation.cpp` we evaluate the $C(r)$ for the Hénon map

$$x_{t+1} = 1 + y_t - ax_t^2, \quad y_{t+1} = bx_t$$

with parameter values $a = 1.4$ and $b = 0.3$ and $t = 0, 1, 2, \dots$. From $C(r)$ we find that $\nu \approx 1.2$.

```
// henoncorrelation.cpp
```

```
#include <iostream>
```

```
#include <cmath> // for sqrt
```

```
using namespace std;
```

```
unsigned long H(double* x,double* y,double r,unsigned long T)
```

```
{
```

```
    double norm;
```

```
    unsigned long sum = 0;
```

```
    for(unsigned long i=0;i<T;i++)
```

```

for(unsigned long j=0;j<T;j++)
{
if(i != j)
{
norm = sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j]));
if(r >= norm) sum++;
}
}
return sum;
}

int main(void)
{
unsigned long T = 20000;           // number of iterations
double* x = new double[T];  double* y = new double[T];
x[0] = 1.161094; y[0] = -0.09541356; // initial values
for(unsigned long j=0;j<T-1;j++)
{
x[j+1] = 1.0+y[j]-1.4*x[j]*x[j]; y[j+1] = 0.3*x[j];
}
double r = 0.001;
while(r <= 0.008)
{
double Cr = H(x,y,r,T);
cout << "r= " << r << " " << "Cr= " << Cr/((double)(T*T)) << endl;
r += 0.001;
}
delete[] x; delete[] y;
return 0;
}

```

1.2.6 Capacity

Let M be a subset of \mathbf{R}^n . We assume that the set M is contained in an invariant manifold of some dynamical system (in our case the strange attractor of the Hénon model). If N_ϵ is the minimum number of boxes of side ϵ in \mathbf{R}^n needed to cover the set M , the *capacity* (also called *box-counting dimension*) is defined as

$$C := \lim_{\epsilon \rightarrow 0} \frac{\ln N_\epsilon}{\ln(1/\epsilon)}.$$

If a set has volume V , the number of boxes of side ϵ needed to cover the set is roughly

$$N_\epsilon \approx V\epsilon^{-C}.$$

This equation may be rewritten as

$$\ln N_\epsilon \approx C \ln(1/\epsilon) + \ln V$$

which provides a more practical method of computing the capacity C than the definition, since the latter has a slowly vanishing correction to the capacity $V/\ln(1/\epsilon)$. By plotting

$$\log N_\epsilon \quad \text{versus} \quad \log(1/\epsilon)$$

for decreasing values of ϵ , the formula for C gives the capacity as the asymptotic slope. Let D_H be the Hausdorff dimension. Then we have $D_H \leq C$.

In the C++ program `capacity.cpp` we find the capacity for the Hénon map, where $a = 1.4$ and $b = 0.3$. The numerical simulation yields the value $C \approx 1.26$.

```
// capacity.cpp

#include <iostream>
#include <cmath> // for floor, log
using namespace std;

unsigned long Neps_func(double* x,double* y,unsigned T,double eps)
{
    unsigned i, j, k, Nx, Ny, Neps = 0;
    double mx, my, xmin, ymin, xmax, ymax;
    xmin = x[0]; xmax = x[0]; ymin = y[0]; ymax = y[0];
    for(i=1;i<T;i++)
    {
        if(x[i] < xmin) xmin = x[i]; if(x[i] > xmax) xmax = x[i];
        if(y[i] < ymin) ymin = y[i]; if(y[i] > ymax) ymax = y[i];
    }
    Nx = (unsigned)((xmax-xmin)/eps+1.0);
    Ny = (unsigned)((ymax-ymin)/eps+1.0);
    mx = ((double)Nx-1.0)/(xmax-xmin);
    my = ((double)Ny-1.0)/(ymax-ymin);
    unsigned long** box = NULL; box = new unsigned long*[Ny];
    for(j=0;j<Ny;j++) box[j] = new unsigned long[Nx];

    for(i=0;i<Ny;i++)
        for(j=0;j<Nx;j++) box[i][j] = 0;

    for(i=0;i<T;i++)
    {
        k = (unsigned long) floor(mx*(x[i]-xmin)+0.5);
        j = (unsigned long) floor(my*(y[i]-ymin)+0.5);
        box[j][k] = 1;
    }
    for(i=0;i<Ny;i++)
        for(j=0;j<Nx;j++) Neps += box[i][j];

    for(i=0;i<Ny;i++) delete[] box[i];
    delete[] box;
}
```

```

    return Neps;
}

int main(void)
{
    unsigned long T = 200000;
    double* x = new double[T]; double* y = new double[T];
    x[0] = 1.161094; y[0] = -0.09541356;
    for(int j=0;j<T-1;j++)
    { x[j+1] = 1.0+y[j]-1.4*x[j]*x[j]; y[j+1] = 0.3*x[j]; }

    double eps = 0.01;
    unsigned Neps = Neps_func(x,y,T,eps);
    cout << "Neps = " << Neps << " " << "eps = " << eps << endl;
    cout << "log(Neps) = " << log((double) Neps) << " "
        << "log(1.0/eps) = " << log(1.0/eps) << endl;
    eps = 0.005;
    Neps = Neps_func(x,y,T,eps);
    cout << "Neps = " << Neps << " " << "eps = " << eps << endl;
    cout << "log(Neps) = " << log((double) Neps) << " "
        << "log(1.0/eps) = " << log(1.0/eps) << endl;
    eps = 0.002;
    Neps = Neps_func(x,y,T,eps);
    cout << "Neps = " << Neps << " " << "eps = " << eps << endl;
    cout << "log(Neps) = " << log((double) Neps) << " "
        << "log(1.0/eps) = " << log(1.0/eps) << endl;
    eps = 0.001;
    Neps = Neps_func(x,y,T,eps);
    cout << "Neps = " << Neps << " " << "eps = " << eps << endl;
    cout << "log(Neps) = " << log((double) Neps) << " "
        << "log(1.0/eps) = " << log(1.0/eps) << endl;
    delete[] x; delete[] y;
    return 0;
}

```

1.2.7 Hyperchaos

We consider a system of first order autonomous ordinary difference equations

$$x_{1t+1} = f_1(x_{1t}, x_{2t}), \quad x_{2t+1} = f_2(x_{1t}, x_{2t}).$$

We assume that f_1 and f_2 are smooth functions. We also assume that the solution (x_{1t}, x_{2t}) is bounded. For certain systems we can find so-called *hyperchaos*. This means the system admits two one-dimensional Liapunov exponents $(\lambda_1^I, \lambda_2^I)$ and one two-dimensional Liapunov exponent. Next we derive an equation for the two-dimensional Liapunov exponent λ^{II} .

The variational equation is given by

$$\begin{aligned} y_{1t+1} &= \frac{\partial f_1}{\partial x_1}(\mathbf{x} = \mathbf{x}_t)y_{1t} + \frac{\partial f_1}{\partial x_2}(\mathbf{x} = \mathbf{x}_t)y_{2t} \\ y_{2t+1} &= \frac{\partial f_2}{\partial x_1}(\mathbf{x} = \mathbf{x}_t)y_{1t} + \frac{\partial f_2}{\partial x_2}(\mathbf{x} = \mathbf{x}_t)y_{2t}. \end{aligned}$$

Let (v_{1t}, v_{2t}) satisfy the variational equation, i.e.

$$\begin{aligned} v_{1t+1} &= \frac{\partial f_1}{\partial x_1}(\mathbf{x} = \mathbf{x}_t)v_{1t} + \frac{\partial f_1}{\partial x_2}(\mathbf{x} = \mathbf{x}_t)v_{2t} \\ v_{2t+1} &= \frac{\partial f_2}{\partial x_1}(\mathbf{x} = \mathbf{x}_t)v_{1t} + \frac{\partial f_2}{\partial x_2}(\mathbf{x} = \mathbf{x}_t)v_{2t}. \end{aligned}$$

Let $\{\mathbf{e}_1, \mathbf{e}_2\}$ be the standard basis in \mathbf{R}^2 , i.e.,

$$\left\{ \mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

We can write

$$\mathbf{y}_t = y_{1t}\mathbf{e}_1 + y_{2t}\mathbf{e}_2, \quad \mathbf{v}_t = v_{1t}\mathbf{e}_1 + v_{2t}\mathbf{e}_2.$$

Next we calculate $\mathbf{y}_t \wedge \mathbf{v}_t$, where \wedge denotes the *Graßmann product* (also called *exterior product* or *wedge product*). We find

$$\begin{aligned} \mathbf{y}_t \wedge \mathbf{v}_t &= (y_{1t}\mathbf{e}_1 + y_{2t}\mathbf{e}_2) \wedge (v_{1t}\mathbf{e}_1 + v_{2t}\mathbf{e}_2) \\ &= y_{1t}v_{2t}\mathbf{e}_1 \wedge \mathbf{e}_2 + y_{2t}v_{1t}\mathbf{e}_2 \wedge \mathbf{e}_1 \\ &= (y_{1t}v_{2t} - y_{2t}v_{1t})\mathbf{e}_1 \wedge \mathbf{e}_2 \end{aligned}$$

where we have used the *distributive law*

$$(a\mathbf{e}_i + b\mathbf{e}_j) \wedge (c\mathbf{e}_k + d\mathbf{e}_l) = (ac)\mathbf{e}_i \wedge \mathbf{e}_k + (ad)\mathbf{e}_i \wedge \mathbf{e}_l + (bc)\mathbf{e}_j \wedge \mathbf{e}_k + (bd)\mathbf{e}_j \wedge \mathbf{e}_l$$

and that $\mathbf{e}_i \wedge \mathbf{e}_j = -\mathbf{e}_j \wedge \mathbf{e}_i$. It follows that $\mathbf{e}_j \wedge \mathbf{e}_j = 0$. The Graßmann product is also associative. We define

$$w_t := y_{1t}v_{2t} - y_{2t}v_{1t}$$

and evaluate the time evolution of w_t . Since

$$w_{t+1} = y_{1t+1}v_{2t+1} - y_{2t+1}v_{1t+1}$$

we find

$$\begin{aligned} w_{t+1} &= \left(\frac{\partial f_1}{\partial x_1}(\mathbf{x}_t)y_{1t} + \frac{\partial f_1}{\partial x_2}(\mathbf{x}_t)y_{2t} \right) \left(\frac{\partial f_2}{\partial x_1}(\mathbf{x}_t)v_{1t} + \frac{\partial f_2}{\partial x_2}(\mathbf{x}_t)v_{2t} \right) \\ &\quad - \left(\frac{\partial f_2}{\partial x_1}(\mathbf{x}_t)y_{1t} + \frac{\partial f_2}{\partial x_2}(\mathbf{x}_t)y_{2t} \right) \left(\frac{\partial f_1}{\partial x_1}(\mathbf{x}_t)v_{1t} + \frac{\partial f_1}{\partial x_2}(\mathbf{x}_t)v_{2t} \right). \end{aligned}$$

Consequently

$$w_{t+1} = \left(\frac{\partial f_1}{\partial x_1}(\mathbf{x}_t) \frac{\partial f_2}{\partial x_2}(\mathbf{x}_t) - \frac{\partial f_1}{\partial x_2}(\mathbf{x}_t) \frac{\partial f_2}{\partial x_1}(\mathbf{x}_t) \right) w_t$$

where the initial value w_0 is given by $w_0 := y_{10}v_{20} - y_{20}v_{10}$. The *two-dimensional Liapunov exponent* is given by

$$\lambda^{II} := \lim_{T \rightarrow \infty} \frac{1}{T} \ln |w_T|$$

where λ^{II} depends on $x_{10}, x_{20}, y_{10}, y_{20}, v_{10}, v_{20}$. Let λ^I be the maximal one-dimensional Liapunov exponent and let λ^{II} be the maximal two-dimensional Liapunov exponent. If $\lambda^I > 0$ and $\lambda^{II} > 0$ we say that the system shows *hyperchaotic behaviour*. We have

$$\lambda^{II} = \lambda_1^I + \lambda_2^I$$

where λ_1^I and λ_2^I are the two one-dimensional Liapunov exponents.

As an example consider the coupled logistic equation. It is given by

$$x_{1t+1} = rx_{1t}(1 - x_{1t}) + e(x_{2t} - x_{1t}), \quad x_{2t+1} = rx_{2t}(1 - x_{2t}) + e(x_{1t} - x_{2t})$$

where $t = 0, 1, 2, \dots$ and r and e are bifurcation parameters with $1 \leq r \leq 4$. For $e = 0$ we have two uncoupled logistic equations. Depending on the initial conditions and the parameter values one can find the following behaviour: (i) orbits tend to a fixed point, (ii) periodic behaviour, (iii) quasiperiodic behaviour, (iv) chaotic behaviour, (v) hyperchaotic behaviour and (vi) x_{1t} and x_{2t} explode, i.e. for a finite time the state variables x_{1t} and (or) x_{2t} tend to infinity. Let

$$f_1(x_1, x_2) = rx_1(1 - x_1) + e(x_2 - x_1), \quad f_2(x_1, x_2) = rx_2(1 - x_2) + e(x_1 - x_2).$$

Thus we find for the variational equation

$$y_{1t+1} = (r(1 - 2x_{1t}) - e)y_{1t} + ey_{2t}, \quad y_{2t+1} = ey_{1t} + (r(1 - 2x_{2t}) - e)y_{2t}.$$

Furthermore we find

$$w_{t+1} = (r^2(1 - 2x_{1t})(1 - 2x_{2t}) - 2re(1 - x_{1t} - x_{2t}))w_t.$$

In the C++ program `hyperchaos.cpp` we evaluate the one-dimensional and two-dimensional Liapunov exponent for $e = 0.06$ and r in the range $r \in [3.2...3.7]$. For example, for $e = 0.06$ and $r = 3.7$ there is numerical evidence that the system shows hyperchaotic behaviour.

```
// hyperchaos.cpp

#include <fstream>    // for ofstream, close
#include <cmath>      // for fabs, log
using namespace std;

double f1(double x1, double x2, double r, double e)
{ return r*x1*(1.0-x1)+e*(x2-x1); }
```

```

double f2(double x1,double x2,double r,double e)
{ return r*x2*(1.0-x2)+e*(x1-x2); }

double v1(double x1,double x2,double y1,double y2,double r,double e)
{ return (r-2.0*r*x1-e)*y1+e*y2; }

double v2(double x1,double x2,double y1,double y2,double r,double e)
{ return e*y1+(r-2.0*r*x2-e)*y2; }

double varext(double x1,double x2,double w,double r,double e)
{ return (r*r*(1.0-2.0*x1)*(1.0-2.0*x2)-2.0*e*r*(1.0-x1-x2))*w; }

int main(void)
{
    int T = 700; // number of iterations
    double r, e; // bifurcation parameters
    double rmin = 3.2, rmax = 3.7;
    r = rmin; e = 0.06;
    ofstream data("lambda.dat");
    while(r <= rmax)
    {
        double x11 = 0.7, x22 = 0.3; // initial values
        double x1, x2;
        // remove the transients
        for(int t=0;t<10;t++)
        { x1 = x11; x2 = x22; x11 = f1(x1,x2,r,e); x22 = f2(x1,x2,r,e); }

        double y11 = 0.5, y22 = 0.5;
        double w;
        double w1 = 0.5;
        double y1, y2;

        for(int t=0;t<T;t++)
        {
            x1 = x11; x2 = x22; y1 = y11; y2 = y22; w = w1;
            x11 = f1(x1,x2,r,e); x22 = f2(x1,x2,r,e);
            y11 = v1(x1,x2,y1,y2,r,e); y22 = v2(x1,x2,y1,y2,r,e);
            w1 = varext(x1,x2,w,r,e);
        }
        double lambdaI = log(fabs(y11) + fabs(y22))/((double) T);
        double lambdaII = log(fabs(w1))/((double) T);
        data << r << " " << lambdaI << " " << lambdaII << "\n";
        r += 0.005;
    }
    data.close();
    return 0;
}

```

1.2.8 Domain of Attraction

Consider a system of nonlinear first order autonomous difference equations

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{r}), \quad t = 0, 1, 2, \dots$$

where $\mathbf{r} = (r_1, r_2, \dots, r_p)$ are bifurcation parameters. Let us assume that the initial values \mathbf{x}_0 are given. The behaviour for $t \rightarrow \infty$ depends on the bifurcation parameters. In particular one is interested in finding the domain for the bifurcation parameter values \mathbf{r} where the solution escapes to infinity, i.e.,

$$\|\mathbf{x}_t\| \rightarrow \infty \quad \text{for } t \rightarrow \infty.$$

The domain of attraction can have a fractal structure. Obviously, the domain also depends on the initial values. Thus one keeps the initial values fixed.

As an example consider the coupled logistic equation

$$x_{1t+1} = rx_{1t}(1 - x_{1t}) + e(x_{2t} - x_{1t}), \quad x_{2t+1} = rx_{2t}(1 - x_{2t}) + e(x_{1t} - x_{2t})$$

where $t = 0, 1, 2, \dots$ and r and e are bifurcation parameters with $1 \leq r \leq 4$. For $e = 0$ we have two uncoupled logistic equations. Depending on the initial conditions and the parameter values one can find the following behaviour for $t \rightarrow \infty$: (i) orbits tend to a fixed point, (ii) periodic behaviour, (iii) quasiperiodic behaviour, (iv) chaotic behaviour, (v) hyperchaotic behaviour and (vi) x_{1t} and (or) x_{2t} tend to infinity.

In the C++ program `domain.cpp` we find the escape domain, for the parameter regions

$$e \in [0.04, 0.09] \quad \text{and} \quad r \in [3.5, 4.0].$$

The initial values are fixed to $x_{10} = 0.7$ and $x_{20} = 0.3$. Obviously, the initial values must be chosen so that $x_{10} \neq x_{20}$ otherwise the coupling term will cancel out and we have two uncoupled logistic maps.

```
// domain.cpp

#include <fstream> // for ofstream, close
#include <cmath> // for fabs, log
using namespace std;

double f1(double x1, double x2, double r, double e)
{ return r*x1*(1.0-x1)+e*(x2-x1); }

double f2(double x1, double x2, double r, double e)
{ return r*x2*(1.0-x2)+e*(x1-x2); }

int main(void)
{
```

```

int T = 800; // number of iterations
double r, e; // bifurcation parameters
double rmin = 3.6, rmax = 4.0;
double emin = 0.04, emax = 0.08;
r = rmin; e = emin;
ofstream data("domain.dat");
while(e <= emax)
{
while(r <= rmax)
{
double x11 = 0.7, x22 = 0.3;
double x1, x2;
for(int t=0;t<=T;t++)
{
x1 = x11; x2 = x22; x11 = f1(x1,x2,r,e); x22 = f2(x1,x2,r,e);
}
if((fabs(x11) > 20) || (fabs(x22) > 20))
{
data << r << " " << e << "\n";
}
r += 0.0005;
}
e += 0.00005;
r = rmin;
}
data.close();
return 0;
}

```

1.2.9 Newton Method in the Complex Domain

In connection with chaotic behaviour and fractals the *Newton method* is considered in the complex domain \mathbf{C} . Let f be a differentiable complex valued function. Given an approximate value of z_0 to the solution of $f(z) = 0$ the Newton method finds the next approximation by calculating

$$z_{t+1} = z_t - \frac{f(z_t)}{f'(z_t)}$$

where $t = 0, 1, 2, \dots$ and $f'(z_t) \neq 0$ is the derivative of f at $z = z_t$. One calls

$$\left\{ \hat{\mathbf{C}} : g(z) = z - \frac{f(z)}{f'(z)} \right\}$$

the Newton transformation associated with the function f . The general expectation is that a typical orbit $\{f^{(t)}(z_0)\}$ which starts from an initial guess $z_0 \in \mathbf{C}$, will converge to one of the roots.

Example. Consider the polynomial

$$p(z) = z^4 - 1$$

for $z \in \mathbf{C}$. There are four distinct complex numbers, a_j ($j = 1, 2, 3, 4$) such that

$$p(a_j) = 0.$$

These are called the *roots*, or the zeros, the polynomial $p(z)$. The roots are given by

$$a_1 = 1, \quad a_2 = -1, \quad a_3 = i, \quad a_4 = -i.$$

The Newton method tells us to consider the dynamical system

$$\left\{ \widehat{\mathbf{C}} : f(z) := z - \frac{p(z)}{p'(z)} \right\}$$

with $p' \equiv dp/dz = 4z^3$. We call f the Newton transformation associated with the function p . The general expectation is that a typical orbit


$$\{ f^{(t)}(z_0) \} = \{ z_0, f(z_0), f(f(z_0)), \dots \}$$

which starts from an initial "guess" $z_0 \in \mathbf{C}$, will converge to one of the roots of p . For the present case we find that the Newton transformation is given by

$$f(z) = \frac{3z^4 + 1}{4z^3}.$$

We expect the orbit of z_0 to converge to one of the numbers a_1, a_2, a_3 or a_4 . If we choose z_0 close enough to a_j then it is readily proved that

$$\lim_{n \rightarrow \infty} f^{(n)}(z_0) = a_j, \quad \text{for } j = 1, 2, 3, 4.$$

If, on the other hand, z_0 is far away from all of the a_j 's, then what happens? Perhaps the orbit of z_0 converges to the root of $p(z)$ closest to z_0 ? Or perhaps the orbit does not settle down, but wanders, hopelessly, forever? 

In the C++ program `complexnewton.cpp` we use the `complex` class of C++.

```
// complexnewton.cpp

#include <iostream>
#include <complex>
using namespace std;

int main(void)
{
    int T = 1000; // number of iterations
    complex<double> z0(0.4,0.2);
    complex<double> z1 = (3.0*z0*z0*z0*z0+1.0)/(4.0*z0*z0*z0);
```

```

double eps = 0.0001;
while(abs(z1-z0) > eps)
{
z0 = z1; z1 = (3.0*z0*z0*z0*z0+1.0)/(4.0*z0*z0*z0);
}
cout << "root = " << z1 << endl;
return 0;
}

```

The iteration provides the root $\text{complex}(0, -1)$, i.e. $-i$.

1.2.10 Newton Method in Higher Dimensions

We consider the system

$$f_1(x_1, x_2, \dots, x_n) = 0, \dots, f_n(x_1, x_2, \dots, x_n) = 0.$$

We denote by \mathbf{x} the vector with components (x_1, x_2, \dots, x_n) and by \mathbf{f} the vector with components (f_1, f_2, \dots, f_n) . Then the system can be written as one vector equation, $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. If we take the gradients of the components, we obtain a function matrix called the *Jacobian matrix* J . It is defined as follows

$$J(\mathbf{f}(\mathbf{x})) := \left(\frac{\partial f_i}{\partial x_k} \right) \equiv \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}.$$

The problem is now to solve the equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. We suppose that the vector \mathbf{y} is the exact solution and that our present approximation \mathbf{x} can be written as $\mathbf{x} = \mathbf{y} + \mathbf{h}$. We compute $f_i(x_1, \dots, x_n)$ and call these known values g_i . Hence

$$f_i(y_1 + h_1, y_2 + h_2, \dots, y_n + h_n) = g_i.$$

The first term is zero by definition, and neglecting higher order terms we obtain with $J_{ik} = (\partial f_i / \partial x_k)_{\mathbf{x}=\mathbf{y}}$:

$$J\mathbf{h} = \mathbf{g} \quad \text{and} \quad \mathbf{h} = J^{-1}\mathbf{g}.$$

We suppose that the matrix J is nonsingular. The vector \mathbf{h} found in this way in general does not give us an exact solution. We arrive at the iteration formula

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - J^{-1}\mathbf{f}(\mathbf{x}^{(t)}).$$

In one dimension this formula becomes the usual Newton formula. The matrix J must be nonsingular. The matrix J changes from step to step. This suggests a

simplification to reduce computational efforts: replace $J(\mathbf{x}^{(t)})$ by $J(\mathbf{x}^{(0)})$. In the C++ program we consider the case $n = 2$ with

$$f_1(x_1, x_2) = x_1^2 + x_2^2 - 1, \quad f_2(x_1, x_2) = x_1 - x_2.$$

This system admits two solutions, namely $(x_1, x_2) = (1/\sqrt{2}, 1/\sqrt{2})$ and $(x_1, x_2) = (-1/\sqrt{2}, -1/\sqrt{2})$.

```
// twonewton.cpp

#include <iostream>
#include <cmath>      // for fabs
using namespace std;

double g1(double x, double y)
{ return (x-(x*x-y*y+2.0*x*y-1.0)/(2.0*(x+y))); }

double g2(double x, double y)
{ return (y-(-x*x+y*y+2.0*x*y-1.0)/(2.0*(x+y))); }

int main(void)
{
    double x0, y0, x1, y1, eps;
    int t = 0;
    x1 = 3.5, y1 = 20.3; // initial values
    eps = 0.0005;
    do
    {
        x0 = x1; y0 = y1;
        x1 = g1(x0, y0); y1 = g2(x0, y0);
        t++;
    } while((fabs(x0-x1) > eps) && (fabs(y0-y1) > eps));

    cout << "t = " << t << endl;
    cout << "x1 = " << x1 << endl;
    cout << "y1 = " << y1 << endl;
    return 0;
}
```

1.2.11 Ruelle-Takens-Newhouse Scenario

In this route to chaos we have the following sequence when the bifurcation parameter r is changed. A stationary point (fixed point) bifurcates to a periodic orbit, which then bifurcates to a doubly periodic orbit formed by the surface of a torus, which then bifurcates to a system with chaotic behaviour. Newhouse, Ruelle and Takens [81] conjectured that small nonlinearities would destroy triply periodic motion. They proved the following.

Theorem. Let V be a constant vector field on the torus $T^n = \mathbf{R}^n/\mathbf{Z}^n$. If $n \geq 3$ every C^2 neighbourhood of V contains a vector field V^T with a strange Axiom A attractor. If $n \geq 4$, we may take C^∞ instead of C^2 .

A dynamical system is (or satisfies) Axiom A if its nonwandering set

- (i) has a hyperbolic structure, and
- (ii) is the closure of the set of closed orbits of the system.

We define a point p to be non-wandering if, for all neighbourhoods U of p , $f(U) \setminus U$ is nonempty for arbitrary large $t \in \mathbf{R}$ or $t \in \mathbf{N}$. The term *non-wandering point* is an unhappy one, since not only may the point wander away from its original position but it may never come back again. The set of all non-wandering points of the map f are called non-wandering set of f .

We consider the map (Lopez-Ruiz and Perez-Garica [69], [70])

$$x_{t+1} = r(3y_t + 1)x_t(1 - x_t), \quad y_{t+1} = r(3x_t + 1)y_t(1 - y_t)$$

which shows the Ruelle-Takens-Newhouse transition to chaos. The control parameter is r . We calculate the variational equation symbolically

$$\begin{aligned} u_{t+1} &= r(3y_t + 1)(1 - 2x_t)u_t + 3rx_t(1 - x_t)v_t \\ v_{t+1} &= 3ry_t(1 - y_t)u_t + r(3x_t + 1)(1 - 2y_t)v_t \end{aligned}$$

and then iterate these four equations using the data type `double`. The largest one-dimensional Liapunov exponent is calculated approximately

$$\lambda \approx \frac{1}{T} \ln(|u_T| + |v_T|)$$

where $r = 1.0834$ and T is large. The fixed points of the map are given by the solution of the system

$$r(3y^* + 1)x^*(1 - x^*) = x^*, \quad r(3x^* + 1)y^*(1 - y^*) = y^*.$$

We find five fixed points

$$\begin{aligned} x_1^* &= \frac{1}{3r} \left(-\sqrt{4r^2 - 3r} + r \right), & y_1^* &= \frac{1}{3r} \left(-\sqrt{4r^2 - 3r} + r \right) \\ x_2^* &= \frac{1}{3r} \left(\sqrt{4r^2 - 3r} + r \right), & y_2^* &= \frac{1}{3r} \left(\sqrt{4r^2 - 3r} + r \right) \\ x_3^* &= (r - 1)/r, & y_3^* &= 0 \\ x_4^* &= 0, & y_4^* &= 0 \\ x_5^* &= 0, & y_5^* &= (r - 1)/r. \end{aligned}$$

The fixed points (x_1^*, y_1^*) and (x_2^*, y_2^*) exist only for $r \geq 3/4$.

```

// ruelle.cpp

#include <iostream>
#include <cmath>
#include "symbolicc++.h"
using namespace std;

template <class T> T f(T x,T y,T r)
{ return r*(T(3)*y+T(1))*x*(T(1)-x); }

template <class T> T g(T x,T y,T r)
{ return r*(T(3)*x+T(1))*y*(T(1)-y); }

int main(void)
{
    int T = 500;    // number of iterations
    double x2, y2, u2, v2;
    Symbolic x("x"), x1("x1"), y("y"), y1("y1"), r("r"),
        u("u"), u1("u1"), v("v"), v1("v1");
    x1 = f(x,y,r); y1 = g(x,y,r);
    cout << "x1 = " << x1 << endl;
    cout << "y1 = " << y1 << endl;
    u1 = df(x1,x)*u+df(x1,y)*v; // variational equation
    v1 = df(y1,y)*v+df(y1,x)*u; // variational equation
    cout << "u1 = " << u1 << endl;
    cout << "v1 = " << v1 << endl; cout << endl;
    // initial values
    Equations values = (x==0.3,y==0.4,r==1.0834,u==0.5,v==0.6);
    for(int t=1;t<T;t++)
    {
        x2 = x1[values]; y2 = y1[values];
        u2 = u1[values]; v2 = v1[values];
        values = (r==1.0834,x==x2,y==y2,u==u2,v==v2);
        cout << "The Liapunov exponent for t = " << t << " is "
            << log(fabs(double(rhs(values,u)))+fabs(double(rhs(values,v)))))/t
            << endl;
    }
    return 0;
}

```

1.2.12 Periodic Orbits and Topological Degree

We consider the problem of finding the solutions of a system of nonlinear equations of the form

$$\mathbf{f}_n = \mathbf{0}_n$$

where $\mathbf{f}_n = (f_1, f_2, \dots, f_n) : D_n \subset \mathbf{R}^n \rightarrow \mathbf{R}^n$ is a function from a domain D_n into \mathbf{R}^n , $\mathbf{0}_n = (0, 0, \dots, 0)$ and $\mathbf{x} = (x_1, x_2, \dots, x_n)$. The above system can be written as

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0. \end{aligned}$$

Topological degree theory gives us information on the existence of solutions of the above system, their number and their nature. Kronecker introduced the concept of topological degree in 1869, while Picard in 1892 provided a theorem for computing the exact number of solutions of the system. Numerical methods based on topological degree theory have been applied to numerous dynamical systems.

In order to define the concept of the topological degree we consider the function \mathbf{f}_n to be continuous on the closure $\overline{D_n}$ of D_n , satisfying also $\mathbf{f}_n(\mathbf{x}) \neq \mathbf{0}_n$ for \mathbf{x} on the boundary $b(D_n)$ of D_n . We also consider the solutions to be simple, i.e. the determinant of the corresponding Jacobian matrix ($J_{\mathbf{f}_n}$) at the solution, to be different from zero. Then the *topological degree* of \mathbf{f}_n at $\mathbf{0}_n$ relative to D_n is defined as

$$\text{deg}[\mathbf{f}_n, D_n, \mathbf{0}_n] := \sum_{\mathbf{x} \in \mathbf{f}_n^{-1}(\mathbf{0}_n)} \text{sgn}(\det J_{\mathbf{f}_n}(\mathbf{x})) = N_+ - N_-$$

where $\det J_{\mathbf{f}_n}$ is the determinant of the Jacobian matrix of \mathbf{f}_n , sgn is the sign function, N_+ the number of roots with $\det J_{\mathbf{f}_n} > 0$ and N_- the number of roots with $\det J_{\mathbf{f}_n} < 0$. It is evident that if a nonzero value of $\text{deg}[\mathbf{f}_n, D_n, \mathbf{0}_n]$ is obtained then there exists at least one solution of system $\mathbf{f}_n(\mathbf{x}) = \mathbf{0}_n$ within D_n .

A practical way to find the topological degree is the computation of the Kronecker integral. In particular, under the assumption of the definition given above of the topological degree the $\text{deg}[\mathbf{f}_n, D_n, \mathbf{0}_n]$ can be computed as

$$\text{deg}[\mathbf{f}_n, D_n, \mathbf{0}_n] = \frac{\Gamma(n/2)}{2\pi^{n/2}} \int_{b(D_n)} \int \dots \int \frac{\sum_{j=1}^n A_j dx_j \dots dx_{j-1} dx_{j+1} \dots dx_n}{(f_1^2 + f_2^2 + \dots + f_n^2)^{n/2}}$$

where

$$A_j := (-1)^{n(j-1)} \det \begin{pmatrix} f_1 & \partial f_1 / \partial x_1 & \dots & \partial f_1 / \partial x_{j-1} & \partial f_1 / \partial x_{j+1} & \dots & \partial f_1 / \partial x_n \\ f_2 & \partial f_2 / \partial x_1 & \dots & \partial f_2 / \partial x_{j-1} & \partial f_2 / \partial x_{j+1} & \dots & \partial f_2 / \partial x_n \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ f_n & \partial f_n / \partial x_1 & \dots & \partial f_n / \partial x_{j-1} & \partial f_n / \partial x_{j+1} & \dots & \partial f_n / \partial x_n \end{pmatrix}$$

and $\Gamma(x)$ is the gamma function. In order to find the number N of solutions of $\mathbf{f}_n = \mathbf{0}_n$ we consider the function

$$\mathbf{f}_{n+1} = (f_1, f_2, \dots, f_n, f_{n+1})^T : D_{n+1} \subset \mathbf{R}^{n+1} \rightarrow \mathbf{R}^{n+1}$$

where

$$f_{n+1} := y \det J_{\mathbf{f}_n}$$

$\mathbf{R}^{n+1} : x_1, x_2, \dots, x_n, y$ and D_{n+1} is the product of D_n with a real interval on the y -axis containing $y = 0$. Then the exact number N of the solutions of the equation $\mathbf{f}_n(\mathbf{x}) = \mathbf{0}_n$ is given by

$$N = \deg[\mathbf{f}_{n+1}, D_{n+1}, \mathbf{0}_{n+1}].$$

Example. Consider the case of a set of two equations

$$f_1(x_1, x_2) = 0, \quad f_2(x_1, x_2) = 0.$$

We find that the number N of roots in the domain $D_2 = [a, b] \times [c, d]$ is given by

$$N = \frac{1}{2\pi} \int_{b(D_2)} (P_1(x_1, x_2)dx_1 + P_2(x_1, x_2)dx_2) + \epsilon \int \int_{D_2} \frac{Q(x_1, x_2)dx_1dx_2}{(f_1^2 + f_2^2 + \epsilon^2 J^2)^{3/2}}$$

where ϵ is an arbitrary positive value,

$$P_j(x_1, x_2) = \frac{\left(f_1 \frac{\partial f_2}{\partial x_j} - f_2 \frac{\partial f_1}{\partial x_j}\right) \epsilon J}{(f_1^2 + f_2^2)(f_1^2 + f_2^2 + \epsilon^2 J^2)^{1/2}}, \quad j = 1, 2$$

and

$$Q(x_1, x_2) = \det \begin{pmatrix} f_1 & \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 \\ f_2 & \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 \\ J & \partial J / \partial x_1 & \partial J / \partial x_2 \end{pmatrix}$$

with J denoting the determinant of the Jacobian matrix of $\mathbf{f}_2 = (f_1, f_2)$. ♣

1.2.13 JPEG file

JPEG, unlike other formats like PPM, PGM, and GIF, is a lossy compression technique; this means visual information is lost permanently. The key to making JPEG work is choosing what data to throw away. JPEG is the image compression standard developed by the Joint Photographic Experts Group. It works best on natural images (scenes). We describe general JPEG compression for greyscale images; however, JPEG compresses color images just as easily. For instance, it compresses the red-green-blue parts of a color image as three separate greyscale images - each compressed to a different extent, if desired. JPEG divides up the image into 8 by 8 pixel blocks, and then calculates the two-dimensional discrete cosine transform (DCT) of each block. The two-dimensional *discrete cosine transform* is given by

$$f(k_1, k_2) = 4 \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) \cos\left(\frac{(2n_1+1)k_1\pi}{2N_1}\right) \cos\left(\frac{(2n_2+1)k_2\pi}{2N_2}\right)$$

where $x(n_1, n_2)$ is an input image with $n_1 = 0, 1, \dots, N_1-1$ and $n_2 = 0, 1, \dots, N_2-1$. The k_1, k_2 are the coordinates in the transform domain, where $k_1 = 0, 1, \dots, N_1$ and $k_2 = 0, 1, \dots, N_2$. A quantizer rounds off the DCT coefficients according to the

quantization matrix. This step produces the "lossy" nature of JPEG, but allows for large compression ratios. JPEG's compression technique uses a variable length code on these coefficients, and then writes the compressed data stream to an output file (*.jpg). For decompression, JPEG recovers the quantized DCT coefficients from the compressed data stream, takes the inverse transforms and displays the image. Instead of the discrete cosine transform in newer versions discrete wavelets are used.

In the Java program `JPEG1.java` we convert the phase-portrait for the Ikeda-Laser map into a JPEG file (`output.jpg`).

```
// JPEG1.java

import com.sun.image.codec.jpeg.*;
import java.awt.*;
import java.awt.geom.Line2D;
import java.awt.image.BufferedImage;
import java.io.FileOutputStream;

public class JPEG1 extends Frame
{
    BufferedImage bi;
    Graphics2D g2;

    public JPEG1()
    {
        bi = new BufferedImage(400,400,BufferedImage.TYPE_INT_RGB);
        g2 = bi.createGraphics();
        double x = 0.5, y = 0.5; // initial values
        double x1, y1;
        double c1 = 0.4, c2 = 0.9, c3 = 9.0, rho = 0.85;
        int T = 20000; // number of iterations
        for(int t=0;t<T;t++)
        {
            x1 = x; y1 = y;
            double taut = c1-c3/(1.0+x1*x1+y1*y1);
            x = rho+c2*x1*Math.cos(taut)-y1*Math.sin(taut);
            y = c2*(x1*Math.sin(taut)+y1*Math.cos(taut));
            double m = 90*x+200; double n = 90*y+200;
            g2.draw(new Line2D.Double(m,n,m,n));
        }
        try {
            FileOutputStream jpegOut = new FileOutputStream("output.jpg");
            JPEGImageEncoder jie = JPEGCodec.createJPEGEncoder(jpegOut);
            jie.encode(bi);
            jpegOut.close();
            System.exit(0);
        }
        catch(Exception e) { }
    }
}
```

```
} // end constructor JPEG1()

public static void main(String args[])
{
    JPEG1 jp = new JPEG1();
}
}
```