1. Programming Method

The four coupled first order ODEs have to be solved simultaneously. We can rewrite the set of equations as a vector equation:

$$\frac{d\overrightarrow{x}}{dt} = f\left(\overrightarrow{x}, t\right)$$

where $\overrightarrow{x} = \left[\ x,\ y,\ p_x,\ p_y\ \right]$. The transformation $f()$ can carried out by an RK4 algorithm capable of handling vectors of any order, as given below:

n dimensional RK4 solver: from util.c

```
/* n dimensional RK4 algorithm for calculating a single step.
 * This implementation is loosely based on the one given in Numerical Recipes.
 * As we cannot multiply arrays with numbers in C (without using a loop), an
 * approach different from the traditional way (k1, k2, ...) has been followed.
 */
void rk4(double x, double y[], int n, deriv_func f, double h, double yout[])
{
        int i;
        double hh, xh, *dy, *dys, *yt;

        yt = vector(n);
        dy = vector(n);         /* derivatives of y1, y2, ... */
        dys = vector(n);        /* sum of derivatives */
        hh = 0.5 * h;
        xh = x + hh;

        f(x, y, dy);                            /* first step */
        for (i = 0; i < n; ++i) {
                yt[i] = y[i] + hh * dy[i];      /* y + k1 / 2 */
                dys[i] = dy[i];
        }
        f(xh, yt, dy);                          /* second step */
        for (i = 0; i < n; ++i) {
                yt[i] = y[i] + hh * dy[i];      /* y + k2 / 2 */
                dys[i] += 2 * dy[i];
        }
        f(xh, yt, dy);                          /* third step */
        for (i = 0; i < n; ++i) {
                yt[i] = y[i] + h * dy[i];       /* y + k3 */
                dys[i] += 2 * dy[i];
        }
        f(x + h, yt, dy);                       /* fourth step */
        for (i = 0; i < n; ++i)
                yout[i] = y[i] + h / 6.0 * (dys[i] + dy[i]);    /* final y */
```

```
        free(yt);
        free(dy);
        free(dys);
}
```

2. Setting up the Hénon–Heiles System

Inspecting the Hénon–Heiles system requires repeatedly calculating various quantities (like p_x, potential, energy, etc.). The routines below were written to undertake these tasks in a modular way. There are also routines for producing equipotential lines, plotting phase spaces, etc.

henon_heiles.c

```
/* Hénon–Heiles related functions */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "util.h"

/* Return the value of the Hénon–Heiles potential at (x, y) */
double hh(double x, double y)
{
        return (x * x + y * y) / 2.0 + x * x * y - pow(y, 3) / 3.0;
}

/* Compute the time derivative of v and store it in vdot */
void hh_evol(double t, double v[4], double vdot[4])
{
        double x, y;
        x = v[0], y = v[1];

        /* Hénon–Heiles differential equations of motion */
        vdot[0] = v[2];
        vdot[1] = v[3];
        vdot[2] = -x - 2 * x * y;
        vdot[3] = -y - x * x + y * y;
}

/* Return x-coordinate of the equipotential line u at y
 * Returns -1 if the point is invalid */
double hh_equi(double y, double u)
{
        double xsqr;
```

```
        xsqr = (u - y * y / 2 + pow(y, 3) / 3) / (y + 0.5);
        if (xsqr > 0.0 || eq(xsqr, 0.0))
                return sqrt(xsqr);
        else
                return -1;
}

/* Return x-momentum for a given energy, (x, y) and py
 * Returns -1 if the point is invalid */
double px(double e, double x, double y, double py)
{
        double pxsqr;

        pxsqr = 2 * (e - hh(x,y)) - py * py;
        if (pxsqr > 0.0 || eq(pxsqr, 0.0))
                return sqrt(pxsqr);
        else
                return -1;
}

/* Compute the energy from the Hamiltonian, given coordinates x */
double en(double *x, int n)
{
        double ke;

        ke = 0.5 * (pow(x[2], 2) + pow(x[3], 2));
        return ke + hh(x[0], x[1]);
}

/* Compute the next iteration of the area preserving map studied by
 * Hénon and Heiles in their paper, for a given 'a' */
void hh_map(double xin[2], double xout[2], double a)
{
        double xt;

        xt = xin[0] + a * (xin[1] - pow(xin[1], 3));
        xout[1] = xin[1] - a * (xt - pow(xt, 3));
        xout[0] = xt;
}

/* Print the coordinates of the equipotential lines for a number of u values
 * Prints only the positive half (x > 0) of the curves; mirror to get full */
void plot_equi()
{
        double x, y, u[] = { 1 / 6.0, 1 / 8.0, 1 / 12.0, 1 / 24.0, 1 / 100.0 };
        double xf, yf;
        int i;
```

```
        for (i = 0; i < LEN(u); ++i) {
                for (y = 1; y >= -0.5; y -= 0.0001) {
                        x = hh_equi(y, u[i]);
                        if (x >= 0) {
                                printf("%lf\t%lf\n", x, y);
                                xf = x;
                                yf = y;
                        }
                }
                /* Print the mirror of the last point, to get continuation while
                 * plotting */
                printf("%lf\t%lf\n", -xf, yf);
                printf("\n\n");
        }
}

/* Iterate hh_map() n times starting from the initial point x, for given 'a' */
void plot_map(double *x, double a, int n)
{
        int i;

        for (i = 0; i < n; ++i) {
                print_vec(x, 2);
                hh_map(x, x, a);
        }
}

/* Print the first n transformations of the initial point x in the x = 0, px > 0
 * section for a given energy */
void phase_space(double *x, double energy, int n)
{
        int pcount;
        long t;
        double xnew[4], h;

        pcount = t = 0;
        h = 0.001;
        while (pcount < n) {
                rk4(h * t++, x, 4, hh_evol, h, xnew);
                /* Check if we crossed the x = 0 plane with positive momentum */
                if (xnew[0] * x[0] < 0 && x[2] > 0.0) {
                        printf("%lf\t%.12lf\t", h * t, en(x, 4));
                        print_vec(x, 4);
                        ++pcount;
                }
                copy_vec(xnew, x, 4);
        }
}
```

3. Equipotential Lines

The equipotential lines of the Hénon–Heiles potential are calculated by the `plot_equi()` function above. The energies used in the original paper are hard-coded in the function. The points printed are plotted using a graphing tool like `gnuplot` :

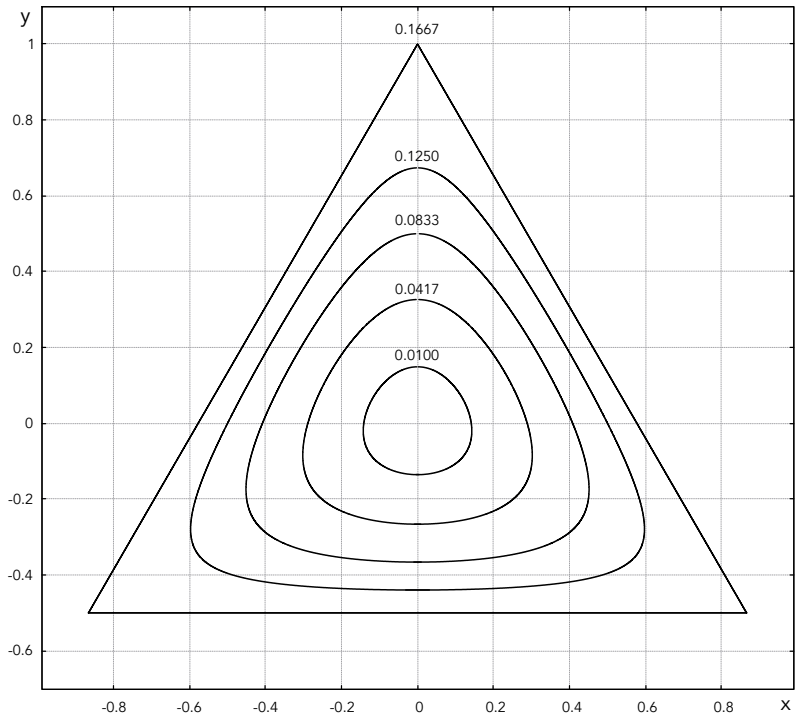

Fig. 1: Equipotential lines of Hénon–Heiles potential

4. Phase Space plots

The first standalone program is to reproduce the three phase space plots in the original paper corresponding to $E = 0.0833, 0.125$ and 0.16667. It uses the functions from `henon_heiles.c` , hence that has been included. The initial points in each case were found by some guesswork and trial and error, to get matching results with the paper:

hh_phase.c

```
#include <stdio.h>
#include "henon_heiles.c"

/* Program to closely reproduce the phase space diagrams obtained by Hénon and
 * Heiles (fig. 4 to 6) for three different energies */
int main()
{
        int i;
        double egy, x[4];
        /* Initial points for the three figures */
        double fig4[][2] = {
```

```
                { -0.365, 0.0 }, { -0.35, 0.0 }, { -0.3614, 0.0 },
                { -0.29, 0.0 }, { -0.125, 0.0 }, { -0.12, 0.0 }, { 0.0, 0.0 },
                { 0.2, 0.0 }, { 0.0, 0.24 }, { 0.0, -0.24 }
        };
        double fig5[][2] = {
                { -0.41, 0.0 }, { -0.3614, 0.0 }, { -0.1, 0.0 }, { 0.0, 0.0 },
                { 0.1, 0.0 }, { 0.2, 0.0 }, { 0.562, 0.0 }, { 0.0, 0.5 - EPS },
                { 0.0, -0.225 }, { 0.0, -0.2 }, { 0.0, 0.2 }, { 0.0, 0.225 }
        };
        double fig6[][2] = {
                { 0.0, 0.0 }, { 0.5, 0.0 }, { 1.0 - EPS, 0.0 }, { 0.3, 0.0 },
                { -0.2, -0.44 }, { -0.2, 0.44 }
        };
        /* Demonstrating for energy = 1 / 12.0 (fig. 4) */
        /* Change values and variables correspondingly for the remaining two. */
        egy = 1 / 12.0;
        for (i = 0; i < LEN(fig4); ++i) {
                x[0] = 0.0;
                x[1] = fig4[i][0];
                x[3] = fig4[i][1];
                x[2] = px(egy, x[0], x[1], x[3]);
                phase_space(x, egy, 1000);
                printf("\n\n");
        }
        return 0;
}
```

The program outputs both E v/s t values as well as 1000 phase space points. Plots:

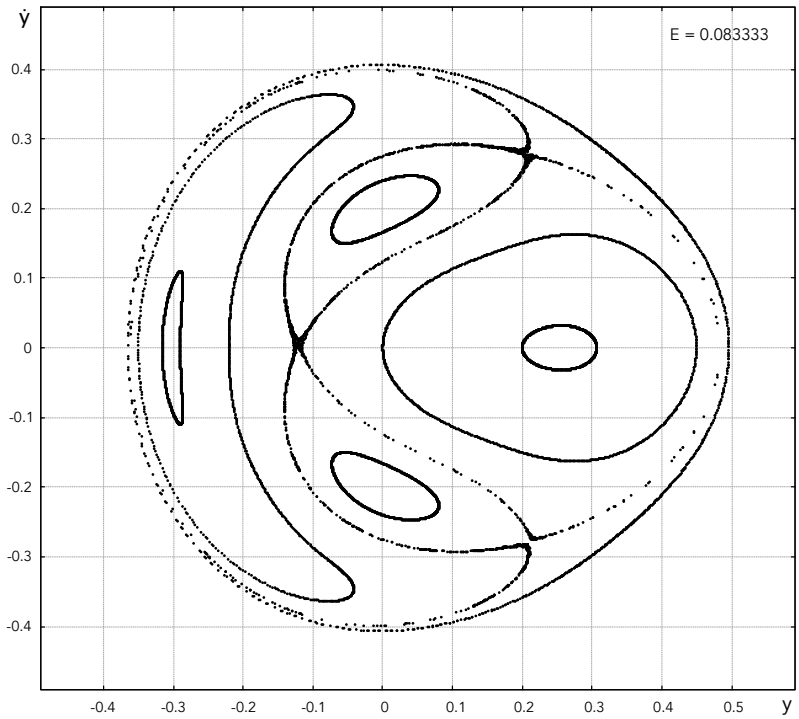

FIG. 2 (a): Phase space for E = 0.0833

Even thought these are invariant curves, we can spot slightly dense island-like phenomena near the points where two trajectories cross each other (these are unstable fixed points).

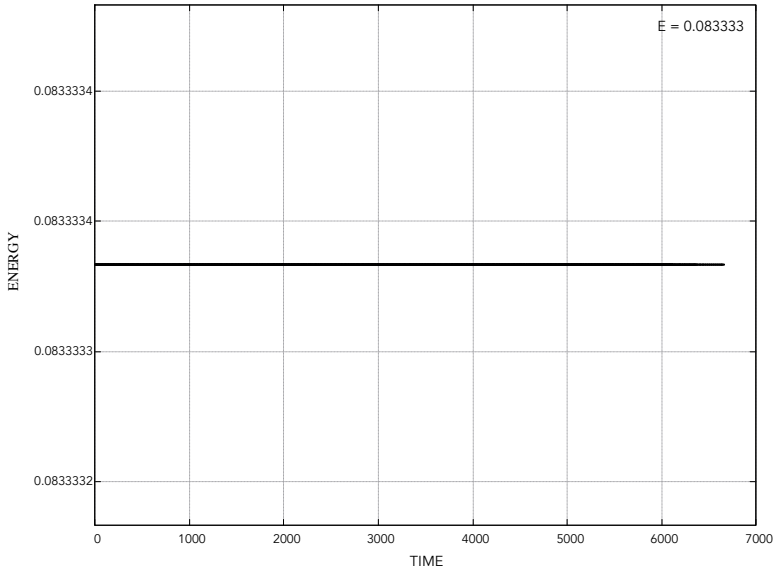

FIG. 2 (b): Evolution of E v/s t for E = 0.0833

We observe that the energy does not drop even slightly over time, unlike stated in the paper. May be the calculations are more robust in modern systems.

Changing the value of the parameters in the above program, we obtain the remaining two phase space diagrams with their energy evolutions:

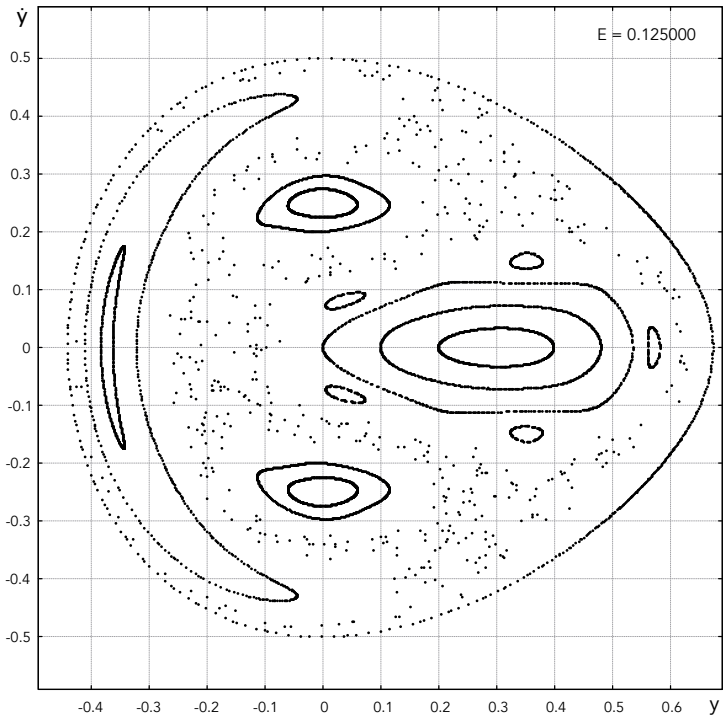

FIG. 3 (a): Phase space for E = 0.125

Here we have more ergodic points than in the paper. The island phenomena is replicated pretty accurately. The boundary curve still looks more an invariant curve than an ergodic trajectory.

Another observation is that the *extent* of a family of invariant curves is somewhat visible as the area left out by the surrounding ergodic points. If we generate quite a large number of ergodic points, we may then be able to find small isolated islands by looking for gaps in the ergodic region.

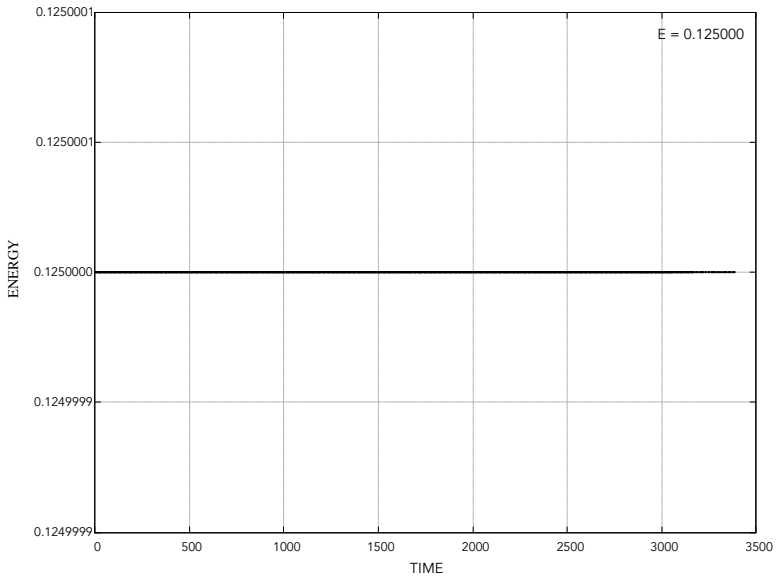

Fig. 3 (b): E v/s t for E = 0.125

Again, the energies are not fluctuating with time. Here is the last and mostly chaotic phase space:

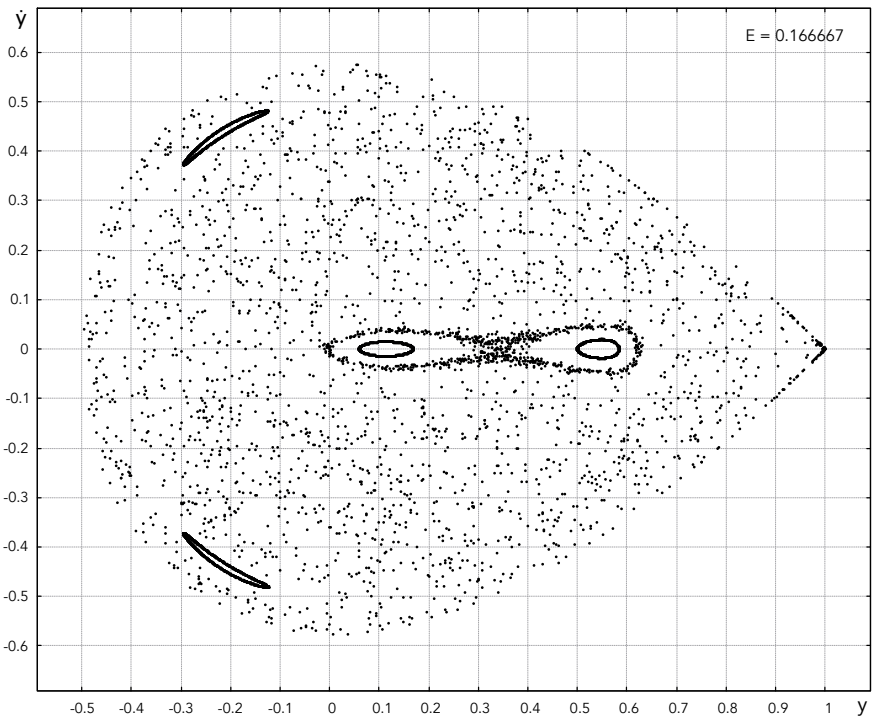

Fig. 4 (a): Phase space for E = 0.16667

Again we have plotted considerably more ergodic points to show the features in higher contrast. The '8'-shaped loop of small islands is quite prominent. The outer boundary no longer looks like an invariant curve; even if it is, the distribution of points on it is somewhat erratic. And we can notice a number of small gaps in the ergodic region, in support of our previous speculation.

The fact that the energy remain stable even for chaotic trajectories is proven by the diagram below:

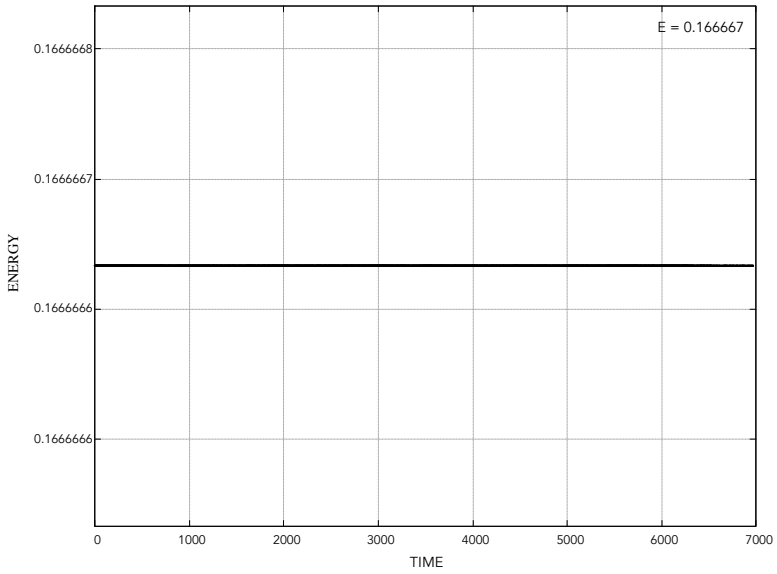

Fig. 4 (b): E v/s t for E = 0.16667

5. Degree of Chaos

As in the paper, the degree of chaos for a given energy is estimated by the relative area covered by the invariant curves to the total area of the phase space. What we use is a typical hit-or-miss Monte-Carlo method, whereby a sample of random points are picked up from the phase space, and evolved to see if they belong to an invariant trajectory ('hit') by the criterion given in the paper. As the sample size is small, the built-in system `rand()` of C has been sufficient.

hh_area.c

```
#include <stdio.h>
#include "henon_heiles.c"

/* Program to calculate the relative area covered by invariant curves in the
 * phase space section (x = 0, px > 0) for different energies, by Monte-Carlo
 * method (using 100 random points). Also calculates the Lyapunov Exponents for
 * the first three trajectories for each energy */
int main()
{
        int pc, invc, i, j;
        long t;
```

```
	double lda, mu, muc = 1e-4, h = 1e-3;
	double x1[4], x1new[4], x2[4];
	double einv[] = { 50.0, 25.0, 16.0, 12.0, 10.5, 10.0, 9.5, 9.0, 8.5,
			8.0, 7.0, 6.5, 6.0 };
	const int np = 100;

	srand(time(NULL));	/* initialise rand() with system time */
	printf("#Energy\t\tlambda_1\tlambda_2\tlambda_3\tRelative area\n");
	for (j = 0; j < LEN(einv); ++j) {
		printf("%lf\t", 1 / einv[j]);
		for (invc = i = 0; i < np; ++i) {
			x1[0] = 0.0;
			/* Generate random points until we get a valid one */
			do {
				x1[1] = frand(-0.5, 1.0);
				x1[3] = frand(-0.5, 0.5);
				x1[2] = px(1.0 / einv[j], x1[0], x1[1], x1[3]);
			} while (x1[2] < 0);

			copy_vec(x1, x2, 4);
			x2[1] += 1e-7;  /* Make x2 slightly separated from x1 */
			mu = 0.0;
			t = pc = 0;
			/* Calculate mu for 25 transformations of x */
			while (pc < 25) {
				rk4(h * t, x1, 4, hh_evol, h, x1new);
				rk4(h * t++, x2, 4, hh_evol, h, x2);
				/* Check if we crossed the x = 0 plane with px > 0 */
				if (x1new[0] * x1[0] < 0 && x1[2] > 0.0) {
					mu += pow(dist(x1, x2, 4), 2);
					++pc;
				}
				copy_vec(x1new, x1, 4);
			}
			if (mu < muc)
				++invc;		/* we have a 'hit' */
			/* Print lambda for only three trajectories */
			if (i < 3) {
				lda = log(dist(x1, x2, 4) * 1e7) / (h * t);
				printf("%e\t", lda);
			}
		}
		/* Print ratio of invariant trajectories to the total */
		printf("%lf\n", invc / (float) np);
	}
	return 0;
}
```

The sample size (100) is apparently small, but it produces an impressively accurate graph:

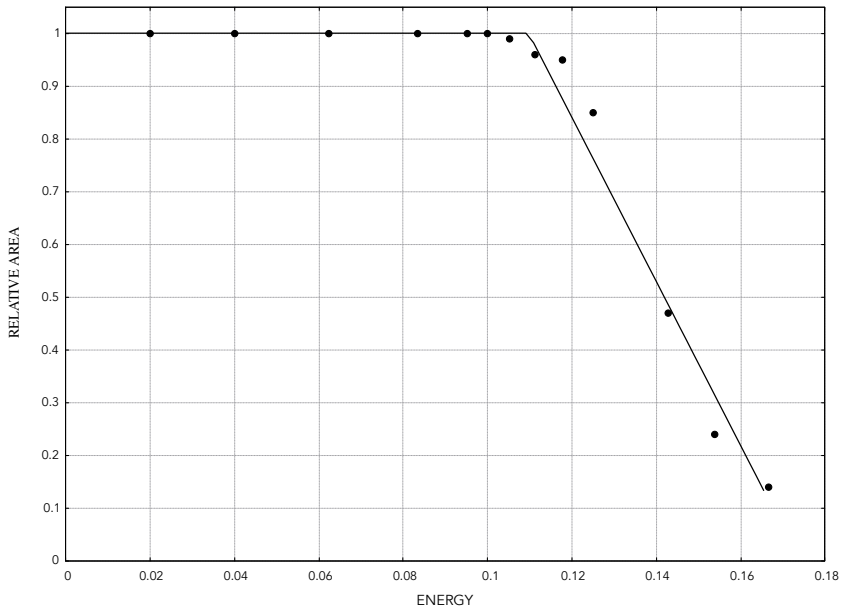

FIG. 5: Relative area covered by invariant curves as a function of energy

Two interesting deviations were observed while calculating and plotting this graph:

- Sometimes some orbits with energy slightly smaller than the estimated critical energy of 0.111 gave $\mu > \mu_c$ (this happened particularly with $E = 0.1$). This points to the sensitivity of the criterion of μ_c, as mentioned in the paper.
- For the same energy (0.1), even two points on an invariant curve drifted apart, while *remaining* on the curve, when allowed to evolve for a really large number of RK4 steps ($\sim$50 million).

As this is a transitionary region with onset of chaos, this behaviour gives some idea of how the system might be "breaking into" chaos.

The table below summarises the output of the above program, along with the Lyapunov exponent values for three trajectories:

Energy	λ_1	λ_2	λ_3	**Relative area of invariant curves**
0.020000	9.771441e-03	1.044621e-02	7.332720e-03	1.000000
0.040000	1.133927e-02	1.496972e-02	1.329128e-02	1.000000
0.062500	1.682915e-02	1.749793e-02	1.661485e-02	1.000000
0.083333	1.648947e-02	1.852790e-02	1.735407e-02	1.000000

0.095238	1.932965e-02	8.916117e-03	1.680053e-02	1.000000
0.100000	8.104184e-03	1.831143e-02	2.299707e-02	1.000000
0.105263	1.237413e-02	1.837490e-02	1.588195e-02	0.990000
0.111111	5.000967e-02	3.366436e-02	2.335504e-02	0.960000
0.117647	1.402974e-02	6.087431e-02	9.180440e-03	0.950000
0.125000	2.302307e-02	2.176195e-02	1.200965e-02	0.850000
0.142857	7.547252e-02	5.018983e-02	3.731820e-02	0.470000
0.153846	1.050241e-01	4.510402e-02	8.651894e-02	0.240000
0.166667	9.533965e-02	6.258770e-02	8.489454e-02	0.140000

As expected, λ grows with E. There is an order-of-magnitude difference of $\sim$10 for the lowest and highest energies, which is quite large for an exponent.

6. Mapping

Finally we plot the area preserving map studied at the end of the paper. Again the initial points are found by trial and error to best replicate the map, and give some additional detail. Code:

hh_map.c

```
#include <stdio.h>
#include "henon_heiles.c"

/* Program to closely reproduce the map obtained by Hénon and Heiles (fig. 8) */
int main()
{
        int i;
        /* Initial points */
        double fig8[][2] = {
                { 0.0, 0.2 }, { 0.0, 0.4 }, { 0.0, 0.48 }, { 0.0, 0.5 },
                { 0.0, 0.7 }, { 0.8, 0.0 }, { -0.3, -0.3 }, { 0.3, 0.3 },
                { 0.0, 0.7 }, { 0.8, 0.0 }, { -0.3, -0.3 }, { 0.3, 0.3 },
                { 0.0, 0.615 }, { 0.615, 0.0 }, { 0.0, 0.0 }
        };

        for (i = 0; i < LEN(fig8); ++i) {
                plot_map(fig8[i], 1.6, 2000);
                printf("\n\n");
        }
```

```
        return 0;
}
```

The result:

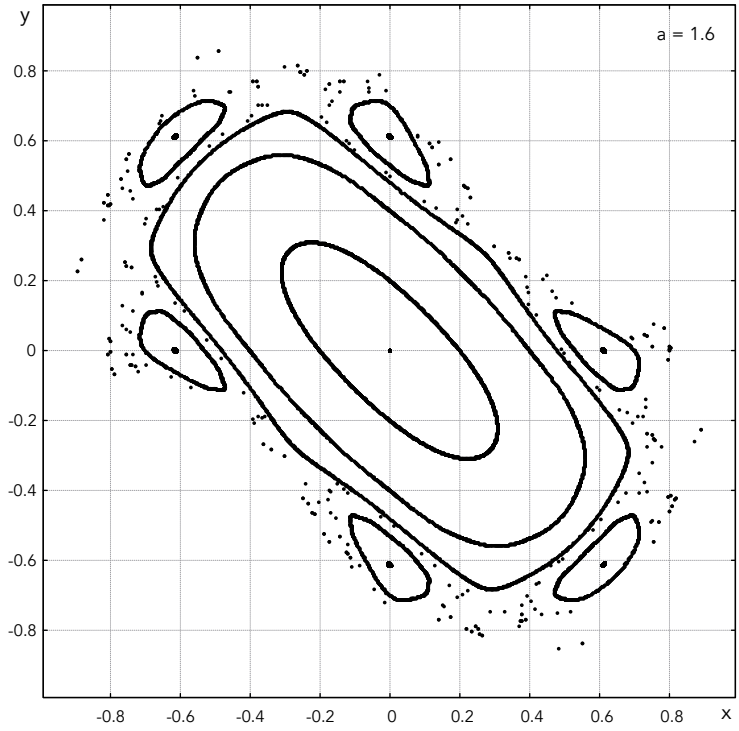

FIG. 6: The iterated map, for a = 1.6

An interesting feature of the map is that, even if we grossly increase the number of iterations, we don't get more spread of ergodic points, indicating that many of them are small invariant islands. What we have done instead was to choose a larger set of initial points and iterate a moderate number of times.

The larger islands have striking periodicity, that plotting only alternate points make some of them disappear completely.

Some utility functions have been used throughout the programs for various small tasks, like printing the elements of a vector. These functions, even thought unrelated to the physical problem at hand, have been attached below for completeness:

util.c

```
/* Utilities for general purpose programming */
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <math.h>
#include "util.h"

/* Dynamically allocate an n element array of doubles */
double *vector(int n)
{
        double *v;

        v = (double *) malloc((size_t) n * sizeof(double));
        return v;
}

/* Print the elements of an array of size n */
void print_vec(double *v, int n)
{
        for (; n > 0; --n)
                printf("%.7lf\t", *v++);
        putchar('\n');
}

/* Copy array s to t, both having size n */
void copy_vec(double *s, double *t, int n)
{
        for (; n > 0; --n)
                *t++ = *s++;
}

/* Calculate the n dimensional distance between two vectors */
double dist(double *v1, double *v2, int n)
{
        double d;

        for (d = 0.0; n > 0; --n)
                d += pow((*v1++ - *v2++), 2);
        return sqrt(d);
}

/* Return a random real number between lb and ub (both inclusive) */
double frand(double lb, double ub)
{
    return (ub - lb) * (float) rand() / RAND_MAX + lb;
}

/* Compare two real numbers; tolerance can be defined as EPS */
enum bool eq(double x, double y)
{
        if (fabs(x - y) < EPS)
                return true;
```