PH5730 : METHODS OF COMPUTATIONAL PHYSICS - ASSIGNMENT #3
Submitted on : 0 /10/2019

PH18C032
SHIVAPRASAD . V
II^ND YEAR M.Sc.
DEPT. OF PHYSICS

LYAPUNOV SPECTRUM of DYNAMICAL SYSTEMS

- by periodic Gram-Schmidt Orthonormalisation of the - time-evolving Lyapunov - vectors.

In typical non-linear systems, the smallest perturbations in the initial point in the coordinate space result in trajectories that exponentially diverge from the original trajectory, over time. This is especially true for chaotic orbits. The exponential parameter is called Lyapunov exponent, which can be different along different coordinates. e.g. along a coordinate x, if δx_0 is the initial perturbation, then :

$$\delta x(t) = e^{\lambda_1 t} \delta x_0 \qquad - (1)$$

and λ_1 is the Lyapunov exponent along x. (limiting value of) A d-dimensional system will have d such exponents, one along each of its coordinate direction. Hence, the initial perturbation, in general, is a d-dim. vector $\delta \vec{x}_0 \equiv \vec{w}_0$, called the Tangent vector or the Lyapunov vector ; hence,

$$|\vec{w}(t)| = e^{\lambda t} |\vec{w}(0)| \qquad - (2)$$

Thus the Lyapunov exponent can be determined :-

$$\lambda = \lim_{t \to \infty} \frac{1}{t} \ln \left| \frac{\vec{w}(t)}{\vec{w}(0)} \right| \qquad - (3)$$

The problem with this approach is that, as the perturbations are exponentially growing, $\left|\frac{\vec{w}(t)}{\vec{w}(0)}\right|$ may easily blow up to a huge value, overflowing the storage limits of ordinary computational storage-classes. But to get an accurate value of λ, we (really) need to do as large enough number of iterations (as $t \to \infty$).

Another issue we face is, the evolving vector will soon align itself along that direction, which is growing the fastest, thus overshadowing information about the growth (or contraction) along other directions. To mitigate both these problems, we periodically orthonormalise the tangent vectors, making use of the Gram-Schmidt procedure.

* G.S. orthonormalisation :-

Given a set of n linearly independent vectors, $\{\vec{v}_k\}$, a new set of n orthonormal vectors can be constructed as by recursively appling the formula :-

$$\boxed{\vec{u}_k^{(i)} = \vec{u}_k^{(i-1)} - \text{proj}_{u_i}(\vec{u}_k^{(i-1)})} \quad \text{— (4)}$$

$i = 1, 2 \ldots (k-1)$
$k = 1, 2, 3 \ldots n$

with $\vec{u}_k^{(0)} = \vec{v}_k$ & final $\vec{u}_k = \dfrac{\vec{u}_k^{(k-1)}}{\|u_k^{(k-1)}\|}$

In eq. (4), $\text{proj}_u(\vec{v})$ denotes the "projection" of vector $\vec{v}$ along $\vec{u}$, defined as :-

$$\text{proj}_u(\vec{v}) = (\vec{v} \cdot \hat{u})\,\hat{u} = \frac{(\vec{v} \cdot \vec{u})}{(\vec{u} \cdot \vec{u})}\,\vec{u} \quad \text{— (5)}$$

This is called the "modified" G.S. procedure. As given by eq. (4), it is numerically stable, compared to the standard G.S. procedure which is numerically unstable.

* Application to Lyapunov-spectrum calculation :-

We first start with 'd' orthonormal vectors, where d is the dimension of the system. A natural choice is the rows of Identity, i.e., $\mathbb{I}_d$. Then, we evolve each of them in time. e.g., a flow is given by an eqn. of the form:

$$\dot{\vec{x}} = f_{\{\mu\}}(\vec{x}) \quad \text{— (6)}$$

where $\{\mu\}$ is a set of parameters of the system.

$$\therefore \delta\dot{\vec{x}} = \delta\left[f_{\{\mu\}}(\vec{x})\right]$$

$$= \vec{\nabla} f_{\{\mu\}} \cdot \delta\vec{x}$$

as $\delta\vec{x} \equiv \vec{w}$, $\delta\dot{\vec{x}} \equiv \dot{\vec{w}}$ $\quad \{\delta\dot{\vec{x}} = \delta\frac{d(\vec{x})}{dt} = \frac{d}{dt}(\delta\vec{x}) = \frac{d\vec{w}}{dt}$

hence, $\boxed{\dot{\vec{w}} = J\vec{w}}$ — (7)

where J is the matrix $\vec{\nabla}_{\vec{x}} f_{[\vec{x}]}$ is also the Jacobian of the transformation, (6). After a small no. of steps (say 100), we orthonormalise the set of tangent vectors $\{\vec{w}_k\}$ by the modified G.S. procedure (4), storing the normalisation constant of each vector as $\{\alpha_k^{(j)}\}$, where j indicates time-step:

~~$\alpha_k^{(j)} = w_k^{(j)} \|\vec{u}_k^{(j)}\|$~~ —

$$\boxed{\alpha_k^{(j)} = \|\vec{u}_k^{(k-1)}(t_j \Delta t)\|} \quad — (8)$$

where $\vec{u}_k^{(k-1)}$ is the orthogonal vector just before normalisation. Thus, the final vector-set, will ~~have~~ have been scaled by:

$$|\vec{w}_k(t)| = \alpha_k^{(1)} \alpha_k^{(2)} \ldots\ldots \alpha_k^{(n)} \underbrace{|\vec{w}_k(0)|}_{1} \quad \{t = n\Delta t$$

$\therefore$ eq. (3) $\Rightarrow$ $\lambda_k = \frac{1}{n\Delta t} \ln |\alpha_k^{(1)} \alpha_k^{(2)} \ldots\ldots \alpha_k^{(n)}|$

$$\Rightarrow \boxed{\lambda_k = \frac{1}{n\Delta t} \sum_{i=1}^{n} \ln \alpha_k^{(i)}} \quad — (9)$$

Hence we are able to convert a diverging product of $\alpha_k^{(i)}$'s into a sum of their logarithms, which scales the value down well within the limits of storage-classes. The periodic orthogonalisation prevents all vectors from aligning along one direction.

The largest of $\{\lambda_k\}$ is called Largest or Maximal Lyapunove exponent. Also, another check on λ_k's is :-

$$\sum_k \lambda_k = \mathrm{Tr}(J) \quad — (10)$$

is called the net Lyapunov exponent of the system. For dissipative systems, $\sum_k \lambda_k < 0$ & for Hamiltonian systems (non-dissipative), $\sum_k \lambda_k \doteq 0$

$=$.

* The Systems studied in this work — Lorenz, Rössler & Henon-Heiles.

— P.T.O —

1) Lorenz System :-

coordinate evolution

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho(x - z) - y$$
$$\dot{z} = xy - \beta z$$

Typical parameters :-

$\sigma = 10$; $\rho = 28$ or 60 ; $\beta = \frac{8}{3}$

tangent evolution

$$\begin{bmatrix} \dot{w}_1 \\ \dot{w}_2 \\ \dot{w}_3 \end{bmatrix} = \underbrace{\begin{bmatrix} -\sigma & \sigma & 0 \\ (\rho - z) & -1 & -x \\ y & x & -\beta \end{bmatrix}}_{J} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$Tr(J) = -\sigma - 1 - \beta < 0$$

(dissipative)

2) Rössler attractor :-

coordinate evolution

$$\dot{x} = -y - z$$
$$\dot{y} = x + ay$$
$$\dot{z} = b + z(x - c)$$

Typical parameters :-

$a = 0.1$; $b = 0.1$; $c = 14$

tangent evolution

$$\begin{bmatrix} \dot{w}_1 \\ \dot{w}_2 \\ \dot{w}_3 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -1 \\ 1 & a & 0 \\ z & 0 & (x - c) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$Tr(J) = a + x - c$$

3) Hénon-Heiles Hamiltonian system :-

coordinate evolution

$$\dot{x} = p_x$$
$$\dot{y} = p_y$$
$$\dot{p}_x = -x - 2xy$$
$$\dot{p}_y = -y - x^2 + y^2$$

Parameters : $E = 1/6 = 0.166\ldots$

tangent evolution

$$\begin{bmatrix} \dot{w}_1 \\ \dot{w}_2 \\ \dot{w}_3 \\ \dot{w}_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -(2y+1) & -2x & 0 & 0 \\ -2x & -1+2y & 0 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

$$Tr(J) = 0 \quad \text{(non-dissipative)}$$

---

* Additional notes :-

* As all these evolution equations are ODE's, RK4 algorithm is used to evolve them. A d-dimensional system has $[d(d+1)]$ components (one coordinate vector and d tangent vectors, each having d-components) to be evolved simultaneously. Hence, 1) & 2) uses 12-dim RK4 & 3) uses 20-dim RK4

* The Lorenz & Rössler attractors were also plotted for verification

1. Program Organisation

- Gram-Schmidt Procedure: This require some vector manipulation operations like *dot product, subtraction, scalar-multiplication, normalization,* etc. which C-arrays do not support natively. So helper functions were written for each of these operations, which are appended at the end. Only the main functions are given below:

util.c:

```
#include <stdlib.h>
#include "util.h"

/* Orthonormalize a set of <m> n-dimensional vectors by Gram-Schmidt procedure *
 * and store their original norms in <a> */
void ortho(double **u, int m, int n, double *a)
{
        int i, k;
        double *p;

        for (k = 0; k < m; ++k) {
                /* Subtract components along all u[i < k] from u[k] */
                for (i = 0; i < k; ++i) {
                        p = proj(u[k], u[i], n);
                        sub(u[k], p, n);
                        free(p);
                }
                /* Normalize u[k] & store its original norm */
                a[k] = norm(u[k], n);
        }
}

/* Return the projection of v along u as a dynamically allocated vector *
 * proj(v, u) = v.u * u / u.u */
double *proj(double *v, double *u, int n)
{
        double a, *t;

        t = vector(n);
        copy_vec(u, t, n);
        a = dot(u, u, n);
        if (a == 0.0)
                scale(t, n, 0.0);       /* projection along zero vector = 0 */
        else
                scale(t, n, dot(u, v, n) / a);
        return t;
}
```

- Main program: Contains a very general Lyapunov-spectrum calculator function that accepts the tangent-derivative function (passable to the RK4 integrator) of a smooth dynamical system of any dimension.

  The evolution equations for both coordinates and tangents of Lorenz and Rössler-attractor systems are included in the main program itself, while those of Hénon-Heiles system are imported from the earlier source file (assignment 1):

lyapunov.c:

```
/* Program to compute the Lyapunov spectrum of various smooth dynamical systems
 * by the  method of tangent-vectors evolution with periodic Gram-Schmidt ortho-
 * normalization.
 * Language: C  (standard: ANSI C)                                Version: 1.0
 * Author: Shivaprasad V                                          Date: 5 Oct 2019
 * Credentials: PH18C032, M.Sc. Physics '18-'20, IIT Madras, IND
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "util.h"
#include "henon_heiles.h"
/* Lorenz attractor parameters */
#define SIGMA 16.0
#define RHO 45.92
#define BETA 4.0
/* Rössler attractor parameters */
#define A 0.15
#define B 0.20
#define C 10.0

/* Compute the time derivatives of coordinates r[] & store them in rdot[] */
void lorenz(double t, double r[3], double rdot[3])
{
        double x, y, z;
        x = r[0], y = r[1], z = r[2];

        /* Lorenz equations */
        rdot[0] = SIGMA * (y - x);
        rdot[1] = x * (RHO - z) - y;
        rdot[2] = x * y - BETA * z;
}

/* Compute the time derivative of tangents in phase space for Lorenz system.
 * w[] should contain the position vector of the current point, followed by
 * the 3 tangent vectors => 12 components; same order for derivatives in wdot[]
 */
void lortan(double t, double w[12], double wdot[12])
```

```
{
        int i;
        double x, y, z, w1, w2, w3;
        x = w[0], y = w[1], z = w[2];   /* store the current x, y, z coord. */

        lorenz(t, w, wdot);             /* evolve the coordinates part */
        for (i = 3; i < 12; i += 3) {   /* loop through each tangent vector */
                w1 = w[i], w2 = w[i + 1], w3 = w[i + 2];
                /* Lorenz system tangent evolution equations */
                wdot[i] = SIGMA * (w2 - w1);
                wdot[i + 1] = (RHO - z) * w1 - w2 - x * w3;
                wdot[i + 2] = x * w2 + y * w1 - BETA * w3;
        }
}

/* Compute the time derivatives of coordinates r[] & store them in rdot[] */
void rossler(double t, double r[3], double rdot[3])
{
        double x, y, z;
        x = r[0], y = r[1], z = r[2];

        /* Rössler attractor equations */
        rdot[0] = -y - z;
        rdot[1] = x + A * y;
        rdot[2] = B + z * (x - C);
}

/* Compute the time derivative of tangents in phase space for Rössler attractor.
 * Identical to lortan() above */
void rosstan(double t, double w[12], double wdot[12])
{
        int i;
        double x, y, z, w1, w2, w3;
        x = w[0], y = w[1], z = w[2];   /* store the current x, y, z coord. */

        rossler(t, w, wdot);            /* evolve the coordinates part */
        for (i = 3; i < 12; i += 3) {   /* loop through each tangent vector */
                w1 = w[i], w2 = w[i + 1], w3 = w[i + 2];
                /* Rössler attractor tangent evolution equations */
                wdot[i] = -w2 - w3;
                wdot[i + 1] = w1 + A * w2;
                wdot[i + 2] = z * w1 + (x - C) * w3;
        }
}

/* Compute and print the Lyapunov spectrum of a <d> dimensional system whose
 * tangent-evolution is specified by <tanevol>. r[] is the initial position-
 * vector & <b> is the base to which Lyapunov exponents are to be calculated. */
```

```
void lyaspec(deriv_func tanevol, double r0[], int d, double b)
{
        const int n = 1e7;              /* no. of RK4 steps */
        const int l = (d + 1) * d;      /* no. of components to evolve */
        int i, t;
        double h, *w, *a, *lna, **v;

        /* Initialize */
        w = vector(l);          /* to store coordinates + tangent components */
        a = vector(d);          /* to store scaling factors @ each ortho-step */
        lna = vector(d);        /* to store sum of log of scaling factors */
        set(w, l, 0.0);
        set(lna, d, 0.0);
        copy_vec(r0, w, d);     /* store coordinates at the beg of w */
        /* Store pointers to the tangent vectors, for passing them to ortho() */
        v = (double **) malloc((size_t) d * sizeof(double *));
        for (i = 0; i < d; ++i) {
                v[i] = &w[(i + 1) * d];         /* pointer to i-th tan vector */
                v[i][i] = 1.0;                  /* make orthonormal initially */
        }
        /* Evolution */
        h = 0.001, t = 0;
        while (t < n) {
                rk4(h * t++, w, l, tanevol, h, w);
                if (t % 100 == 0) {     /* orthonormalise at every 100th step */
                        ortho(v, d, d, a);
                        /* Sum over the natural log of each scaling factor */
                        for (i = 0; i < d; ++i)
                                lna[i] += log(a[i]);
                }
        }
        /* Divide by total time and convert the exponent to base <b> */
        scale(lna, d, 1.0 / (n * h * log(b)));
        print_vec(lna, d, "%.2lf\t");
        /* Cleanup */
        free(w); free(a); free(lna); free(v);
}

/* Main program: compute the Lyapunov spectrum (to the base 'e') of 3 systems */
int main()
{
        double egy, r[4];

        /* Lorenz and Rössler attractors */
        r[0] = 1.0; r[1] = 0.0; r[2] = 5.0;
        printf("Lyapunov spectra (to the base e):\n\n");
        printf("Lorenz System: with σ = %.2lf, ρ = %.2lf, β = %.2lf\n"
                "λ1, λ2, λ3:\t", SIGMA, RHO, BETA);
```

```
        lyaspec(lortan, r, 3, M_E);
        putchar('\n');
        printf("Rössler Attractor: with a = %.2lf, b = %.2lf, c = %.2lf\n"
               "λ1, λ2, λ3:\t", A, B, C);
        lyaspec(rosstan, r, 3, M_E);
        putchar('\n');
        /* Hénon-Heiles system: for an invariant trajectory */
        egy = 1 / 6.0;
        r[0] = r[3] = 0.0;
        r[1] = 0.5;
        r[2] = px(egy, r[0], r[1], r[3]);
        printf("Hénon-Heiles potential: Invariant trajectory with E = %.3lf\n"
               "λ1, λ2, λ3, λ4:\t", egy);
        lyaspec(hh_tan, r, 4, M_E);
        putchar('\n');
        /* Hénon-Heiles system: for an ergodic trajectory */
        r[1] = 0.3;
        r[2] = px(egy, r[0], r[1], r[3]);
        printf("Hénon-Heiles potential: Ergodic trajectory with E = %.3lf\n"
               "λ1, λ2, λ3, λ4:\t", egy);
        lyaspec(hh_tan, r, 4, M_E);
        return 0;
}
```

henon_heiles.c:

```
...
/* Compute the time derivative of tangents in phase space for H-H potential.
 * w[] should contain the position vector of the current point, followed by
 * the 4 tangent vectors => 20 components; same order for derivatives in wdot[]  */
void hh_tan(double t, double w[20], double wdot[20])
{
        int i;
        double x, y, w1, w2;
        x = w[0], y = w[1];             /* store the current x, y coordinates */

        hh_evol(t, w, wdot);            /* evolve the coordinates part */
        for (i = 4; i < 20; i += 4) {   /* loop through each tangent vector */
                w1 = w[i], w2 = w[i + 1];
                /* Hénon-Heiles tangent evolution equations */
                wdot[i] = w[i + 2];
                wdot[i + 1] = w[i + 3];
                wdot[i + 2] = -(2 * y + 1) * w1 - 2 * x * w2;
                wdot[i + 3] = -2 * x * w1 + (2 * y - 1) * w2;
        }
}
...
```

2. Output

Main program output:

```
Lyapunov spectra (to the base e):

Lorenz System: with σ = 16.00, ρ = 45.92, β = 4.00
λ1, λ2, λ3:     1.50    -0.00   -22.50

Rössler Attractor: with a = 0.15, b = 0.20, c = 10.00
λ1, λ2, λ3:     0.09    0.00    -9.80

Hénon-Heiles potential: Invariant trajectory with E = 0.167
λ1, λ2, λ3, λ4: 0.00    0.00    -0.00   -0.00

Hénon-Heiles potential: Ergodic trajectory with E = 0.167
λ1, λ2, λ3, λ4: 0.10    0.00    -0.00   -0.10
```

Plots (code not attached):

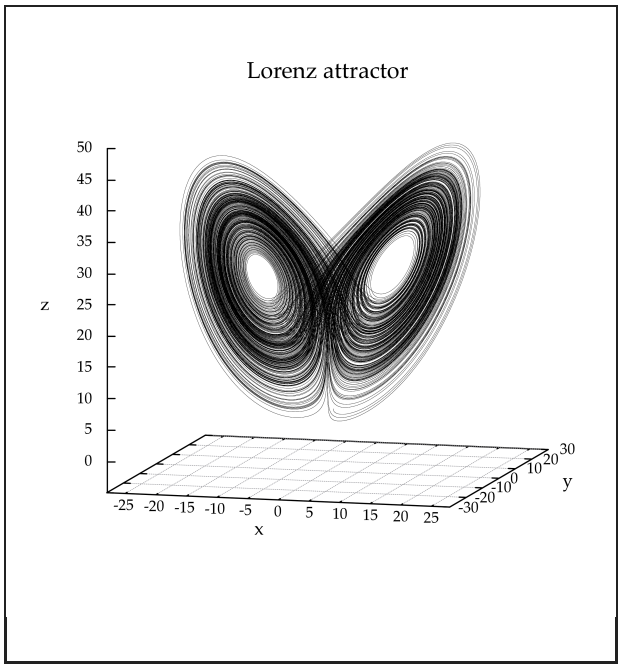

Fig. 1

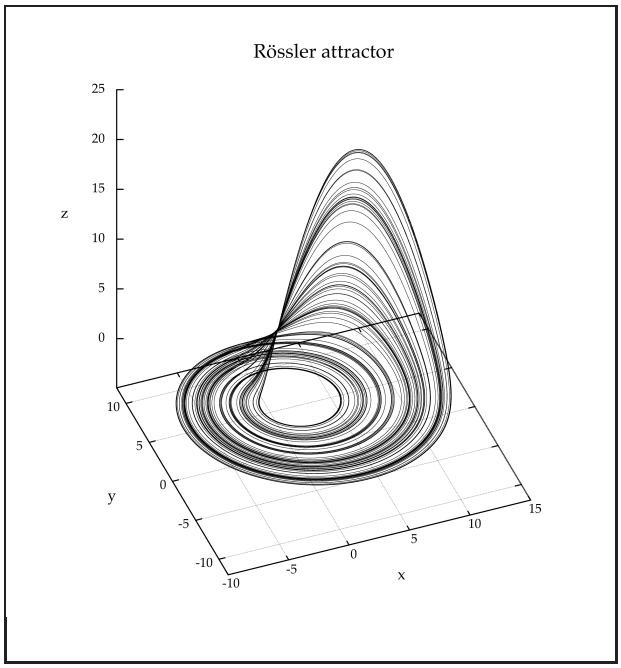

Fig. 2

3. Comparison

Lorenz system

Parameters	**Lyapunov Spectrum (base e)**	**Lyapunov Spectrum (base 2)**	**Reference values**[†] **(base 2)**
$\sigma = 16.0$	$\lambda 1 = 1.50$	2.17	2.16
$\rho = 45.92$	$\lambda 2 = 0.00$	0.00	0.00
$\beta = 4.00$	$\lambda 3 = -22.50$	-32.46	-32.4

Rössler system

Parameters	Lyapunov Spectrum (base e)	Lyapunov Spectrum (base 2)	Reference values† (base 2)
a = 0.15 b = 0.20 c = 10.0	$\lambda 1 = 0.09$ $\lambda 2 = 0.00$ $\lambda 3 = -9.80$	0.13 0.00 −14.14	0.13 0.00 −14.1

Hénon–Heiles system

Parameters	Lyapunov Spectrum – invariant curve	Lyapunov Spectrum – ergodic curve
E = 0.167	$\lambda 1 = 0.00$ $\lambda 2 = 0.00$ $\lambda 3 = 0.00$ $\lambda 4 = 0.00$	0.10 0.00 0.00 −0.10
$\Sigma\lambda_k$	0.00	0.00

†Ref.: *Determining Lyapunov exponents from a time series,* A. Wolf et al., p. 289

The obtained values are found to match closely with the reference values. The sum of Lyapunov exponents are also found to be equal to the traces of the corresponding Jacobians, as explicitly shown in the last case.

4. Appendix

Helper functions for Gram-Schmidt orthonormalization function:

util.c:

```
/* Utilities for genral purpose programming */
#include <stdio.h>
#include <math.h>
#include "util.h"

/* Set all the elements of vector v to k */
void set(double *v, int n, double k)
{
        for (; n > 0; --n)
                *v++ = k;
}

/* Compute inner-product of two vectors: u.v */
double dot(double *u, double *v, int n)
{
        double s;
```

```
        for (s = 0.0; n > 0; --n)
                s += *u++ * *v++;
        return s;
}

/* Multiply vector by a scalar k */
void scale(double *v, int n, double k)
{
        for (; n > 0; --n)
                *v++ *= k;
}

/* Subtract vector u from v: v - u */
void sub(double *v, double *u, int n)
{
        for (; n > 0; --n)
                *v++ -= *u++;
}

/* Normalize given vector v and return its original norm */
double norm(double *v, int n)
{
        double a;

        a = sqrt(dot(v, v, n));
        if (a != 0.0) {         /* don't normalize zero vector */
                for (; n > 0; --n)
                        *v++ /= a;
        }
        return a;
}

/* Print the elements of an array of size n in given format */
void print_vec(double *v, int n, const char *fmt)
{
        for (; n > 0; --n)
                printf(fmt, *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++;
}
```