Math 6307 Course Project: Modern methods of exploration of numerical ODE solutions using Python3

Maxie Dion Schmidt

Georgia Institute of Technology
School of Mathematics

maxieds@gmail.com
mschmidt34@gatech.edu

http://people.math.gatech.edu/∼mschmidt34/
https://github.com/maxieds/GATechMath6307ODEsCourseProject

November 2, 2021

(Last compiled with LaTeX2e on October 28, 2021)

Goals of the presentation

- Explore options of modern packages for Python3 that facilitate exploring ODE (systems of ODEs) solutions numerically
- Give a few examples of generic, **re-usable** numerical methods for a toy 1D ODE problem
- Define and motivate the study of *chaotic attractors* (corresponding to parameterized multidimensional systems of ODEs – typically 3D and 4D in a time variable, or 2D projections of such systems)
- Show some particular experiment types for the *Rössler attractor* that can be extended to other applications and use cases
- All source code, presentation materials and package install notes for this project are freely available under the GPL-V3 at `https://github.com/maxieds/GATechMath6307ODEsCourseProject`

Basic examples

Basic examples of solving ODEs in Python3

Setup: Defining a common model problem

- Typically we look at an IVP of the following form: $y' = f(t, y), y(t_0) = y_0$
- For the purposes of exploring our options in Python3, we will take a special case of this problem type
- The special case is defined for some parameter k as: $y' = (t - y^k)(3 - ty - 2y^2)$ subject to $y(0) = 1$
- That is: $f(t, y) := (t - y^k)(3 - ty - 2y^2)$ with $(t_0, y_0) := (0, 1)$
- The analysis of this "toy" model problem facilitates comparing the functionality and ease of use for numerically approximating its solutions (for various fixed k) of common libraries and algorithms in Python3

Explicit forward Euler method

- Given an IVP: $y' = f(t, y), y(t_0) = y_0$
- The forward Euler method is an explicit method for iteratively generating numerical solutions to this ODE provided that we can evaluate f clearly
- Convergence to solution and other numerical analysis of the algorithm (e.g., LTE and GTE truncation errors at each step) and corresponding accuracy given fixed h is standard reading
- $y_{n+1} = y_n + f(t_n, y_n)\,(t_{n+1} - t_n) = y_n + h \cdot f(t_n, y_n)$

Explicit forward Euler method – Motivation

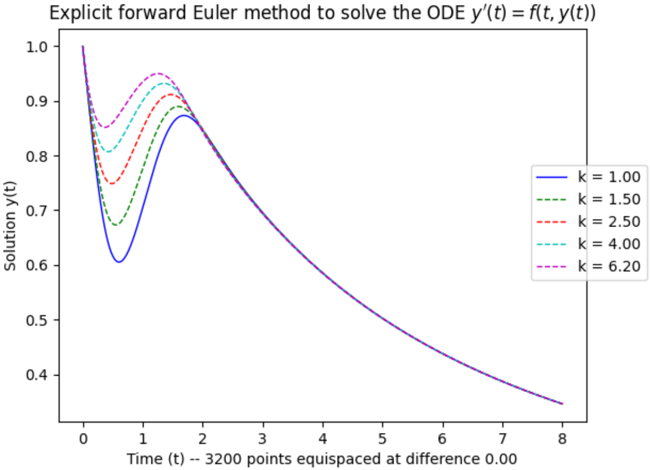
- **Key derivation:** $y'(t_0) \approx \frac{y(t_0+h)-y(t_0)}{h}$ $(*)$, where $y' = f(t, y) \implies y(t_0 + h) - y(t_0) \approx \int_{t_0}^{t_0+h} f(t, y(t))dt$
- Now we approximate the the RHS integral using a left-hand-endpoint Riemann sum ($n = 1$ rectangle) to obtain that $\int_{t_0}^{t_0+h} f(t, y(t))dt \approx h \cdot f(t_0, y(t_0))$
- Forward Euler is the simplest method using this line of reasoning
- Modifications can be given, including taking $y'\left(t + \frac{h}{2}\right) \approx \frac{y(t+h)-y(t)}{h}$ in place of $(*)$ above, leading to the *midpoint method*
- Implicit backwards Euler forms another variant

Explicit foward Euler method (source code flavor)

```
def ExplicitForwardEuler(ftyFunc, icPoint, solInterval, h):
    f = ftyFunc
    (t0, y0) = icPoint
    (solA, solB) = solInterval
    numGridPoints = math.floor(float((solB - solA) / h))
    tPoints = np.linspace(solA, solB, numGridPoints + 1)
    yPoints = [ y0 ]
    curYn = y0
    for n in range(0, numGridPoints):
        tn = tPoints[n]
        nextYn = curYn + f(tn, curYn) * h
        yPoints += [ nextYn ]
        curYn = nextYn
    # ... Matplotlib plotting code for the solution ...
```

Explicit foward Euler method (results)

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run "ExplicitForwardEulerMethod.py"
```



Runga-Kutta (RK4) method

- Given an IVP: $y' = f(t, y), y(t_0) = y_0$
- $t_n = t_0 + h \cdot n$ (fixed / uniform step size)
- $k_1 = f(t_n, y_n)$, $k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{k_1 h}{2}\right)$, $k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{k_2 h}{2}\right)$, $k_4 = f\left(t_n + h, y_n + k_3 h\right)$
- $y_{n+1} = y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$ (recursion to evaluate)
- Taking the weighted average at four slopes leads to more weight given to slopes closer to the midpoint of each subinterval
- LTE $= O(h^5)$ and AccumulatedTruncationError $= O(h^4)$
- If f does not depend on y, the RK4 is the same as *Simpson's rule*

More general explicit Runga-Kutta methods

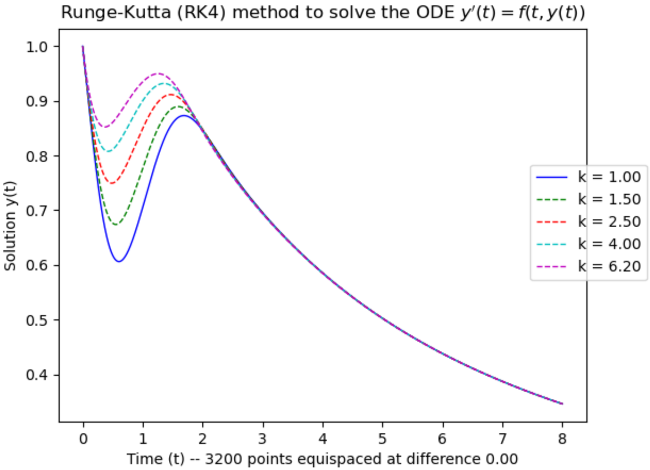
- The family of *explicit RK methods* (s-stage) is paramterized by:
  $y_{n+1} = y_n + \sum_{i=1}^{s} hb_i k_i$ where
  $k_1 = f(t_n, y_n)$
  $k_2 = f(t_n + c_2 h, y_n + h \cdot (a_{21} k_1))$
  $k_3 = f(t_n + c_3 h, y_n + h \cdot (a_{31} k_1 + a_{32} k_2))$
  $\dots$
  $k_s = f(t_n + c_s h, h \cdot (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}))$.
- That is, the explicit RK method is completely determined by the parameters $(a_{ij})_{1 \leq j < i \leq s}$, $(b_i)_{1 \leq i \leq s}$ and $(c_j)_{2 \leq j \leq s}$
- We require *consistency* insomuch as $\sum_{i=1}^{s} b_i = 1$
- A popular convention is to require that $\sum_{j=1}^{i-1} a_{ij} = c_i$ for $i \in \{2, 3, \dots, s\}$

Runga-Kutta (RK4) method (source code flavor)

```python
def RungeKuttaRK4(ftyFunc, icPoint, solInterval, h):
    f = ftyFunc
    (t0, y0) = icPoint
    (solA, solB) = solInterval
    numGridPoints = math.floor(float((solB - solA) / h))
    tPoints = np.linspace(solA, solB, numGridPoints + 1)
    yPoints = [ y0 ]
    curYn = y0
    for n in range(0, numGridPoints):
        tn = tPoints[n]
        k1 = f(tn, curYn)
        k2 = f(tn + h / 2.0, curYn + k1 * h / 2.0)
        k3 = f(tn + h / 2.0, curYn + k2 * h / 2.0)
        k4 = f(tn + h, curYn + k3 * h)
        nextYn = curYn + h / 6.0 * (k1 + 2 * k2 + 2 * k3 +
            k4)
        yPoints += [ nextYn ]
        curYn = nextYn
```

Runga-Kutta (RK4) method (results)

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run ”ImplicitRungeKuttaMethod.py”
```



Overview of general multistep methods

- General multistep method with s steps:

$$y_{n+s} + \sum_{j=0}^{s-1} a_j y_{n+j} = \sum_{m=0}^{s} h b_m f(t_{n+m}, y_{n+m})$$

- Polynomial interpolation:

$$p(t_{n+i}) := f(t_{n+i}, y_{n+i}), \text{ for } i \in \{0, 1, \ldots, s-1\}$$

- Lagrange's exact polynomial interpolation formula under this requirement:

$$p(t) = \sum_{0 \leq j < s} \frac{(-1)^{s-j-1} f(t_{n+j}, y_{n+j})}{j!(s-j-1)!h^{s-1}} \times \prod_{\substack{0 \leq i < s \\ i \neq j}} (t - t_{n+i})$$

- Approximations to initial conditions and foundation for building the numerical solutions:

$$y_n = y_{n-1} + \int_{t_{n-1}}^{t_n} p(t) dt, \text{ for } n \in \{1, 2, \ldots, s-1\}$$

Step solver method – Adams-Bashforth (ABF-s)

- The multistep ABF-s (s-step) case: $a_{s-1} = -1$; $a_{s-2} = \cdots = a_0 = 0$
- The multistep ABF-s (s-step) case: Substitute $f(t_{n+j}, y_{n+j}) \mapsto p(t_{n+j})$ in the Lagrange interpolation formula from above
- The ABF-s method coefficient multipliers yield:

$$b_{s-j-1} = \frac{(-1)^j}{j!(s-j-1)!} \times \int_0^1 \prod_{\substack{0 \leq i < s \\ i \neq j}} (u+i) du, \text{ for } j \in \{0, 1, \ldots, s-1\}$$

- Recursion in the ABF-3 method:

$$y_{n+3} = y_{n+2} + h \left(\frac{23}{12} f(t_{n+2}, y_{n+2}) - \frac{4}{3} f(t_{n+1}, y_{n+1}) + \frac{5}{12} f(t_n, y_n) \right)$$

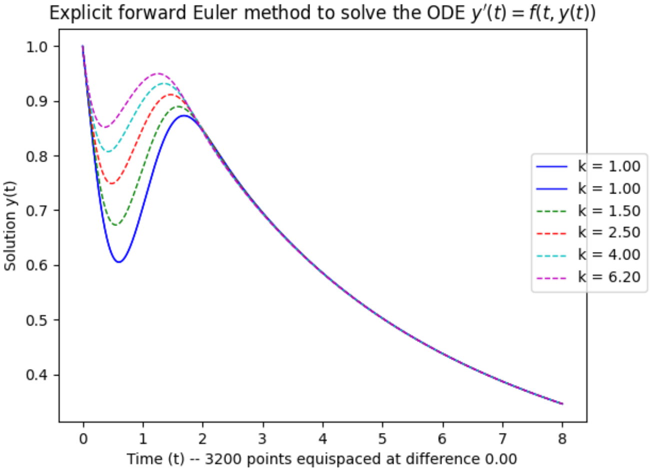
- **Note:** Single step ABF-1 is the forward Euler method

Step solver method – ABF3 (source code flavor)

```python
def LagrangePolynomialInterpolation(ftyFunc, numStepsS, prevYPoints, t0, h, n):
    f              = ftyFunc
    s              = numStepsS
    yPoints        = prevYPoints
    tDiffProdFunc = lambda t, j: reduce(operator.mul, [ t - (t0 + h * (n + i)) if i != j else 1 for i in range(0, s) ])
    ptFunc         = lambda t: sum([ (-1)**(s-j-1) * f(t0 + h * (n + j), yPoints[n+j]) / factorial(j) / \
                                    factorial(s-j-1) / (h**(s-1)) * tDiffProdFunc(t, j) \
                                    for j in range(0, n + len(yPoints)) ])
    nextYPoints    = []
    lastYPoint     = prevYPoints[-1]
    for sidx in range(0, s):
        tnpim1 = t0 + h * (n + sidx - 1)
        tnpi = t0 + h * (n + sidx)
        ynpi = lastYPoint + sympy.integrate(ptFunc(tvar), (tvar, tnpim1, tnpi))
        yPoints += [ ynpi ]
        nextYPoints += [ ynpi ]
        lastYPoint = ynpi
    return nextYPoints
def AdamsBashforthABF3(ftyFunc, icPoint, solInterval, h):
    f              = ftyFunc
    s              = 3
    (t0, y0)       = icPoint
    (solA, solB)   = solInterval
    numGridPoints = math.floor(float((solB - solA) / h))
    tPoints        = np.linspace(solA, solB, numGridPoints + 1)
    yPoints        = [ y0 ] + LagrangePolynomialInterpolation(f, s-1, [ y0 ], t0, h, n=0)
    curYn          = y0
    for n in range(0, numGridPoints + 1 - s):
        tn2, tn1, tn = tPoints[n+2], tPoints[n+1], tPoints[n]
        yn2, yn1, yn = yPoints[n+2], yPoints[n+1], yPoints[n]
        fn2, fn1, fn = f(tn2, yn2), f(tn1, yn1), f(tn, yn)
        nextYn = yn2 + h * (23.0 / 12.0 * fn2 - 4.0 / 3.0 * fn1 + 5.0 / 12.0 * fn)
        yPoints += [ nextYn ]
```

Step solver method – ABF3 (results)

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run "ImplicitStepSolverMethod.py"
```

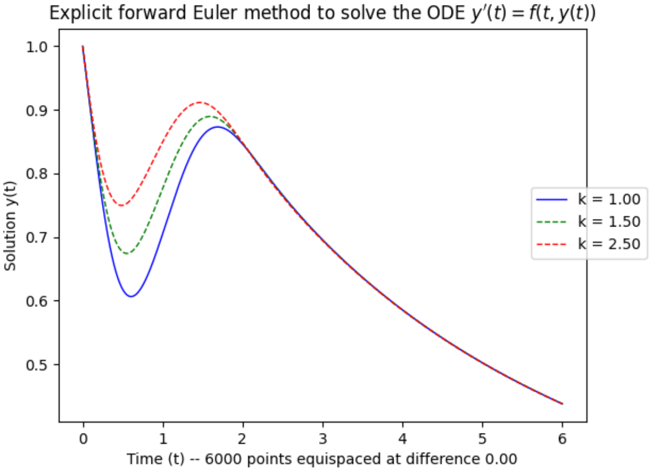


GEKKO library (source code flavor)

```
gk = GEKKO(remote=False)
def yPowerODEFunc(kpow):
    return lambda t, y: (t - y**kpow) * (3 - y * t - 2 * (y**2))
if __name__ == "__main__":
    kPowParams    = [ 1.0, 1.5, 2.5 ]
    drawStyles    = [ GetDistinctDrawStyle(n) for n in range(0, len(kPowParams)) ]
    gridSpacingH  = 0.001
    solInterval   = (0, 6.0)
    (solA, solB)  = solInterval
    icPoint       = (0, 1)
    (t0, y0)      = icPoint
    numGridPoints = math.floor(float((solB - solA) / gridSpacingH))
    gk.options.IMODE = 4
    gk.options.TIME_SHIFT = 0
    gk.options.SOLVER = 1
    axFig = plt.figure(1)
    for (kidx, kpow) in enumerate(kPowParams):
        k               = gk.Param()
        y               = gk.Var(value=y0)
        gk.time         = np.linspace(solA, solB, numGridPoints + 1)
        t               = gk.Param(value=gk.time)
        k.value         = kpow
        ftyFunc         = yPowerODEFunc(k)(t, y)
        gk.Equation(y.dt() == ftyFunc)
        gk.options.MAX_ITER = 250 * math.floor(kpow)
        gk.solve(disp=VERBOSE)
        pltDrawStyle = drawStyles[kidx]
        pltLegendLabel = "k = %1.2f" % kpow
        plt.plot(gk.time, y, pltDrawStyle, label=pltLegendLabel, linewidth=1)
    plt.xlabel("Time (t) -- %d points equispaced at difference %1.2f" % (numGridPoints, gridSpacingH))
    plt.ylabel('Solution y(t)')
    plt.title(r'Explicit forward Euler method to solve the ODE $y^{\prime}(t) = f(t, y(t))$')
    axFig.legend(loc='center right')
    plt.show()
```

GEKKO library (results)

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run "PythonGEKKOSolver.py"
```



Generating vector field plots for 2D systems

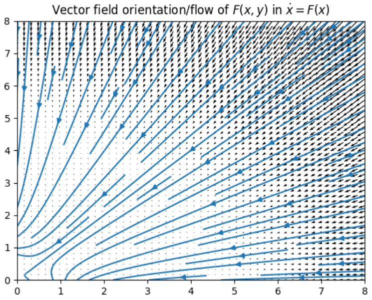
- **Problem 4 from the midterm:** Consider the non-linear ODE $\dot{x} = F(x)$ on $\mathbb{R}^2$ defined such that $F(x,y) = \left(x(1-x^2-y^2)-y, y(1-x^2-y^2)+x\right)^T$
- The function $F(x,y)$ defines a vector field in 2D
- Solutions $(x(t), y(t))$ are witnessed along a hyperbola as can be seen by evaluating the system of equations in polar coordinate
- We can get to initial grips with the solutions to this problem and visualize the vector field at hand using standard plotting functions in `matplotlib.pyplot` (imported as `plt`)

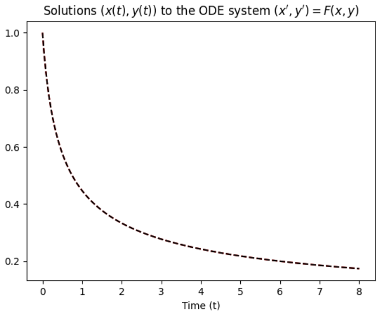
Generating vector field plots (source code flavor)

```python
from scipy.integrate import odeint

def PlotVectorField(FxyFunc, xRange, yRange):
    Fxy = lambda x, y: np.array(list(FxyFunc(x, y)))
    xGridPoints, yGridPoints = np.meshgrid(xRange, yRange)
    xv, yv = sympy.var('x y')
    (uQuiver, vQuiver) = Fxy(xGridPoints, yGridPoints)
    xmin, xmax = min(xGridPoints.flatten()), max(xGridPoints.flatten())
    ymin, ymax = min(yGridPoints.flatten()), max(yGridPoints.flatten())
    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)
    plt.quiver(xGridPoints, yGridPoints, uQuiver, vQuiver)
    plt.streamplot(xGridPoints, yGridPoints, uQuiver, vQuiver)

def SolveODE2DSystemWithVectorField(FxyFunc, icPoint, solInterval, h):
    Fxy = lambda s, time: FxyFunc(s[0], s[1])
    (t0, (x0, y0)) = icPoint
    (solA, solB) = solInterval
    numGridPoints = math.floor(float((solB - solA) / h))
    timeSpecT = np.linspace(solA, solB, numGridPoints + 1)
    odeIntSol = odeint(Fxy, [ x0, y0 ], timeSpecT)
    xtSolPoints = odeIntSol[:, 0]
    ytSolPoints = odeIntSol[:, 1]
    axFig = plt.figure(1)
    plt.xlabel(r'Time (t)')
    plt.plot(timeSpecT, xtSolPoints, GetDistinctDrawStyle(2), label=r'$x(t)$')
    plt.plot(timeSpecT, ytSolPoints, GetDistinctDrawStyle(6), label=r'$y(t)$')
```

Results

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run "ExploringVectorFieldsAndODESystems.py"
```





The `plt.quiver` function shows the magnitude of the vectors (as black arrows, above left) where the `plt.streamplot` shows the orientation/directions of the flow of the field without indicating magnitudes along the curves (in blue, above left)

Key applications for numerical exploration

Chaotic attractors

Definitions and motivation

- A very precise definition of *chaotic attractor* is developed using criteria based on topological constructions in the references (see [4, 1])
- We will stick to a high-level qualitative description motivating study of the behavior of systems of this type
- When considering dynamical systems, an *attractor* is a set of states (orbits) towards which a system (of ODE solutions) tends to evolve
- System values within some small range of the *attractor* set stay close even if perturbed slightly (e.g., by slightly shifting an ODE initial condition)
- A *chaotic attractor* is correspondingly an attractor admitting system that exhibit apparently randomized behavior and disorderly irregularities in form
- Systems that form a *chaotic attractor* type are highly sensitive to initial conditions

Famous examples of chaotic attractors

- We will focus on numerical exploration of the *Rössler attractor* system
- Other famous examples that extend applications of these numerical ideas in Python 3 include the following chaotic attractor system variants: The *Robin attractor*, the *Lorenz–63 model* (3D solutions), and the *Lorenz–96 model*
- There is much on these special cases in the literature (we do not have enough time to cover them all here)

Our prototype attractor problem for numerical investgation

Key application: The Rössler attractor

Definition of the Rössler attractor problem

- Non-linear 3D systems of ODEs determined by parameters $(a, b, c) \in \mathbb{R}^3$
- Precise system: $(x', y', z') = (-y - z, x + ay, b + z(x - c))$
- Rössler famously studied the "classic" case with $(a, b, c) := (0.2, 0.2, 5.7)$ (important characteristic properties of other parameter special cases are known)
- Often times to simplify considerations, we consider a projection of the system corresponding to setting one of the XYZ-components to zero, e.g., the projection in the XY-plane seen by setting $z := 0$

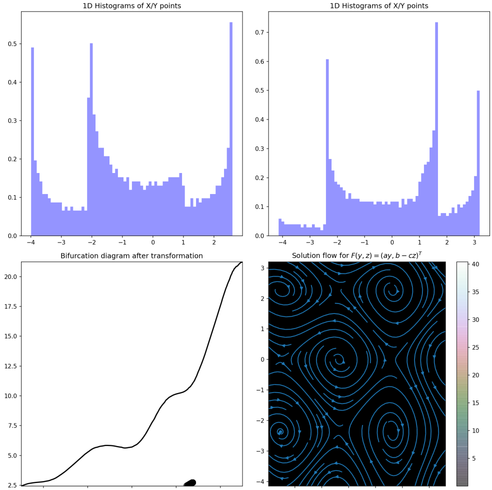
Preliminary numerical exploration of solutions

- We can use the `scipy.integration.odeint` function to numerically solve the projected system for *explicit* numerical values of the parameters (a, b, c)
- For the 3D plots, we transform the Z-component of the plot by taking the Euclidean norm of the projected point (see source code)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerMiscPlots.py"
```

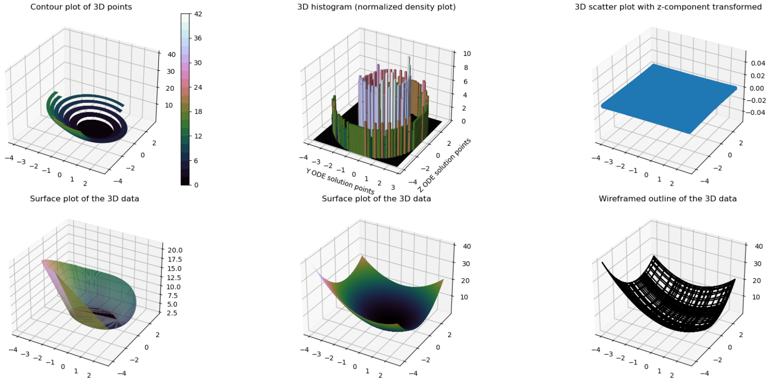
Exploring the classical parameter solution (1D plots)

The solution projected into the XY-plane (by setting $Z = 0$) with the "classic" parameters $(a, b, c) = (0.2, 0.2, 5.7)$.



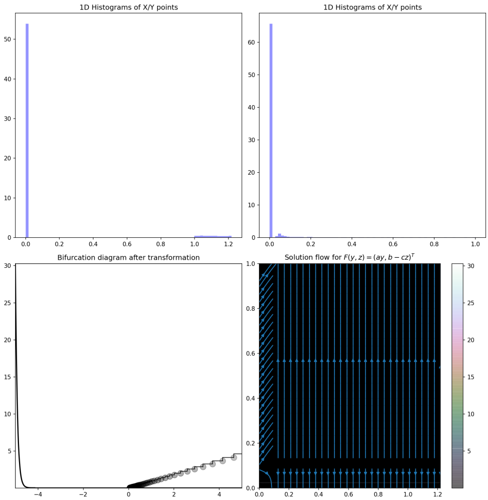
Exploring the classical parameter solution (3D plots)

The solution projected into the XY-plane (by setting $Z = 0$) with the "classic" parameters $(a, b, c) = (0.2, 0.2, 5.7)$.



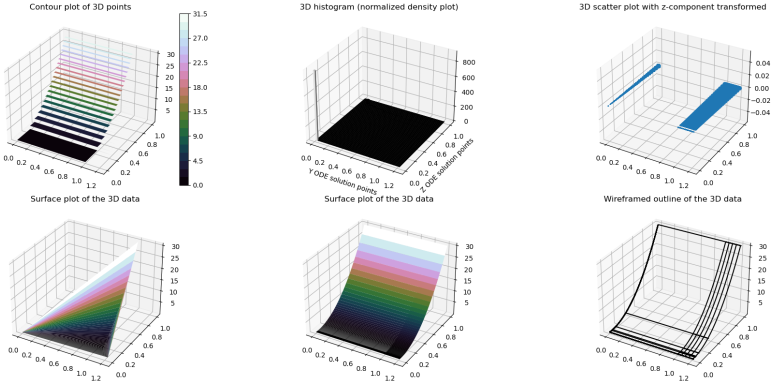
Exploring the classical parameter solution (1D plots)

The solution projected into the YZ-plane (by setting $X = 0$) with the "classic" parameters $(a, b, c) = (0.2, 0.2, 5.7)$.



Exploring the classical parameter solution (3D plots)

The solution projected into the YZ-plane (by setting $X = 0$) with the "classic" parameters $(a, b, c) = (0.2, 0.2, 5.7)$.



A modified experiment definition (Experiment V1)

- More so than a theoretically motivated example, we present Python3 source code for a numerical experiment that can be generalized and extended for use in other related applications
- Consider the following generalization of the 1D-parameter *Lyapunov coefficient* that results when $v_0 \in \mathbb{R}^2$ is a fixed constant vector and $R(x, y, z)$ is the 2D Rössler system formed by omitting the projection component:

$$\lambda(a, b, c) := \frac{1}{N} \times \sum_{0 \leq i,j,k < N} \log \left| \frac{\partial R}{\partial t}(x_i, y_j, z_k) \cdot v_0 \right|.$$

- Since $\lambda(a, b, c)$ depends symbolically on the parameters (a, b, c), we consider a relation on real values of these parameters that restricts a grid of these parameters upon which we can form a plot.
- Then for the $(a, b, c) \in \mathbb{R}^3$ that satisfy this relation, we plot a `mathplotlib.pyplot.hexbin` graph of $|\lambda(a, b, c)|$ on the Cartesian y-axis against the values of another function $\mathcal{T}_x(a, b, c)$ on the x-axis

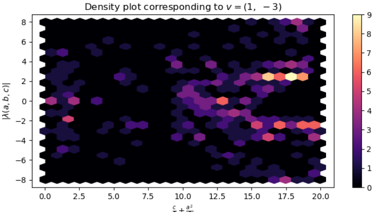
A modified experiment definition (Experiment V1)

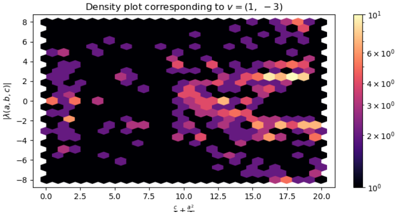
- Limitations and incompatibility: The implementation was tricky in Python3 for a few reasons:
  - There is considerable incompatibility in the types of the objects returned by large / mature Python3 libraries like `sympy`, `scipy`, `numpy` and even within `sage` (the *SageMath* CAS environment)
  - There is really no good way to solve a 3D system of ODEs when the solutions involve symbolic parameters like the unevaluated indeterminates (a, b, c)
  - Numerically evaluating the entire ODE solution for each possibility of (a, b, c) is an inefficient, time-consuming approach

Modified experiment V1 – Variant #1 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments.py"
```

Variant 1: In the projected XY-plane with linear (left) and logarithmic (right) scaling on the X-axis defined by $\mathcal{T}_x(a, b, c) := \frac{c}{a} + \frac{a^2}{c}$, subject to the restriction that $a^2 + b^2 + c^2 = 4$ (for real parameter values).

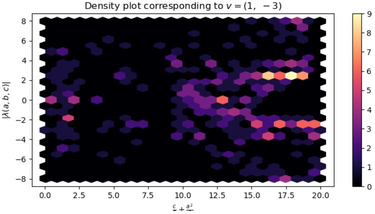


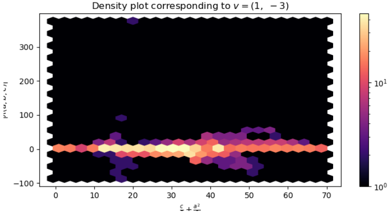


Modified experiment V1 – Variant #2 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments.py"
```

Variant 2: In the projected XY-plane with linear (left) and logarithmic (right) scaling on the X-axis defined by $\mathcal{T}_x(a,b,c) := \frac{c}{a} + \frac{a^2}{c}$, subject to the restriction that $(a-0.2)^4 - (b^2-0.2)^3 + 2(c-5.7)^2 = 4$ (for real parameter values).

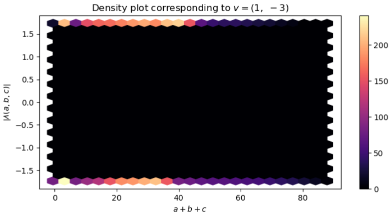


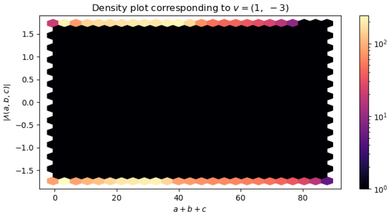


Modified experiment V1 – Variant #3 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments.py"
```

Variant 3: In the projected XY-plane with linear (right) and logarithmic (left) scaling on the X-axis defined by $\mathcal{T}_x(a,b,c) := a+b+c$, subject to the restriction that $(a+b+c)^2 = 3$ (for real parameter values).

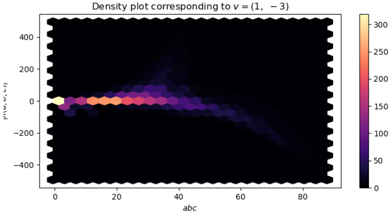


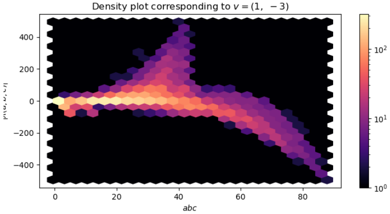


Modified experiment V1 – Variant #4 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments.py"
```

Variant 4: In the projected XY-plane with linear (right) and logarithmic (left) scaling on the X-axis defined by $\mathcal{T}_x(a, b, c) := abc$, subject to the restriction that $(a + b + c)^2 = 3$ (for real parameter values).

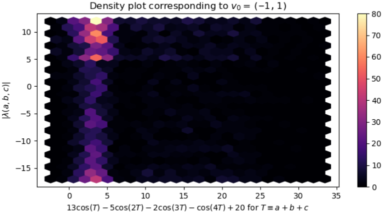


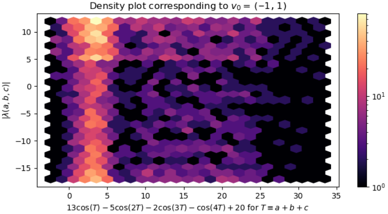


Modified experiment V1 – Variant #5 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments.py"
```

Variant 5 (Parameterizations of a heart shape in the plane): In the projected XY-plane with linear (right) and logarithmic (left) scaling on the X-axis defined by $\mathcal{T}_x(a,b,c) := 13\cos(T) - 5\cos(2T) - 2\cos(3T) - \cos(4T)$ with $T \equiv a+b+c$, subject to the restriction that $(a^2+b^2+ac)^2 = c^2(a^2+b^2)$.





Experiment V2: Problem setup

- The hexagon density plots that resulted in seemingly random choices of the X-axis functions and relations between the parameters suggest that there is more hidden underneath (e.g., some semblance of regularity to be quantified in) the definition of $\lambda(a, b, c)$
- For this experiment, we consider the projected system to XY components (setting $Z = 0$)
- The resulting 2D ODE system only depends on one parameter: a
- So we consider the following function:

$$\lambda(a, u) := \frac{1}{N^2} \times \sum_{0 \leq i,j,k < N} \log \left| (-y_{N,j}, x_{N,i} + a y_{N,j}) \cdot (u, -1) \right| .$$

- Note that we consider a uniform grid for t such that the difference between the N distinct time points goes to zero as $N \to \infty$. Then $(x_{N,i}, y_{N,j}) = (x(t_{N,i}), y(t_{N,j}))$ where the LHS functions are also implicitly functions of the (indeterminate) parameter: a

Experiment V2: Problem setup (cont'd)

- Some arithmetic yields that

$$\lambda(a,u) = \frac{1}{N} \times \underbrace{\log\left[\prod_{0\le i<N} |x_{N,i}|\right]}_{:=\lambda_{0,N}(a)} + \frac{1}{N^2} \times \log\left[\prod_{0\le i,j<N} \left|1 + \frac{(u+a)y_{N,j}}{x_{N,i}}\right|\right].$$

- Want to verify numerically that the following limit exists for each $a, u \in (-\infty, \infty)$:

$$\lambda_0(a) = \lim_{N\to\infty} \frac{\lambda_{0,N}(a)}{N}.$$

- We also pose an ansatz that (for at least some a, u) we should get a limiting probability measure when any $(i,j) \in [0,N)^2 \bigcap \mathbb{Z}^2$ is selected uniformly at random

$$\nu(a,u;t) = \lim_{N\to\infty} \mathbb{P}\left[\frac{y_{N,j}}{x_{N,i}} = t\right], t \in \mathbb{R}$$

Experiment V2: Problem setup (cont'd)

- When this happens, we have that

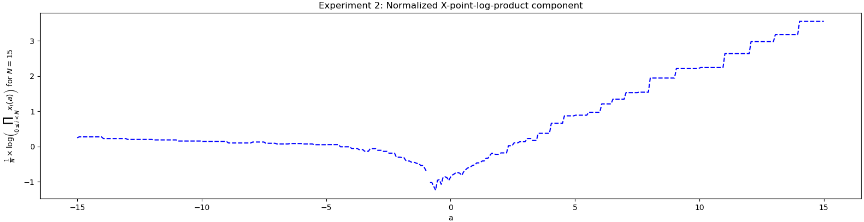
$$
\begin{aligned}
\lambda(a,u) &= \lambda_0(a) + \int_{-\infty}^{\infty} \log|1+(u+a)t|\,\nu(a,u;t)dt \\
&= \lambda_0(a) + \int_{\tau_\ell(a,u)}^{\tau_u(a,u)} \log|1+(u+a)t|\,\nu(a,u;t)dt \\
&= \lambda_0(a) + \log|1+(u+a)t|\left(\mathbb{P}\left[\Theta \le t\right] - 1\right)\Big|_{\tau_\ell(a,u)}^{\tau_u(a,u)} \\
&\quad + \int_{\tau_\ell(a,u)}^{\tau_u(a,u)} \frac{(u+a)\mathbb{P}\left[\Theta \ge t\right]}{1+(u+a)t}dt,
\end{aligned}
$$

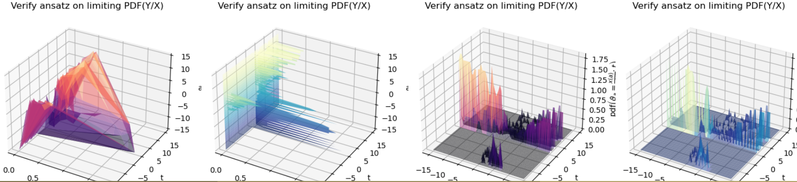
where we have $-\infty < \tau_\ell(a,u), \tau_u(a,u) < \infty$ (the measure has finite support; justified by examining other special cases of this system) and Θ is a random variable distributed such that $\nu(a,u;t)$ is its PDF

Experiment V2: Numerical methods towards the ansatz

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerGenLyapunovExponentExperiments2.py"
```

Note: It is *very* time consuming to run the above script for a fine-grained mesh that gives the $N \to \infty$ behavior. These plots show the numerical results with the plot grid taken on a mesh with of granularity $(h, \Delta a) = (0.075, 0.075)$.





Experiment V2: Observations and open questions

- If we witness the convergence of $\lambda_0(a)$ and $\nu(a, u; t)$ to a probability measure, are there then subsets of such measure inducing parameters where we get particularly "nice" distributions?
- As is typical with chaotic attractor systems, we should (probably?) expect that for any fixed (a, u) where we get this convergence, there should be some "sweet spot" $\widehat{P} = (a_0, u_0)$ such that (a, u) is in a small neighborhood of $\widehat{P}$ and at the main point we find exquisite properties given the territory.
- In running the Python script used to generate the plots on the previous slide, we have had to exclude singularities for several a where the ratio $\vartheta_{N,i,j} = y_{N,j}(a)/x_{N,i}(a) = +\infty$ blows up. What does this indicate?

Concluding remarks and discussion

The End

Questions?

Comments?

Feedback?

Thank you for attending!

References I

K. T. Alligood, T. D. Sauer and J. A. Yorke. *CHAOS: An introduction to dynamical systems* (cf. Chapter 6: *Chaotic attractors*). Springer Textbooks in Mathematical Sciences, New York, 1997.

Euler method. https://en.wikipedia.org/wiki/Euler_method

Linear multistep method. https://en.wikipedia.org/wiki/Linear_multistep_method

C. Robinson. *What is a chaotic attractor?* https://sites.math.northwestern.edu/~clark/publications/chaos.pdf (2021)

Rössler attractor. https://en.wikipedia.org/wiki/R\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\accent127o\egroup\spacefactor\accent@spacefactorssler_attractor

Runge-Kutta method. https://en.wikipedia.org/wiki/Runge-Kutta_methods