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 9, 2021

(Last compiled with LaTeX2e on October 29, 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 that is a *stiff* ODE, or IVP that is highly sensitive to the time mesh step size h (facilitates comparison of the numerical methods)
- The special case is defined simply as $y' = -15y, y(0) = 1$ and hence has the exact solution $y(t) = e^{-15t}$ for all $t \geq 0$
- That is, $f(t, y) := -15y$ with $(t_0, y_0) := (0, 1)$
- We will explore the solutions to this 1D linear (stiff) ODE system using the following basic methods:
  1. Explicit forward Euler algorithm
  2. Explicit Runge-Kutta (RK4) algorithm
  3. Explicit multistep method: Adams-Bashforth (ABF-3)
  4. GEKKO python library ODE solver method (dynamic simultaneous simulation mode + solver)

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
- $\text{LocalTruncationError} = O(h^2)$ (LTE at each step)
- f is Lipschitz $\text{GlobalTruncationError} = O(h)$ (depends on the function, or minimal Lipschitz constant, and the upper bound on the time interval exponentially)
- $y_{n+1} = y_n + f(t_n, y_n)(t_{n+1} - t_n) = y_n + h \cdot f(t_n, y_n)$
  $t_n = t_0 + nh$ for $n \in [0, N)$ and N sufficiently large to guarantee convergence to the solution

Explicit forward Euler method – Motivation

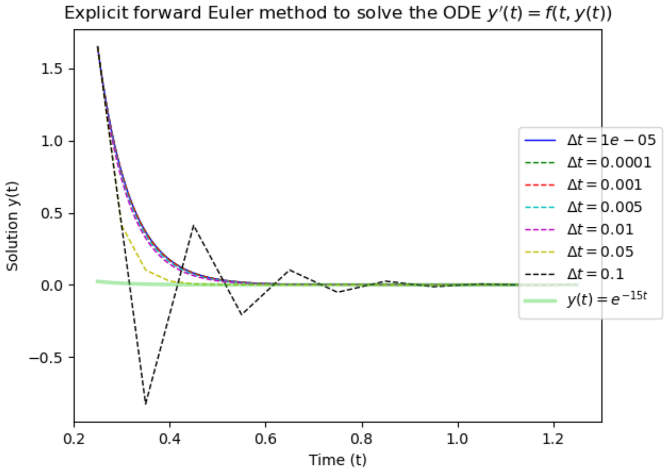
- **Key derivation:**
  $y'(t_0) \approx \frac{y(t_0+h)-y(t_0)}{h}$ $(*)$, where $y' = f(t, y) \implies$
  $y(t_0 + h) - y(t_0) \approx \int_{t_0}^{t_0+h} f(t, y(t))dt$
- Now we approximate the the RHS integral using a left-hand-endpoint Riemann sum ($n = 1$ rectangle) to obtain that
  $\int_{t_0}^{t_0+h} f(t, y(t))dt \approx h \cdot f(t_0, y(t_0))$
- 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"
```



Runge-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
- LocalTruncationError $= O(h^5)$ (LTE) and AccumulatedTruncationError $= O(h^4)$
- If f does not depend on y, the RK4 is the same as *Simpson's rule*

More general explicit Runge-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

  $k_1 = f(t_n, y_n)$

  $k_2 = f(t_n + c_2 h, y_n + h \cdot (a_{21} k_1))$

  $k_3 = f(t_n + c_3 h, y_n + h \cdot (a_{31} k_1 + a_{32} k_2))$

  $\cdots$

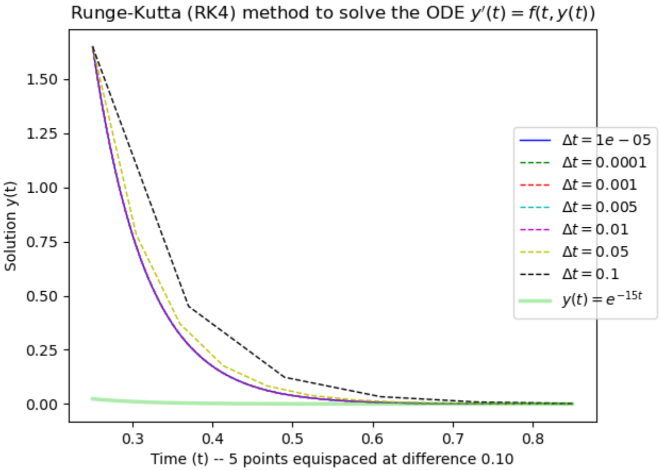
  $k_s = f(t_n + c_s h, h \cdot (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}))$.
- That is, the explicit RK method is completely determined by the parameters $(a_{ij})_{1 \leq j < i \leq s}$, $(b_i)_{1 \leq i \leq s}$ and $(c_j)_{2 \leq j \leq s}$
- We require *consistency* insomuch as $\sum_{i=1}^{s} b_i = 1$
- A popular convention is to require that $\sum_{j=1}^{i-1} a_{ij} = c_i$ for $i \in \{2, 3, \ldots, s\}$

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

```
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
```

Runge-Kutta (RK4) method (results)

```
(ipython) cd Examples/BasicNumericalODESolutionMethods
(ipython) run "RungeKuttaRK4Method.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:

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

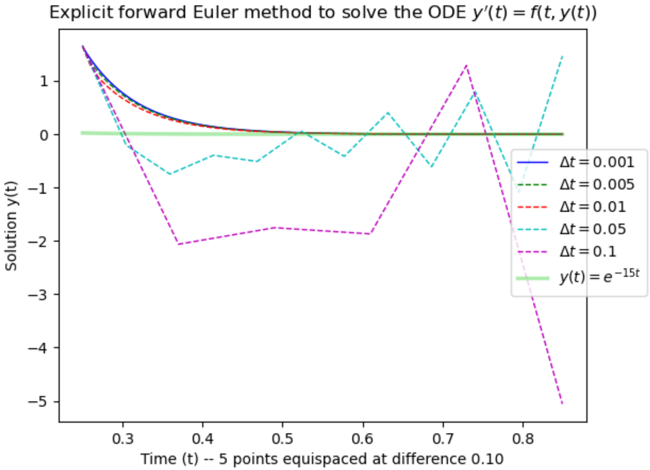
- **Note:** Single step ABF-1 is the forward Euler method

Step solver method – ABF3 (source code flavor)

```
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 "ABF3StepSolverMethod.py"
```

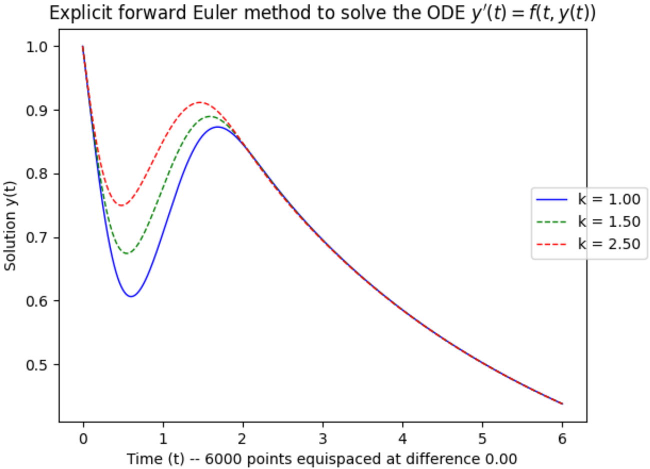


GEKKO library (source code flavor)

```
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

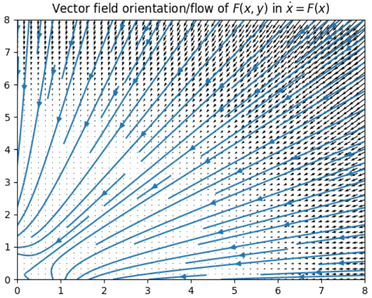
- **Problem 4 from the Fall 2021 Math 6307 midterm:** Consider the non-linear ODE $\dot{x} = F(x)$ on $\mathbb{R}^2$ defined such that $F(x, y) = \left(x(1 - x^2 - y^2) - y, y(1 - x^2 - y^2) + x\right)^T$
- The function $F(x, y)$ defines a vector field in 2D
- Solutions $(x(t), y(t))$ are witnessed along 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`)

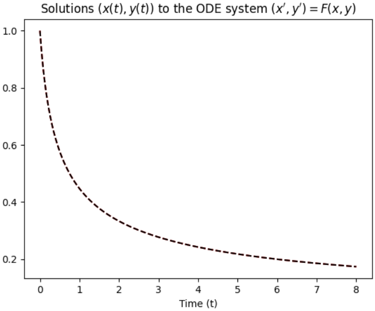
Generating vector field plots (source code flavor)

```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

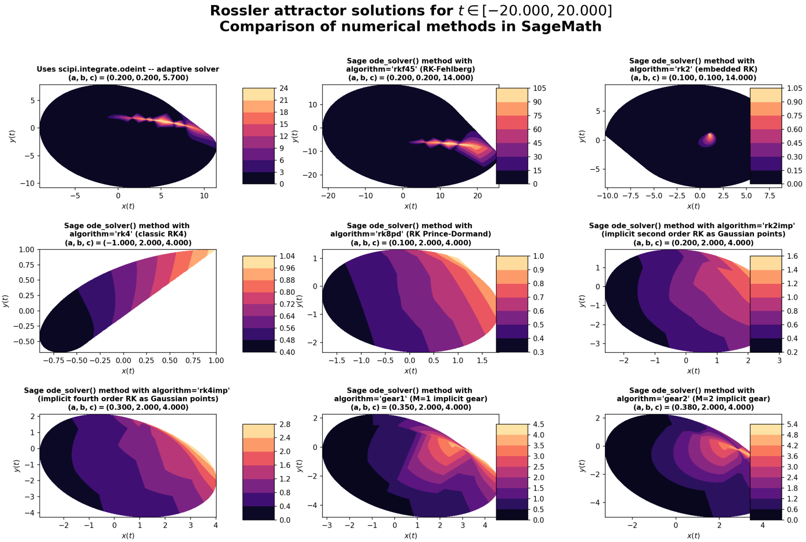
- Non-linear 3D systems of ODEs determined by parameters $(a, b, c) \in \mathbb{R}^3$
- Precise system: $(x', y', z') = (-y - z, x + ay, b + z(x - c))$
- Rössler famously studied the "classic" case with $(a, b, c) := (0.2, 0.2, 5.7)$ (important characteristic properties of other parameter special cases 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

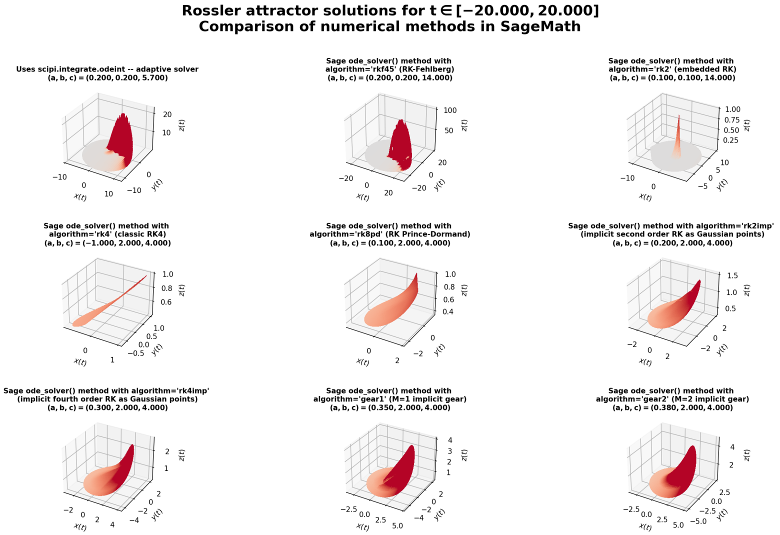
- We can use the `scipy.integration.odeint` function to numerically solve the projected system for *explicit* numerical values of the parameters (a, b, c)
- For the 3D plots, we transform the Z-component of the plot by taking the Euclidean norm of the projected point (see source code)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerPlot3DSystemComparison.py"
(sage) run "RosslerMiscPlots.py"
```

Exploring the classical parameter solution

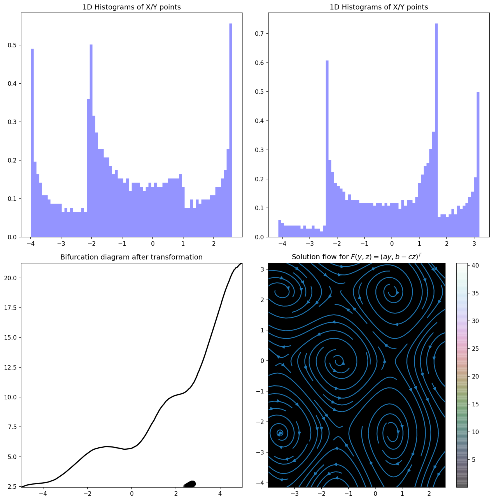


Exploring the classical parameter solution



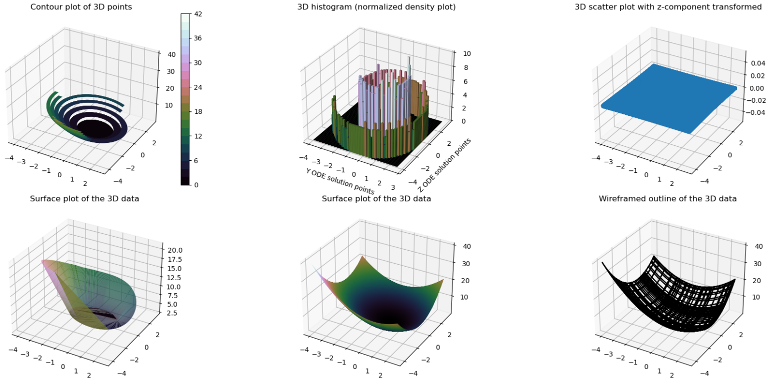
Exploring the classical parameter solution (1D plots)

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



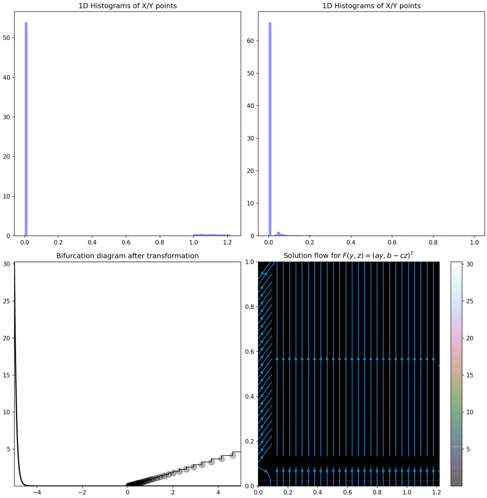
Exploring the classical parameter solution (3D plots)

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



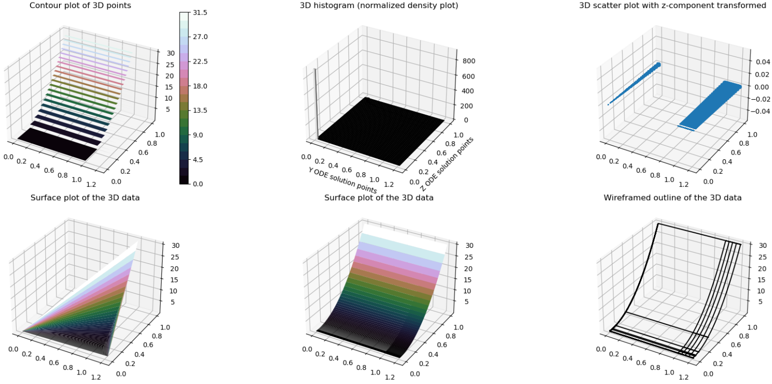
Exploring the classical parameter solution (1D plots)

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)

- We present Python3 source code for a numerical experiment that can be generalized and extended for use in other related applications
- In some senses, it forms another “toy” type experiment that happens to yield interesting results
- **Limitations and incompatibility:** The implementation was tricky in Python3 for a few reasons:
  1. 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)
  2. 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)
  3. Numerically evaluating the entire ODE solution for each possibility of (a, b, c) is an inefficient, time-consuming approach

Modified experiment V1: Definitions (cont'd)

- We consider parameterized numerical solutions to 2D systems of ODEs
- Suppose that we compute a uniformly spaced grid of N time points, $\{t_{N,0}, t_{N,1}, \ldots, t_{N,N-1}\}$ that partition the interval $[a, b]$
- The quasi-numerical-and-symbolic solutions to the ODE depend on a family of parameters $\mathcal{P}$, so that $(x(\mathcal{P}; t_{N,i}), y(\mathcal{P}; t_{N,i})) \approx (x_{N,i}(\mathcal{P}), y_{N,i}(\mathcal{P}))$
- We define an auxiliary function of interest in terms of the vector $\mathbb{R}^2[[\mathcal{P}]]$ as

$$\lambda_N(\mathcal{P}, v_0) := \frac{1}{N^2} \times \sum_{0 \leq i,j < N} \log \left| (x'_{\mathcal{P}}(t_{N,i}), y'_{\mathcal{P}}(t_{N,j})) \cdot v_0^T \right|$$

Modified experiment V1: Definitions (cont'd)

In the Rössler experiment here, we will define $\varpi_N(\mathcal{P}, v_0)$ to be the function in the last equation that results when we project downwards by setting one component of the system to zero, e.g.,

$$\lambda_N(a, v_0) := \frac{1}{N^2} \sum_{0 \leq i,j < N} \log \left| (-y_a(t_{N,j}), x_a(t_{N,i}) + a y_a(t_{N,j})), \cdot v_0^T \right|$$

(Projection to XY Case – Linear)

$$\lambda_N(b, c, v_0) := \frac{1}{N^2} \sum_{0 \leq i,j < N} \log \left| (-z_{b,c}(t_{N,j}), b + z_{b,c}(t_{N,j})(x_{b,c}(t_{N,i}) - c)), \cdot v_0^T \right|$$

(Projection to XZ Case – Non-Linear)

$$\lambda_N(a, v_0) := \frac{1}{N^2} \sum_{0 \leq i,j < N} \log \left| (-y_a(t_{N,i}) - z_a(t_{N,j}), a y_a(t_{N,i})), \cdot v_0^T \right|$$

(Projection to YZ Case – Linear)

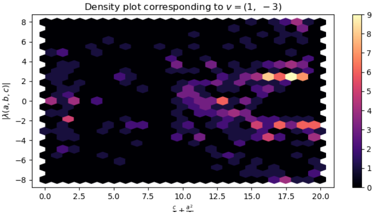
A modified experiment definition (Experiment V1)

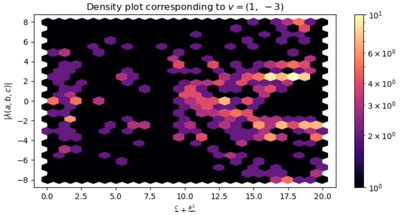
- The idea for numerical exploration here is examine the properties of this function for large $N \to \infty$ as they depend on variations in the symbolic (indeterminate) parameter set $\mathcal{P}$
- Since $\lambda_N(\mathcal{P}, v_0)$ depends symbolically on (a, b, c), we consider varying relations between these parameters that restricts the set of values which we use in the resulting plots
- Then for the satisfactory $(a, b, c) \in \mathbb{R}^3$, we compute the `mathplotlib.pyplot.hexbin` plot with $|\lambda_N(\mathcal{P}, v_0)|$ against the values of another varying user-defined function $\mathcal{T}_x(a, b, c)$ on the coordinate axes

Modified experiment V1 – Variant #1 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment1.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).

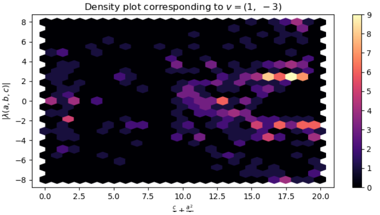


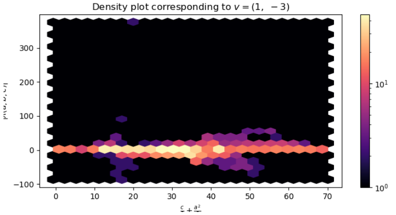


Modified experiment V1 – Variant #2 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment1.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).

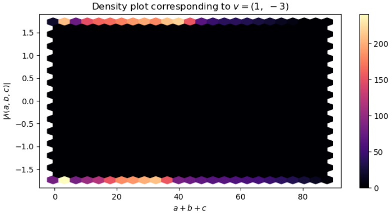


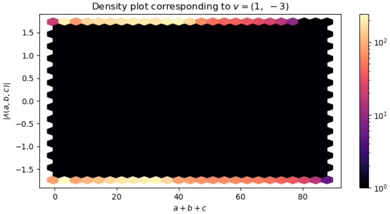


Modified experiment V1 – Variant #3 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment1.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).

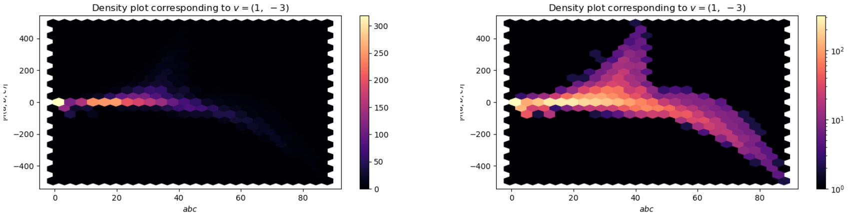




Modified experiment V1 – Variant #4 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment1.py"
```

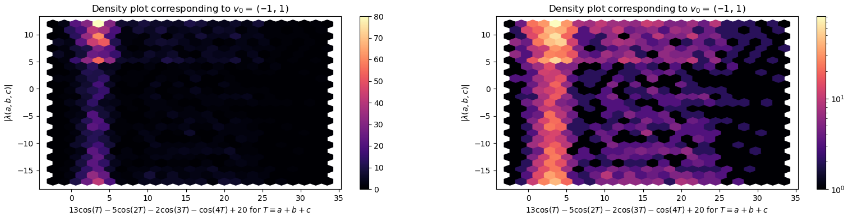
Variant 4: In the projected XY-plane with linear (right) and logarithmic (left) scaling on the X-axis defined by $\mathcal{T}_x(a, b, c) := abc$, subject to the restriction that $(a + b + c)^2 = 3$ (for real parameter values).



Modified experiment V1 – Variant #5 (results)

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment1.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 hexagonally-tiled density plots that resulted in seemingly random choices of $\mathcal{T}_x(a, b, c)$ 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 $\varpi_N(\mathcal{P}, v_0)$
- For this form of the modified experiment, we consider the projected linear system to XY components (by setting $Z = 0$)
- The resulting 2D ODE system only depends on a single indeterminate parameter (and the independent variable t)
- Moreover, standard ODE methods for solving an IVP of the form $\dot{x} = Ax, x(0) = (x_0, y_0)$ for a 2×2 matrix A such that $x(t) = e^{At}(x_0, y_0)^T$ shows that

$$\begin{bmatrix} x_a(t) \\ y_a(t) \end{bmatrix} = \begin{bmatrix} e^{\frac{at}{2}} \left(\cosh\left(\frac{\sqrt{a^2-4}t}{2}\right) - \frac{a \sinh\left(\frac{\sqrt{a^2-4}t}{2}\right)}{\sqrt{a^2-4}} \right) & \frac{e^{\frac{at}{2}} \left(\sqrt{a^2-4}-a\right) \sinh\left(\frac{\sqrt{a^2-4}t}{2}\right)}{\sqrt{a^2-4}} \\ \frac{e^{\frac{at}{2}} \left(\sqrt{a^2-4}+a\right) \sinh\left(\frac{\sqrt{a^2-4}t}{2}\right)}{\sqrt{a^2-4}} & e^{\frac{at}{2}} \left(\cosh\left(\frac{\sqrt{a^2-4}t}{2}\right) + \frac{a \sinh\left(\frac{\sqrt{a^2-4}t}{2}\right)}{\sqrt{a^2-4}} \right) \end{bmatrix} (1, 1)^T$$

Experiment V2: Problem setup (cont'd)

- To avoid complications in the numerical analysis, and for example, to remove a non-compactly supported resulting probability measure in the limit for $\nu_a(\vartheta)$ above, we have to pay attention to only evaluate the solution to the ODE system for t strictly to the left or right of the zero points of $x_a(t)$ and $y_a(t)$
- These zero points are given analytically in respective order as

$$T_{0,x}(a) = -\frac{\log\left(1 - \frac{\sqrt{a^2-4}}{a}\right)}{\sqrt{a^2-2}}, \; T_{0,y}(a) = -\frac{\log\left(1 + \frac{\sqrt{a^2-4}}{a}\right)}{\sqrt{a^2-2}}$$

- It is also desirable to keep the plots of $a \in (-2, 2)$ so that our resulting numerical solutions do not blow up, or oscillate, as $t \to \infty$
- As $\lim_{a\to\pm 2}(T_{0,x}(a), T_{0,y}(a)) = \left(\pm\frac{1}{2}, \mp\frac{1}{2}\right)$, $\lim_{a\to 0}(T_{0,x}(a), T_{0,y}(a)) = (\widehat{\infty}, \widehat{\infty})$ and $\lim_{a\to\pm\infty}(T_{0,x}(a), T_{0,y}(a)) = (0, 0)$ this means we should evaluate $t \in \mathcal{T}$ such that $\mathcal{T} \cap \left\{\frac{1}{2}\right\} = \varnothing \wedge \mathcal{T} \cap (-\infty, 0) = \varnothing$ and omit the parameter value of $a := 0$

Experiment V2: Problem setup (cont'd)

- So we consider the following function:

$$\varpi(a,u) := \frac{1}{N^2} \times \sum_{0\leq i,j<N} \log \left|(-y_a(t_{N,j}), x_a(t_{N,i}) + a y_a(t_{N,j})) \cdot (u,-1)\right|.$$

- Some arithmetic yields that

$$\varpi(a,u) = \underbrace{\frac{1}{N} \times \log \left[\prod_{0\leq i<N} |x_a(t_{N,i})|\right]}_{:=\varpi_{0,N}(a)} \frac{1}{N^2} \times \log \left[\prod_{0\leq i,j<N} \left|1 + \frac{(u+a)y_a(t_{N,j})}{x_a(t_{N,i})}\right|\right].$$

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

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

Experiment V2: Problem setup (cont'd)

- We also pose an ansatz that (for at least some a, u) we should get convergence in distribution insomuch as whenever $(i, j) \in [0, N)^2 \bigcap \mathbb{Z}^2$ is selected uniformly at random

$$\lim_{\Delta t \to 0} \mathbb{P}\left[t \leq \frac{y_{N,j}}{x_{N,i}} \leq t + \Delta t\right] \overset{\mathscr{D}}{\Longrightarrow} \nu_a(t), \text{ as } N \to \infty,$$

  is a probability measure with non-trivial properties
- Stated more precisely, suppose that for fixed a and $t \in \mathcal{T}$ and i. i. d. random variables $\Upsilon_1[\mathcal{T}], \Upsilon_2[\mathcal{T}] \sim \text{Uniform}(\mathcal{T})$, we define $\Theta_a[\mathcal{T}] \overset{\mathscr{D}}{=} \frac{y_a(\Upsilon_2[\mathcal{T}])}{x_a(\Upsilon_1[\mathcal{T}])}$. Then we expect that

$$\nu_a(\theta) = \frac{\partial}{\partial \theta} \mathbb{P}\left[\Theta_a[\mathcal{T}] \leq \theta\right],$$

  is a probability density function

Experiment V2: Problem setup (cont'd)

- When this happens, we have that

$$\begin{aligned}\varpi(a,u) &= \varpi_0(a) + \iint_{(s_1,s_2)\in\mathcal{T}\times\mathcal{T}} \log\left|1 + \frac{(u+a)y_a(s_2)}{x_a(s_1)}\right| ds_1 ds_2 \\ &= \varpi_0(a) + \mathbb{E}\log|1 + (u+a)\Theta_a[\mathcal{T}]| \\ &= \varpi_0(a) + \int_{-\infty}^{\infty} \log|1 + (u+a)\theta|\, \nu_a(\theta) d\theta \\ &= \varpi_0(a) - \log|1 + (u+a)\mathcal{S}_\ell(\mathcal{T},a)| + \int_{-\infty}^{\infty} \frac{(u+a)\mathbb{P}\left[\Theta_a[\mathcal{T}] \geq \theta\right]}{1 + (u+a)\theta} d\theta,\end{aligned}$$

  where the last equation holds when $\nu_a(\theta)$ has bounded support $\text{supp}(\nu_a) = [\mathcal{S}_\ell(\mathcal{T},a), \mathcal{S}_u(\mathcal{T},a)]$
- Moreover, if $\nu_a(\theta)$ has bounded support as above, then the rightmost term in the previous equation is within the bounded interval $\mathcal{I}(\mathcal{T},a)$ where

$$\mathcal{I}(\mathcal{T},a) := \left[\min\left\{\frac{(u+a)\mathbb{E}\Theta_a[\mathcal{T}]}{1+(u+a)\mathcal{S}_\ell(\mathcal{T},a)}, \frac{(u+a)\mathbb{E}\Theta_a[\mathcal{T}]}{1+(u+a)\mathcal{S}_u(\mathcal{T},a)}\right\}, \max\left\{\frac{(u+a)\mathbb{E}\Theta_a[\mathcal{T}]}{1+(u+a)\mathcal{S}_\ell(\mathcal{T},a)}, \frac{(u+a)\mathbb{E}\Theta_a[\mathcal{T}]}{1+(u+a)\mathcal{S}_u(\mathcal{T},a)}\right\}\right]$$

Experiment V2: Numerical methods towards the ansatz

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment2.py"
```

(TODO)

Experiment V2: Numerical methods towards the ansatz

```
(sage) cd Examples/RosslerAttractor
(sage) run "RosslerExperiment2.py"
```

(TODO)

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%C3%B6ssler_attractor

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