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 November 4, 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)

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)
- if f is Lipschitz, then $\text{GlobalTruncationError} = O(h)$ (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

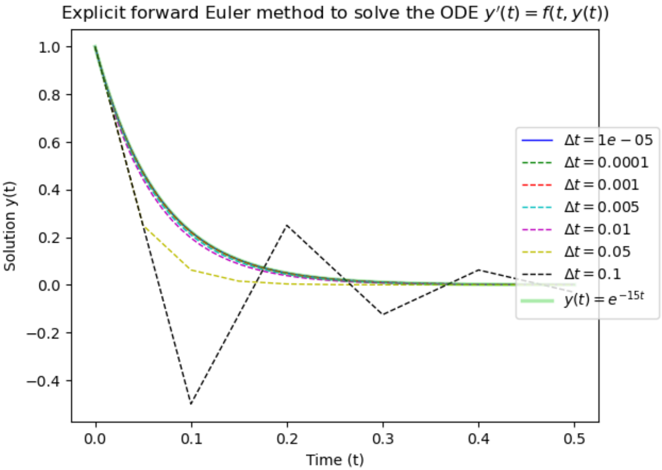
- **Key derivation:**
  $y'(t_0) \approx \frac{y(t_0+h)-y(t_0)}{h}$ $(*)$, where $y' = f(t, y) \implies$
  $y(t_0 + h) - y(t_0) \approx \int_{t_0}^{t_0+h} f(t, y(t))dt$
- Now we approximate the the RHS integral using a left-hand-endpoint Riemann sum ($n = 1$ rectangle) to obtain that
  $\int_{t_0}^{t_0+h}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

Explicit foward Euler method (source code flavor)

```python
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

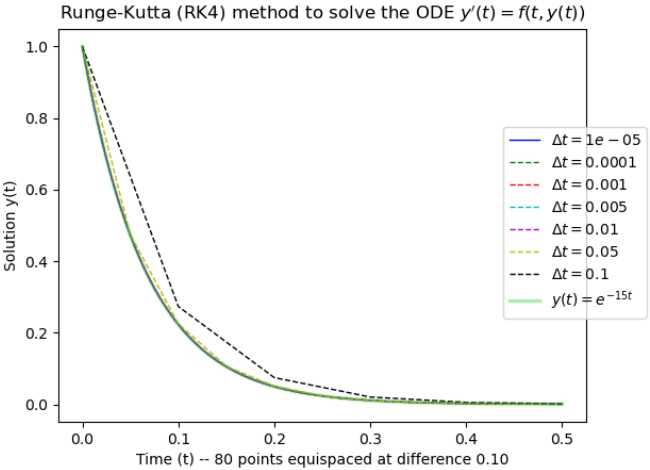
- Given an IVP: $y' = f(t, y), y(t_0) = y_0$
- $t_n = t_0 + h \cdot n$ (fixed / uniform step size)
- $k_1 = f(t_n, y_n)$, $k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{k_1 h}{2}\right)$, $k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{k_2 h}{2}\right)$, $k_4 = f\left(t_n + h, y_n + k_3 h\right)$
- $y_{n+1} = y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$ (recursion to evaluate)
- Taking the weighted average at four slopes leads to more weight given to slopes closer to the midpoint of each subinterval
- $\text{LocalTruncationError} = O(h^5)$ and $\text{GlobalTruncationError} = O(h^4)$
- 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
  $k_1 = f(t_n, y_n)$
  $k_2 = f(t_n + c_2 h, y_n + h \cdot (a_{21} k_1))$
  $k_3 = f(t_n + c_3 h, y_n + h \cdot (a_{31} k_1 + a_{32} k_2))$
  $\dots$
  $k_s = f(t_n + c_s h, h \cdot (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}))$.
- That is, the explicit RK method is completely determined by the parameters $(a_{ij})_{1 \le j < i \le s}$, $(b_i)_{1 \le i \le s}$ and $(c_j)_{2 \le j \le s}$
- 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 $i \in \{2, 3, \dots, s\}$

Runge-Kutta (RK4) method (results)

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



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

- The multistep ABF-s (s-step) case: $a_{s-1} = -1$; $a_{s-2} = \cdots = a_0 = 0$
- 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:

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

- With $f(t_{n+j}, y_{n+j}) \mapsto p(t_{n+j})$, the ABF-$s$ method coefficient multipliers are:

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

- Recursion in the ABF-3 method:

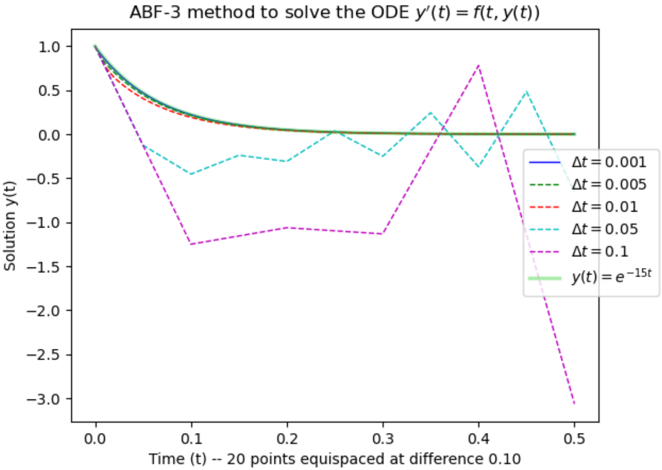
$$y_{n+3} = y_{n+2} + h\left(\frac{23}{12} f(t_{n+2}, y_{n+2}) - \frac{4}{3} f(t_{n+1}, y_{n+1}) + \frac{5}{12} f(t_n, y_n)\right)$$

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 "ABF3StepSolverMethod.py"
```



Key applications for numerical exploration

Chaotic attractors

Definitions and motivation

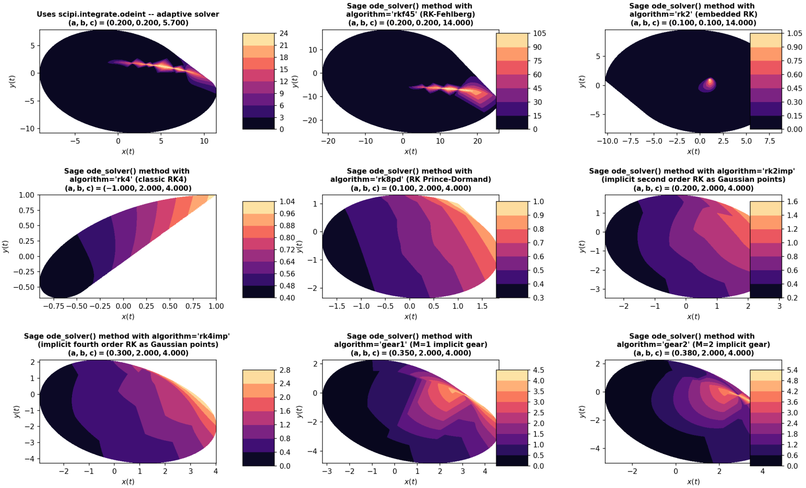
- When considering dynamical systems, an *attractor* is a set of states (orbits) towards which a system (of ODE solutions) tends to evolve
- System values within some small range of the *attractor* set stay close even if perturbed slightly (e.g., by slightly shifting an ODE initial condition)
- A *chaotic 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

Definition of the Rössler attractor problem

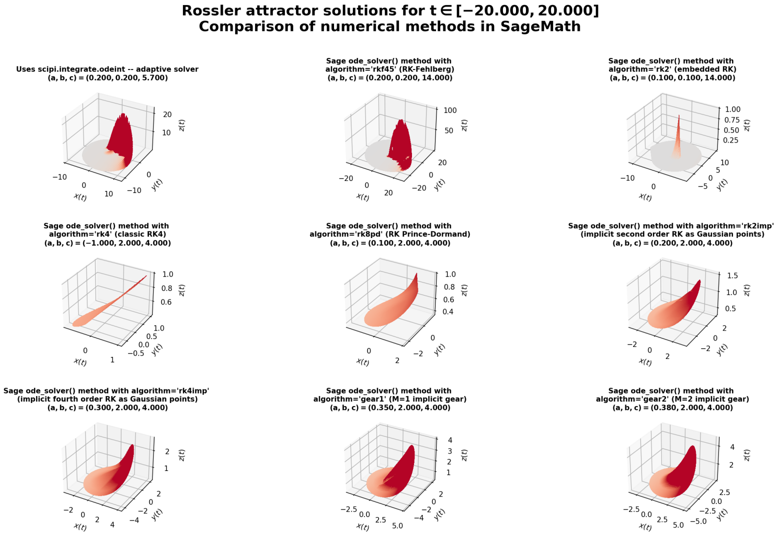
- We will focus on numerical exploration of the *Rössler attractor* system
- Non-linear 3D systems of ODEs determined by parameters $(a, b, c) \in \mathbb{R}^3$
- Precise system: $(x', y', z') = (-y - z, x + ay, b + z(x - c))$
- Rössler famously studied the "classic" case with $(a, b, c) := (0.2, 0.2, 5.7)$ (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 $z := 0$

Exploring the classical parameter solution

Rossler attractor solutions for $t \in [-20.000, 20.000]$
Comparison of numerical methods in SageMath

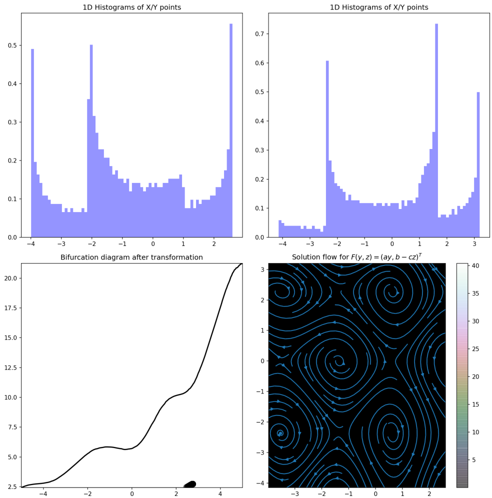


Exploring the classical parameter solution



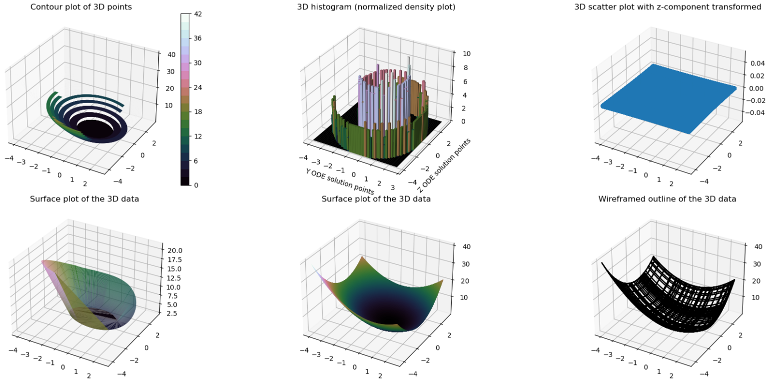
Exploring the classical parameter solution (1D plots)

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 $(a, b, c) = (0.2, 0.2, 5.7)$.



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