ODEs with Python

Brian Bingham

Version 0.1
November 2014

ODEs with Python

Attribution

Contents

Chapter 1

Ordinary Differential Equations

1.1 Differential equations

A **differential equation** (DE) is an equation that describes the derivatives of an unknown function. "Solving a DE" means finding a function whose derivatives satisfy the equation.

For example, when bacteria grow in particularly bacteria-friendly conditions, the rate of growth at any point in time is proportional to the current population. What we might like to know is the population as a function of time. Toward that end, let's define f to be a function that maps from time, t, to population y. We don't know what it is, but we can write a differential equation that describes it:

$$\frac{df}{dt} = af$$

where a is a constant that characterizes how quickly the population increases.

Notice that both sides of the equation are functions. To say that two functions are equal is to say that their values are equal at all times. In other words:

$$\forall t : \frac{df}{dt}(t) = af(t)$$

This is an **ordinary** differential equation (ODE) because all the derivatives involved are taken with respect to the same variable. If the equation related derivatives with respect to different variables (partial derivatives), it would be a **partial** differential equation.

This equation is **first order** because it involves only first derivatives. If it involved second derivatives, it would be second order, and so on.

This equation is **linear** because each term involves t, f or df/dt raised to the first power; if any of the terms involved products or powers of t, f and df/dt it would be nonlinear.

Linear, first order ODEs can be solved analytically; that is, we can express the solution as a function of t. This particular ODE has an infinite number of solutions, but they all have this form:

$$f(t) = be^{at}$$

For any value of b, this function satisfies the ODE. If you don't believe me, take its derivative and check.

If we know the population of bacteria at a particular point in time, we can use that additional information to determine which of the infinite solutions is the (unique) one that describes a particular population over time.

For example, if we know that $f(0) = 5$ billion cells, then we can write

$$f(0) = 5 = be^{a0}$$

and solve for b, which is 5. That determines what we wanted to know:

$$f(t) = 5e^{at}$$

The extra bit of information that determines b is called the **initial condition** (although it isn't always specified at $t = 0$).

Unfortunately, most interesting physical systems are described by nonlinear DEs, most of which can't be solved analytically. The alternative is to solve them numerically.

1.2 Euler's method

The simplest numerical method for ODEs is Euler's method. Here's a test to see if you are as smart as Euler. Let's say that you arrive at time t and measure the current population, y, and the rate of change, r. What do you think the population will be after some period of time Δt has elapsed?

If you said $y + r\Delta t$, congratulations! You just invented Euler's method (but you're still not as smart as Euler).

This estimate is based on the assumption that r is constant, but in general it's not, so we only expect the estimate to be good if r changes slowly and Δt is small.

But let's assume (for now) that the ODE we are interested in can be written so that

$$\frac{df}{dt}(t) = g(y, t)$$

where g is some function that maps (t, y) onto r; that is, given the time and current population, it computes the rate of change. Then we can advance from one point in time to the next using these equations:

$$T_{n+1} = T_n + \Delta t \tag{1.1}$$
$$F_{n+1} = F_n + g(y, t)\ \Delta t \tag{1.2}$$

Table 1.1: ODE Notation

Name	Meaning	Type
t	time	scalar variable
Δt	time step	scalar constant
y	population	scalar variable
r	rate of change	scalar variable
f	The unknown function specified, implicitly, by an ODE.	function $t \to y$
df/dt	The first time derivative of f	function $t \to r$
g	A "rate function," derived from the ODE, that computes rate of change for any t, y.	function $t, y \to r$
T	a sequence of times, t, where we estimate $f(t)$	sequence
F	a sequence of estimates for $f(t)$	sequence

Here $\{T_i\}$ is a sequence of times where we estimate the value of f, and $\{F_i\}$ is the sequence of estimates. For each index i, F_i is an estimate of $f(T_i)$. The interval Δt is called the **time step**.

Assuming that we start at $t = 0$ and we have an initial condition $f(0) = y_0$ (where y_0 denotes a particular, known value), we set $T_1 = 0$ and $F_1 = y_0$, and then use Equations 1.1 and 1.2 to compute values of T_i and F_i until T_i gets to the value of t we are interested in.

If the rate doesn't change too fast and the time step isn't too big, Euler's method is accurate enough for most purposes. One way to check is to run it once with time step Δt and then run it again with time step $\Delta t/2$. If the results are the same, they are probably accurate; otherwise, cut the time step again.

Euler's method is **first order**, which means that each time you cut the time step in half, you expect the estimation error to drop by half. With a second-order method, you expect the error to drop by a factor of 4; third-order drops by 8, etc. The price of higher order methods is that they have to evaluate g more times per time step.

1.3 Another note on notation

There's a lot of math notation in this chapter so I want to pause to review what we have so far. Here are the variables, their meanings, and their types:

So f is a function that computes the population as a function of time, $f(t)$ is the function evaluated at a particular time, and if we assign $f(t)$ to a variable, we usually call that variable y.

Similarly, g is a "rate function" that computes the rate of change as a function of time and population. If we assign $g(y, t)$ to a variable, we call it r.

df/dt is the first derivative of f, and it maps from t to a rate. If we assign $df/dt(t)$ to a variable, we call it r.

It is easy to get df/dt confused with g, but notice that they are not even the same type. g is more general: it can compute the rate of change for any (hypothetical) population at any time; df/dt is more specific: it is the actual rate of change at time t, given that the population is $f(t)$.

1.4 Euler's Method Implementation

1.4.1 Example Problem: Rat Population

As an example, suppose that the rate of population growth for rats depends on the current population and the availability of food, which varies over the course of the year. The governing equation might be something like

$$\frac{df}{dt}(t) = af(t)\left[1 + \sin(\omega t)\right]$$

where t is time in days and $f(t)$ is the population at time t.

a and ω are **parameters**. A parameter is a value that quantifies a physical aspect of the scenario being modeled. Parameters are often constants, but in some models they vary in time.

In this example, a characterizes the reproductive rate, and ω is the frequency of a periodic function that describes the effect of varying food supply on reproduction.

This equation specifies a relationship between a function and its derivative. In order to estimate values of f numerically, we have to transform it into a rate function.

The first step is to introduce a variable, y, as another name for $f(t)$

$$\frac{df}{dt}(t) = ay\left[1 + \sin(\omega t)\right]$$

This equation means that if we are given t and y, we can compute $df/dt(t)$, which is the rate of change of f. The next step is to express that computation as a function called g:

$$g(y, t) = ay\left[1 + \sin(\omega t)\right]$$

1.4.2 Pseudo-code Implementation

```
• Set initial conditions: $y(t = 0) = y_o$
• Specify the time step ($\Delta t$) and the stop time ($t_f$)
• Iterate while $t < t_f$
    – Evaluate the rate function at current time: $r = g(y, t)$
    – Estimate the future: $y(t + \Delta t) = y(t) + r * (\Delta t)$
    – Increment the time: $t = t + (\Delta t)$
```

1.4.3 Python Implementation

Here is an example of how we might implement a Euler solver for the example problem above.

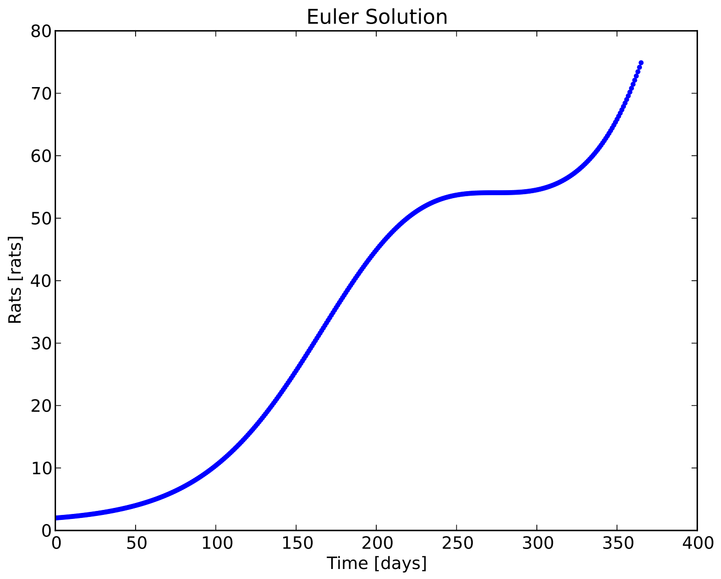
```
import math
import matplotlib.pyplot as plt
# Initial conditions
y0 = 2.0
# Time step and final time
dt = 1.0 # days
tf = 364.0 # days
# Empty lists to hold the time and solution
tt = []
yy = []
# Append initial values
tt.append(0.0)
yy.append(y0)
# Parameters of our equation
a = 0.01
omega = 2.0*math.pi/365.0
# Iteration
while tt[-1] <= tf:
    r = a * yy[-1] * (1+math.sin(omega*tt[-1]))
    yy.append(yy[-1]+r*dt)
    tt.append(tt[-1]+dt)

plt.figure(1)
plt.plot(tt,yy,'.')
plt.xlabel('Time [days]')
plt.ylabel('Rats [rats]')
plt.title('Euler Solution')
plt.show()
```

This code generates the following figure:

1.5 `scipy.integrate.odeint` and `scipy.integrate.ode`

A limitation of Euler's method is that the time step is constant from one iteration to the next. But some parts of the solution are harder to estimate than others; if the time step is small enough to get the hard parts right, it is doing more work than necessary on the easy parts. The ideal solution is to adjust the time step as you go along. Methods that do that are called **adaptive**. The nice people at SciPy (`http://www.scipy.org/`) have implemented a few of these so that we can use more sophisticated numerical integration when necessary[1].

[1]A more accurate depiction might be that SciPy provides wrappers for standard libraries that implement these algorithms. This implementations are often written in FORTRAN and used across a large variety of applications.



1.5.1 `odeint`: lsoda integration

The `odeint` function in the `scipy.integrate` module has a straightforward interface for accomplishing the integration. In order to use `odeint`, you have to write a function that evaluates g as a function of y and t.

Writing the function this way is useful because we can use it with Euler's method or `odeint` to estimate values of f. All we have to do is write a function that evaluates g. Here's what that looks like using the values $a = 0.01$ and $\omega = 2\pi/365$ (one cycle per year):

```
import numpy as np
import matplotlib.pyplot as plt
from ode_rats import rats
from scipy.integrate import odeint
# Initial conditions
y0 = 2.0
# Time gride for integration
tt = np.linspace(0, 364, 365)
# Solve the ODE
yy = odeint(rats, y0, tt)

plt.figure(1)
plt.plot(tt,yy,'.')
plt.xlabel('Time [days]')
plt.ylabel('Rats [rats]')
plt.title('ODE Solution')
plt.show()
```

You can test this function from the iPython interpreter by calling it with different values of `t` and `y`; the result is the rate of change (in units of rats per day):

```
In [1]: from ode_rats import *

In [2]: rats(0,2)
Out[2]: 0.02
```

So if there are two rats on January 1, we expect them to reproduce at a rate that would produce 2 more rats per hundred days. But if we come back in April, the rate has almost doubled:

```
In [3]: rats(120,2)
Out[3]: 0.03760024407947071
```

Since the rate is constantly changing, it is not easy to predict the future rat population, but that is exactly what `odeint` does. Here's how you would use it:

```
import numpy as np
import matplotlib.pyplot as plt
from ode_rats import rats
from scipy.integrate import odeint
# Initial conditions
y0 = 2.0
# Time gride for integration
tt = np.linspace(0, 364, 365)
# Solve the ODE
yy = odeint(rats, y0, tt)

plt.figure(1)
plt.plot(tt,yy,'.')
plt.xlabel('Time [days]')
plt.ylabel('Rats [rats]')
plt.title('ODE Solution')
plt.show()
```

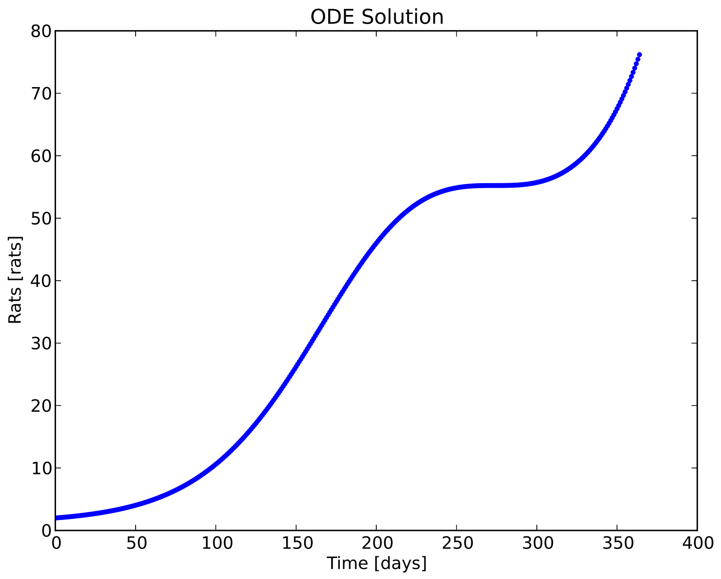
The first argument is a reference to the function that computes g. The second argument is the interval we are interested in, one year. The third argument is the initial population, $f(0) = 2$. The results look like this:

The x-axis shows time from 0 to 365 days; the y-axis shows the rat population, which starts at 2 and grows to almost 80. The rate of growth is slow in the winter and summer, and faster in the spring and fall, but it also accelerates as the population grows.

How much does the final population change if you double the initial population? How much does it change if you double the interval to two years? How much does it change if you double the value of a?

1.5.2 `ode`: Using other algorithms

In my limited experience the use of Euler integration and the lsoda algorithm accessed through `odeint` are sufficient for 99.9% of the problems you are likely to run into. Of course there are exceptions to this - problems that demand tailored algorithms to deal with issues that cause numerical instability for fixed time step algorithms (Euler) and the adaptive time stepping of lsoda.



If you find yourself in a situation where the 'standard' solvers just aren't good enough SciPy has the `ode` function that gives you access to a few more solvers[2]. The interface for `ode` is a little more cumbersome that `odeint`. Here is an example where we use `ode` to solve our rate population problem:

```python
import numpy as np
import matplotlib.pyplot as plt
from ode_rats import rats
from scipy.integrate import ode
# Initial conditions
y0 = 2.0
# Time gride for integration
t0 = 0.0
tF = 364
num_steps = 365
tt = np.linspace(t0, tF, num_steps)
# Setup list to hold solution
yy = []
yy.append(y0)

# Setup the solver
solver = ode(rats)
solver.set_integrator('dopri5') # Use Runge-Kutta 4/5 algorithm
solver.set_initial_value(y0,t0) # Initial conditions: time and value

# Integrate
for t in tt[1:]:
    yy.append(solver.integrate(t))
```

[2]See `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.ode.html` for descriptions of the available algorithms.

```
    if not solver.successful():
        print "WARNING: Itegration not successful!"


plt.figure(1)
plt.plot(tt,yy,'.')
plt.xlabel('Time [days]')
plt.ylabel('Rats [rats]')
plt.title('ODE Solution')
plt.show()
```

1.6 Analytic or numerical?

When you solve an ODE analytically, the result is a function, f, that allows you to compute the population, $f(t)$, for any value of t. When you solve an ODE numerically, you get two 1-D arrays. You can think of these arrays as a discrete approximation of the continuous function f: "discrete" because it is only defined for certain values of t, and "approximate" because each value F_i is only an estimate of the true value $f(t)$.

So those are the limitations of numerical solutions. The primary advantage is that you can compute numerical solutions to ODEs that don't have analytic solutions, which is the vast majority of nonlinear ODEs.

If you are curious to know more about how `odeint` works, you can modify `rats` function to display the points, (t, y), where `odeint` evaluates g. Here is a simple version:

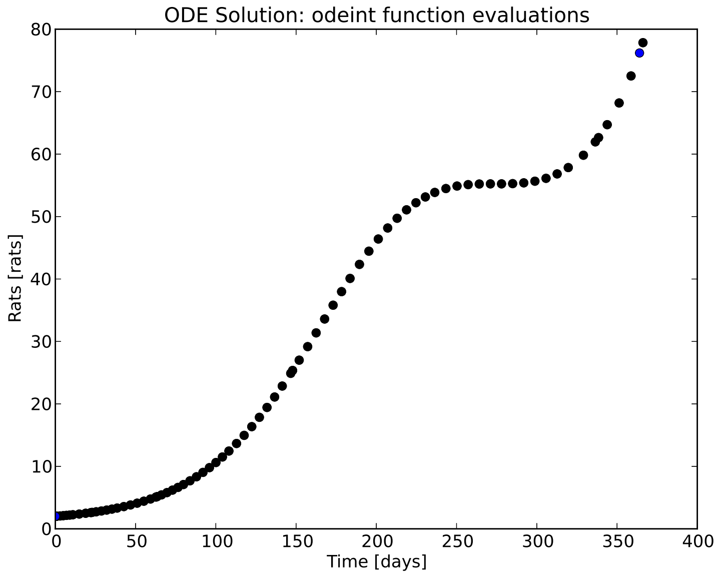
```
import math
import matplotlib.pyplot as plt
def rats(y,t):
    plt.plot(t,y,'ko')
    a = 0.01
    omega = 2.0*math.pi/365.0
    return a * y * (1+math.sin(omega*t))
```

Then we can modify the program that implements `odeint` to use this new function like this:

```
import numpy as np
import matplotlib.pyplot as plt
from ode_rats_plot import rats
from scipy.integrate import odeint
# Initial conditions
y0 = 2.0
# Time gride for integration
tt = np.linspace(0, 364, 2)
# Open a figure window
plt.figure(1)
# Solve the ODE
yy = odeint(rats, y0, tt)

plt.plot(tt,yy,'bo')
plt.xlabel('Time [days]')
plt.ylabel('Rats [rats]')
plt.title('ODE Solution: odeint function evaluations')
plt.show()
```

You should notice that we are asking `odeint` to operate over just two points ($t = 0$ and $t = 364$). The adaptation in the algorithm determines the spacing of the function evaluations.

Each time `rats` is called it plots one data point generating a figure like this:



The circles show the points where `odeint` called the `rats` function.

1.7 Stiffness

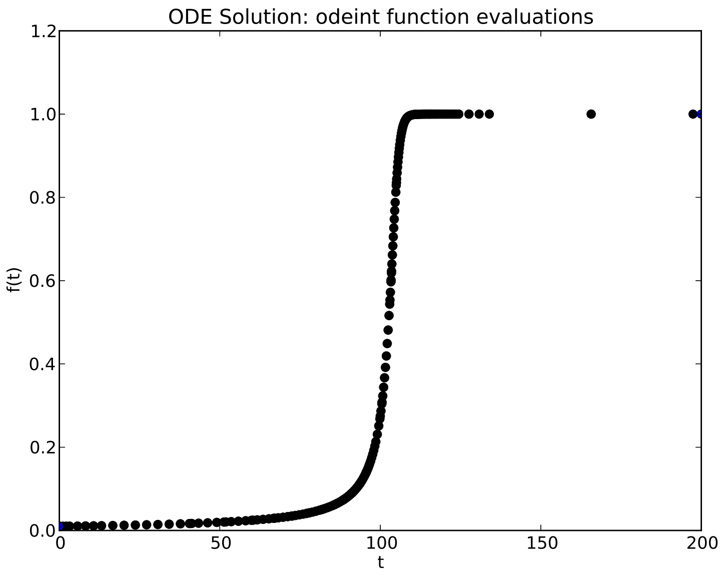
There is yet another problem you might encounter, but if it makes you feel better, it might not be your fault: the problem you are trying to solve might be **stiff**[3].

I won't give a technical explanation of stiffness here, except to say that for some problems (over some intervals with some initial conditions) the time step needed to control the error is very small, which means that the computation takes a long time. Here's one example:

$$\frac{df}{dt} = f^2 - f^3$$

If you solve this ODE with the initial condition $f(0) = \delta$ over the time interval from $t = 0$ to $t = 2/\delta$. As an example, if you consider a case where $\delta = 0.01$, you should see something like this:

[3]The following discussion is based partly on an article from Mathworks available at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html`



After the transition from 0 to 1, the time step is very small and the computation goes slowly. For smaller values of δ, the situation is even worse. Here is a figure where we have zoomed into the transition area to see how often the rate function is being evaluated:

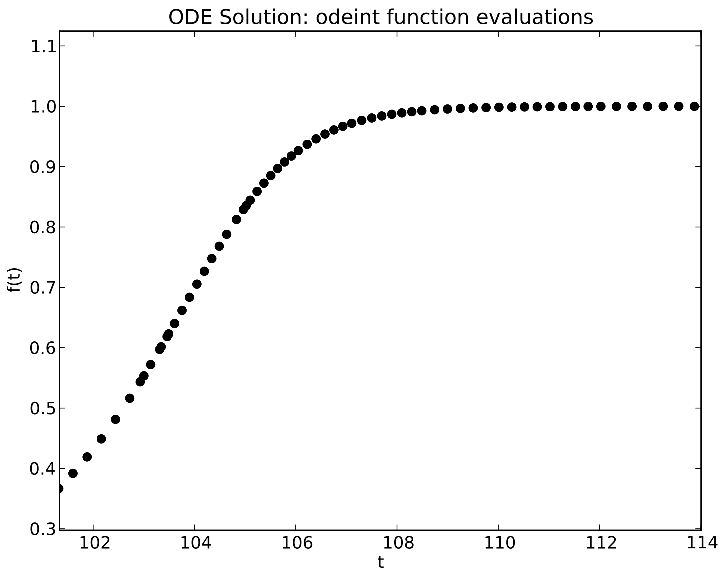
1.8 Conclusion

We've covered three ways to numerically solve ODEs:

1. Implement your own Euler integration
2. Use SciPy's `scipy.integrate.odeint` function to apply the lsoda algorithm.
3. Use SciPy's `scipy.integrate.ode` function to apply another algorithm, such as the Runge-Kutta 4/5 algorithm.

Here is a biased, subjective recommendation on how to plan your attack when faced with a problem that requires a numerical solution to an ODE.

- Are you fairly confident the problem is not 'stiff'? If so, then write you own Euler solution. It is easier to debug and control exactly what is going on.
  - Make sure and verify that you are getting a consistent solution by reducing the time step until there is no change in the solution. If your solution depends on your choice of time step, you have a problem.
- If you think there is a change your problem is 'stiff', but you don't have a strong preference for what algorithm to use then use `odeint`. The interface is a bit easier to deal with than `ode`.
  - Make sure to check that your integration was successful. The `odeint` function will output a dictionary (`infodict`) that you can examine to make sure your answer is reliable.



- If you know your problem requires a particular algorithm, then use the `ode` function.

1.9 Glossary

differential equation (DE): An equation that relates the derivatives of an unknown function.

ordinary DE: A DE in which all derivatives are taken with respect to the same variable.

partial DE: A DE that includes derivatives with respect to more than one variable

first order (ODE): A DE that includes only first derivatives.

linear: A DE that includes no products or powers of the function and its derivatives.

time step: The interval in time between successive estimates in the numerical solution of a DE.

first order (numerical method): A method whose error is expected to halve when the time step is halved.

adaptive: A method that adjusts the time step to control error.

stiffness: A characteristic of some ODEs that makes some ODE solvers run slowly (or generate bad estimates). Some ODE solvers, like `ode23s`, are designed to work on stiff problems.

parameter: A value that appears in a model to quantify some physical aspect of the scenario being modeled.

1.10 Exercises

Exercise 1.1 *Implement a Euler integration solution to Section 1.7. Use the same parameter from the example ($\delta = 0.01$). You should plot the solution in order to compare it to the example.*

Start with a time step of $\Delta t = 10$. Decrease the time step by a factor of 2 until the results stop changing, i.e., find the minimum time step that provides a solution that is consistent with the example.

Exercise 1.2 *Suppose that you are given an 8 ounce cup of coffee at 90 °C and a 1 ounce container of cream at room temperature, which is 20 °C. You have learned from bitter experience that the hottest coffee you can drink comfortably is 60 °C.*

Assuming that you take cream in your coffee, and that you would like to start drinking as soon as possible, are you better off adding the cream immediately or waiting? And if you should wait, then how long?

To answer this question, you have to model the cooling process of a hot liquid in air. Hot coffee transfers heat to the environment by conduction, radiation, and evaporative cooling. Quantifying these effects individually would be challenging and unnecessary to answer the question as posed.

As a simplification, we can use Newton's Law of Cooling[4]*:*

$$\frac{df}{dt} = -r(f - e)$$

where f is the temperature of the coffee as a function of time and df/dt is its time derivative; e is the temperature of the environment, which is a constant in this case, and r is a parameter (also constant) that characterizes the rate of heat transfer.

It would be easy to estimate r for a given coffee cup by making a few measurements over time. Let's assume that that has been done and r has been found to be 0.001 in units of inverse seconds, $1/s$.

- *Using mathematical notation, write the rate function, g, as a function of y, where y is the temperature of the coffee at a particular point in time.*
- *Create an program named* `coffee.py`
- *Add a function called* `rate_func` *to* `coffee.py` *that takes* `y` *and* `t` *as inputs and computes $g(y, t)$. Notice that in this case g does not actually depend on t; nevertheless, your function has to take t as the first input argument in order to work with* `odeint`*.*
  Test your function by adding a line like `rate_func(90,0)` *to* `coffee.py`*, then call* `coffee.py` *from the command-line, e.g.,* `python coffee.py`*.*
- *Once you get* `rate_func(90,0)` *working, modify* `coffee` *to use* `odeint` *to compute the temperature of the coffee (ignoring the cream) for 60 minutes. Confirm that the coffee cools quickly at first, then more slowly, and reaches room temperature (approximately) after about an hour.*
- *Add functionality to your program to generate a figure, using matplotlib, that graphs the temperature of the coffee as a function of time. Make sure to add labels to the axes (with units) and a title.*
- *When you execute the program from the command line (*`python coffee.py`*) the program*

[4] `http://en.wikipedia.org/wiki/Heat_conduction`

should solve the ODE and show a figure window to visualize the solution.

Exercise 1.3 *Continuing with the above exercise...*

- *Write a function called* `mix_func` *that computes the final temperature of a mixture of two liquids. It should take the volumes and temperatures of the liquids as parameters.*
  In general, the final temperature of a mixture depends on the specific heat of the two substances[5]*. But if we make the simplifying assumption that coffee and cream have the same density and specific heat, then the final temperature is* $(v_1 y_1 + v_2 y_2)/(v_1 + v_2)$*, where* v_1 *and* v_2 *are the volumes of the liquids, and* y_1 *and* y_2 *are their temperatures.*
  Add code to `coffee` *to test* `mix_func`*.*
- *Use* `mix_func` *and* `odeint` *to compute the time until the coffee is drinkable if you add the cream immediately.*

Exercise 1.4 *Continuing with the above exercise...*

- *Modify* `coffee` *so it takes an input variable* t *that determines how many seconds the coffee is allowed to cool before adding the cream, and returns the temperature of the coffee after mixing.*
- *Use* `fzero` *to find the time* t *that causes the temperature of the coffee after mixing to be 60* $^\circ C$*.*
- *What do these results tell you about the answer to the original question? Is the answer what you expected? What simplifying assumptions does this answer depend on? Which of them do you think has the biggest effect? Do you think it is big enough to affect the outcome? Overall, how confident are you that this model can give a definitive answer to this question? What might you do to improve it?*

[5] `http://en.wikipedia.org/wiki/Heat_capacity`