Differential Equation Parameter Estimations

By Erico Tjoa
Contents
1. Objective: Parameter Estimation of ODE ................................................................. 3

2. Using Gradient Descent with Cost Function ............................................................ 3
   Theory...................................................................................................................... 3
   Scripts Overview .................................................................................................... 5
   param_estimate.py ................................................................................................. 5
   fitting.py ............................................................................................................... 6
   fitting_post_processing.py .................................................................................... 7

Usage Manual........................................................................................................... 8
   Virtual environment ................................................................................................ 8
   Methods .................................................................................................................. 8

3. Using Scipy.optimize.least_squares ........................................................................ 13
   Scripts Overview .................................................................................................... 13
   scipyode_header.py ............................................................................................... 13
   cubicODE.py ......................................................................................................... 13
   cubicODE_post_processing.py ............................................................................... 14

Methods.................................................................................................................... 14

Bibliography............................................................................................................... 17
1. Objective: Parameter Estimation of ODE

Given a set of data points \{(n, t)\}, estimate the parameters of

\[
\frac{dn}{dt} = G - k_1n - k_2n^2 - k_3n^3
\]  

(1)

Throughout, we also denote \(\frac{dn}{dt} \equiv F(n, p)\), where \(p = (G, k_1, k_2, k_3)\). We are given many data points \{(n_i, t_i)\} which can be used for the estimation.

2. Using Gradient Descent with Cost Function

This gradient descent is in a sense home-made and does not have too much fancy precautionary measures. In other sections, for example, optimization is performed using well-developed packages such as Scipy etc. Do consider using them.

Theory

Before going too far with sophisticated or overly complex methods, let us try the good old gradient descent. In fact, according to this reference [1], such method, typically called nonlinear least square (NLS), is more than sufficient “at least for simple smooth systems” (page 9).

Gradient Descent

Given a function \(y = f(\vec{x})\) where we want to minimize \(y\). Illustrating this with \(\vec{x} = (x_1, x_2)^T\), perform first order Taylor expansion to get \(\Delta y \approx \frac{dy}{dx_1} dx_1 + \frac{dy}{dx_2} dx_2 = \nabla y . d\vec{x}\). Gradient descent prescribes that the small change \(d\vec{x} = -\eta \nabla y\) where \(\eta > 0\) so that \(\Delta y \approx -\eta |\nabla y|^2 \leq 0\), which means \(y\) will always decrease in first order approximation, which is suitable for finding local minimum.

Cost function

I see some lecture notes use what is called the objective function. Recently, while doing deep neural network, I encountered such a function and it is called the cost function. We will stick to this name. The common choice of cost function is the mean squared values MSE.

\[
MSE = \frac{1}{2n} \Sigma_{i=1}^{n} |Y_i - Y0|_2^2
\]  

(2)

where \(Y, Y0\) are defined in the next section and \(n\) is the available number of data points.

Estimating parameters
Given the variables $n$ varying in $t$, data points $(n_i, t_i)$ and the differential equation governing its dynamic $\frac{dn}{dt} = F(n, p)$. We want to estimate the parameters $p = (G, k_1, k_2, k_3)$. Gradient descent will require

$$p \rightarrow p - \eta \nabla_p MSE$$

(3)

$$MSE = \frac{1}{2N} \sum_{i=1}^{N} (n_i - n0_i)^2$$

(4)

$$\frac{\partial MSE}{\partial k} = \frac{1}{N} \sum_{i=1}^{N} (n_i - n0_i) \frac{\partial n_i}{\partial k}$$

(5)

and likewise, the same equations for continuous variables:

$$MSE = \frac{1}{2} \int dt \left( n(t, k) - n0(t) \right)^2$$

(6)

$$\frac{\partial MSE}{\partial k} = \int dt \left[ n(t, p) - n0(t) \right] \frac{\partial n(t, p)}{\partial k}$$

(7)

where $i$ indexes the data points, $N$ the number of data points, $n0_i$ the true/observed values corresponding to time $t_i$. Note that $n = n(t, p)$. Ignoring the rigour in mathematics, we use

$$\frac{d}{dt} \int_{t_0}^{t} dt \frac{\partial n}{\partial k} = \int dt \frac{\partial}{\partial k} \left( \frac{\partial n}{\partial t} \right) = \int dt \ g_k(t, p)$$

where $g_k(t, p) = 1, -n, -n^2, -n^3$ respectively for $k = G, k_1, k_2, k_3$. Reverting back to discrete form, assuming our data points are sufficiently continuous, then

$$\partial_k MSE = \frac{1}{N} \sum_{i=1}^{N} \left[ (n_i - n0_i) \times \frac{1}{N_i} \sum_{j=1}^{N_i} g_k(t, p) \right]$$

(8)

Let us explore this further.

$$\partial_G MSE = \frac{1}{N} \sum_{i=1}^{N} (n_i - n0_i)$$

(9)

$$\partial_{k_1} MSE = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{1}{N_i} (n_i - n0_i)(-n_j)$$

(10)

and likewise for the remaining 2 parameters. The trouble perhaps will come from obtaining $n_j$ for $j = 1, 2, ..., i$. In this case, make sure that our ODE solver used to obtain $n_i$ will output $n_j$ along the way: this turns out to be not a problem.
Scripts Overview

The codes and sample data can also be found here.

_param_estimate.py_

**load_data()**

We load data from a csv file. The csv file has filename specified in the variable *filename*. In this csv file, the first column corresponds to the horizontal axis, which we label time, t. The second column is the vertical axis, which we label n.

The data loaded can be customized in this function. For example, in figure 1 we are only interested in the data starting from the peak value onwards.

![Figure 1](image.png)

*Figure 1.* Blue points refer to raw data, while red line refers to the relevant data to be used for this project.

**read_csv()**

Load data from a csv file.

**update_p()**

This is the main function performing gradient descent.

**dkdMSE()**

This function computes equation (8). *diff_term* refers to *g_k*(t, p).

This function has to be customized when the differential equation changes.

**nabla_MSE()**

This function computes ∇_p|MSE as seen in equation (3).

This function has to be customized when the differential equation changes.

**RK45_wrapper()**
This function is built on top of RK45 from scipy. It will solve for ODE \( \frac{dy}{dt} = f(t, y) \) at the specified set of points \( \{t\} \), yielding \( \{(n, t)\} \).

**fitting.py**

**Load Data.** Set `plot_data=True` to plot the existing data.

**Prepare Model.** The function `def F(y, p)` corresponds to the RHS of the differential equation. The `list_of_derivatives` is the list of \( g_k(t, p) \) in equation (8) for each parameter \( k \).

**Prepare Initial Guess.** In order to perform data fitting to estimate parameters, a good initial guess of the parameters is necessary. `collection_of_p_init` is a list of initial guesses to try and error before proceeding with optimization. The curves will be plotted so that we can observe what are the range of initial parameters that can be used for optimization in the next section “Start Optimizing”.

**Start Optimizing.** Once we know which initial parameters are suitable, set `p_init_max` and `p_init_min`. For example, if the parameters are good in the following range \( G \in [0,0.1), k1, k2, k3 \in [-1,1] \), then set `p_init_min=[0,-1,-1,-1]` and `p_init_max=[0.1,1,1,1]`, each corresponding to the minimum and maximum of \([G,k1,k2,k3]\). When optimizing, uniform random values are drawn from these

<table>
<thead>
<tr>
<th><code>no_of_tries</code></th>
<th>(Integer). Run the simulation over <code>no_of_tries</code> different random initial guesses.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>save_name</code></td>
<td>(String). The optimization data will be saved with <code>save_name</code> as as the prefix. For example, if we set <code>save_name</code> to “fitting_data” and <code>no_of_tries=3</code>, then the data will be saved as:</td>
</tr>
<tr>
<td></td>
<td>1. fitting_data_1.par,</td>
</tr>
<tr>
<td></td>
<td>2. fitting_data_2.par and</td>
</tr>
<tr>
<td></td>
<td>3. fitting_data_3.par</td>
</tr>
<tr>
<td><code>es.learning_rate</code></td>
<td>(Float). In equation (3), this refers to ( \eta ).</td>
</tr>
<tr>
<td><code>no_of_iterations</code></td>
<td>(Integer). The number of times <code>update_p()</code> is called or the gradient descent is performed. Default =100</td>
</tr>
<tr>
<td><code>save_interval</code></td>
<td>(Integer). Of the specified number of iterations, save every <code>save_interval</code>-th iteration. Default = 10. <strong>IMPORTANT.</strong> To prevent lagging in the post-processing part later, make sure <code>save_interval</code> is approximately ( \frac{\text{no_of_iterations}}{10} ), and not much less! Otherwise it might consume too much resources and memory.</td>
</tr>
</tbody>
</table>
### filename

(String). From the previous step, we would have saved the optimization data using `save_name` as the prefix. Set `filename` to be equal to `save_name` in order to view the optimization results. In the example above, we should have, `filename = “fitting_data”`.

| save_data_series | (List of integers). This list is `[n1,n2,...,nk,...]` and each optimization result `filename_nk.par` will be displayed. For example, if you saved
|                  | 1. fitting_data_1.par,
|                  | 2. fitting_data_2.par and
|                  | 3. fitting_data_3.par
|                  | as we did earlier, and you would like to view the first and third optimization, set `save_data_series=[1,3]`. You can also set it to `range(1,4)` to view all of them. |
Usage Manual

We will use command line in Windows.

Virtual environment

We use python 3.6.2 (any recent python 3 versions should be fine). See this link to read how to create a project in a virtual environment. Using virtual environment ensure that we do not mess up package versions for other projects.

Create a virtual environment in the folder MyProject, and we will place our files in it. We will install the following packages in the virtual environments.

pip install kero
pip install matplotlib
pip install scipy
pip install numpy

Methods

1. Go to the command line and enter the following commands.
   
   ```
   cd path/to/MyProject
   Scripts\activate
   ```

2. **Preliminary**

   Make sure that sample_data.csv is in MyProject. In fitting.py, set the following toggles. Note that plot_data is in pe.load_data(). You can replace True with 1 and False with 0 as well.
   
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_data</td>
<td>True</td>
</tr>
<tr>
<td>do_initial_testing</td>
<td>False</td>
</tr>
<tr>
<td>start_op</td>
<td>False</td>
</tr>
</tbody>
</table>

   Now, in the virtual environment, run the command: **python fitting.py**
   
   A plot like figure 1 will be displayed, showing data size (number of data points) and the peak value. The first few data points will be displayed as well.

3. **Initial testing stage**

   In fitting.py, set the following toggles.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_data</td>
<td>False</td>
</tr>
<tr>
<td>do_initial_testing</td>
<td>True</td>
</tr>
<tr>
<td>start_op</td>
<td>False</td>
</tr>
</tbody>
</table>

   Set `collection_of_p_init`. You can place as many guesses as you want. In this example,

   ```
   collection_of_p_init = [
   [0,1e-3,0.5e-3,1.2e-7], # guess 1
   [0,1e-2,0,1e-7], # guess 2
   ```
we try using 3 different initial parameters, where each of them is of the format [G,k1,k2,k3], corresponding to the 4 parameters that we want to fit. Then, run the command: python fitting.py
The results are shown in figure 2. It can be seen that the first and third guesses are closer to the experimental data; we might want to use the parameters near these values for random initiation in the optimization stage. However, in this example, we will not follow closely just for demonstration sake.

4. Optimization stage
In fitting.py, set the following toggles.

<table>
<thead>
<tr>
<th></th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_data</td>
<td></td>
</tr>
<tr>
<td>do_initial_testing</td>
<td>False</td>
</tr>
<tr>
<td>start_op</td>
<td>True</td>
</tr>
</tbody>
</table>

The settings are described in the Overview Section. An example:

```python
p_init_max = [1e-10,0,0,9e-7] # [Gmax,k1max,k2max,k3max]
p_init_min = [-1e-10,2e-2,2e-3,1.2e-7] # [Gmin,k1min,k2min,k3min]
no_of_tries = 3
save_name = 'fitting_data'
es.learning_rate = 1e-16
no_of_iterations = 10
save_interval = 1
```

For demonstration sake, I set no_of_iterations to a small value. By right, we should iterate longer and adjust learning_rate up or down depending on whether the MSE is decreasing too slowly or too quickly. Also, notice that I used roughly 100 times the highest recommended learning rate to save time.

Then, run the command: python fitting.py.
Since we set save_name = 'fitting_data', we should now see fitting_data_1.par, fitting_data_2.par and fitting_data_3.par created in our folders. Recall that the aim of this algorithm is to minimize this MSE value, as shown in figure 3.

5. Viewing the optimization result
We want to load all 3 fitting_data files we have created. Thus, in fitting _post_processing.py, set
```
filename = 'fitting_data'
save_data_series = [1,2,3]
```
Run the command: python fitting _post_processing.py
Figure 4 shows how after every iteration, the resulting curve is approaching the experimental data. ***
Finally, to see what are the optimized $p = (G, k1, k2, k3)$ values, you can see that the periodical update of these values is shown in the command line, as shown in figure 5.

*** Note that the MSE values are not strictly decreasing all the time. This means that the landscape for parameter optimization is highly non-uniform, and the algorithm might have jumped from one valley to another, approaching different local minima. To prevent this from happening, use smaller learning rate at the expense of optimization time. Furthermore, note that there is no guarantee that the minimum value of MSE achieved is the global minimum.

![Figure 2](image2.png)

**Figure 2.** (A) The curves show 3 different plots from 3 different guesses of initial parameters. The red curve is the experimental data. (B). The console shows recommended setting for `learning_rate` variable which we can adjust in the optimization stage.

```
<table>
<thead>
<tr>
<th>TRY</th>
<th>I</th>
<th>mae</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0</td>
<td>175.714971001072335</td>
</tr>
<tr>
<td>1/3</td>
<td>1</td>
<td>165.0814444364903</td>
</tr>
<tr>
<td>1/3</td>
<td>2</td>
<td>154.86712542855432</td>
</tr>
<tr>
<td>1/3</td>
<td>3</td>
<td>145.1602001086188</td>
</tr>
<tr>
<td>1/3</td>
<td>4</td>
<td>135.8673138812225</td>
</tr>
<tr>
<td>1/3</td>
<td>5</td>
<td>127.311866592634</td>
</tr>
<tr>
<td>1/3</td>
<td>6</td>
<td>119.21593072184914</td>
</tr>
<tr>
<td>1/3</td>
<td>7</td>
<td>111.90788111502216</td>
</tr>
<tr>
<td>1/3</td>
<td>8</td>
<td>104.744900139000</td>
</tr>
<tr>
<td>1/3</td>
<td>9</td>
<td>98.39948220838763</td>
</tr>
<tr>
<td>1/3</td>
<td>10</td>
<td>92.62068590133833</td>
</tr>
<tr>
<td>2/3</td>
<td>0</td>
<td>124.64017385955859</td>
</tr>
<tr>
<td>2/3</td>
<td>1</td>
<td>116.43265479008446</td>
</tr>
<tr>
<td>2/3</td>
<td>2</td>
<td>108.2655394675114</td>
</tr>
<tr>
<td>2/3</td>
<td>3</td>
<td>101.6230008982476</td>
</tr>
<tr>
<td>2/3</td>
<td>4</td>
<td>95.04382290271985</td>
</tr>
<tr>
<td>2/3</td>
<td>5</td>
<td>89.01225278967295</td>
</tr>
<tr>
<td>2/3</td>
<td>6</td>
<td>83.52851987620653</td>
</tr>
<tr>
<td>2/3</td>
<td>7</td>
<td>79.98498184783002</td>
</tr>
<tr>
<td>2/3</td>
<td>8</td>
<td>74.1035815359164</td>
</tr>
<tr>
<td>2/3</td>
<td>9</td>
<td>70.25387676243814</td>
</tr>
<tr>
<td>2/3</td>
<td>10</td>
<td>66.82468080943598</td>
</tr>
</tbody>
</table>
```

**Figure 3.** Optimization in progress. MSE shows a decreasing trend.
initial G,k1,k2,k3 = [1e-10, 0.004550609582970161, 0.0014400866372871897, 1.0024566484707023e-07]

(A) $n$ vs $t$

(B) $n$ vs $t$
Figure 4. (A), (B) and (C). Green scatter points show the experimental data. As the algorithm is iterated, the curve goes from red to blue. The figures generally show that the curve approaches the experimental data and that MSE trend decreases along the iteration of $i$.

Figure 5. The command line shows the evolution of the parameters $p = (G, k_1, k_2, k_3)$. 
3. Using Scipy.optimize.least_squares

Here is the link to the least square function.

In the description, least_square “finds a local minimum of the cost function $F(x)$: minimize $F(x) = 0.5 \sum \rho(f_i(x)^2)$, $i = 0, ..., m - 1$” within a given bound.

Recall that we are given many data points $\{(n_i, t_i)\}$ governed by some function $n(t)$ which is often not analytically solvable from the ODE and difficult to determine exactly since the data points are often noisy (which is why we are doing this in the first place). Then, we identify $f_i(x)$ with the norm $||n(t) - n_0||$ where $x \rightarrow t, f_i \rightarrow n$ for each $i = 0,1, ..., m - 1$, $n$ is the model function whose parameter we want to optimize, and $n_0$ is the observed/experimental value of $n = n_i$ corresponding to $t = t_i$ for some $i$. In our implementation, $m$ corresponds to the total number of data points available or

We will implement the second option since it is more general. Also, in this implementation we generalize so that $f_i(x)^2$ is not just squared value, but norm-squared value so that for each data point $i$ we allow multi-valued function $f_i(x)$.

Scripts Overview

scipyode_header.py

RK45_wrapper(). See section 2, Scripts Overview.

read_csv(). See section 2, Scripts Overview.

load_data(). See section 2, Scripts Overview.

cubicODE.py

This is the main script for optimization.

PART 1

Write the relevant ODE here. The equation $\frac{dn}{dt} = f(t, n)$ is written as def $f(t, n, f_{args})$. Note that $f_{args}$ is the list of parameters that will be optimized.

Load data using load_data() defined in scipyode_header.py. Customize what you want to load as you see fit.

PART 2

initial_guess_mode. This mode is for us to try out the initial guesses of the parameters. It plots the experimental data points and the data points generated using initial guesses, as shown in figure. Then we can decide which initial guesses to use for actual optimization.
**optimize_mode.** This mode is where the optimization occurs. The target function is defined to return `collection_of_f_i`. This collection contains the norm $|\ln(t) - n_0|$ we described earlier, which corresponds to $f_i(x)$ in scipy documentation. This mode will save the optimized result into .par file.

**cubicODE_post_processing.py**

Load the optimized parameters and plot them.

**Methods**

1. **PART 1. Loading data.** In scipyode_header.py, customize `load_data()` as you see fit. As it is, it will load the data starting from a maximum point, and assume that the csv file contains 2 columns: column 1 is for variable $t$, column 2 is $n$. The file does not have any header.

   Go into cubicODE.py, set the following toggles.

<table>
<thead>
<tr>
<th>plot_data</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_guess_mode</td>
<td>False</td>
</tr>
<tr>
<td>optimize_mode</td>
<td>False</td>
</tr>
</tbody>
</table>

   Run `cubicODE.py`. You will see figure like figure 1. In this example, the relevant data points (blue) are the points that overlap with the orange plot.

2. **PART 1. Define the ODE.** Write the ODE in `def f(t,n,f_args)`.

3. **PART 2. Choosing initial guesses.** Set any number of initial guesses you want to try out with by setting `collection_of_initial_guesses`. Since we optimize $[G,k1,k2,k3]$, an example guess is $[0,1e-3,0.5e-3,1.2e-7]$. Set the following toggles and then run `cubicODE.py`. Plot like figure 6 will be generated.

<table>
<thead>
<tr>
<th>plot_data</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_guess_mode</td>
<td>True</td>
</tr>
<tr>
<td>optimize_mode</td>
<td>False</td>
</tr>
</tbody>
</table>

4. **PART 2. Optimization.** Once the choices of initial guesses are decided upon, set the following toggles and then run `cubicODE.py`. The optimized parameters from each initial guess will be saved as a .par file, and the save name is set by the variable `save_name`. The code in our example used 3 initial guesses and `save_name = "20190110_cubic"`. Thus, `20190110_cubic_1.par`, `20190110_cubic_2.par` and `20190110_cubic_3.par` will be created.
plot_data | False
---|---
initial_guess_mode | False
optimize_mode | True

5. **Loading optimized parameters.** Go to `cubicODE_post_processing.py`. If we want to load all the 3 saved .par files, then set `load_name = "20190110_cubic"` and `serial_number = [1,2,3]`. Then run `cubicODE_post_processing.py`. The plot results are shown in figure 7(A, B, C), showing good fit. The optimized parameters for each initial guess can be read off from the console, as in figure 7(D).

![Graph](image)

Figure 6. Green plot shows the experimental data points while the rest are generated using RK45 using 3 different initial guesses, given by `collection_of_initial_guesses = [ [0,1e-3,0.5e-3,1.2e-7], [0,1e-2,0,1e-7], [0,0,1.5e-3,1e-7] ]`. 
Figure 7. (A, B, C) shows the results of parameter estimation from 3 different initial guesses. (D) shows the results in the console. For each serial number, the optimized parameters are printed as \([G, k_1, k_2, k_3]\).