Synthetic Data Course on LearnCrunch: Preliminary Exercise

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, January 2023

1 Introduction

The purpose is to get you started with one type of synthetic data – creating your own – with one particular purpose in mind: reconstructing unobserved daily measurements in a non-periodic time series with monthly observations. In the process, you will gain a first exposure to metrics that measure the quality of synthetic data, and what "quality" means to begin with. Many of the concepts and good practices explored during this exercise also apply to different types of synthetizations, as we shall see during the course. The various tasks come with testing using a programming language. This course is based on Python.

In this case, the daily values can be computed exactly. Let's pretend that we don't know them when generating the synthetic data. In the end we will use them to see how good we were at reconstructing (synthetizing) them. The problem is similar to the illustration in Figure 1: you are dealing with a time series where the observations are the orange dots, in this case equally spaced by 80-min increments. You don't know what the smooth curves look like; you can't tell from the data. Here, I produced 16 modified copies of the observed data (the 80-min measurements). I used interpolation – one of several techniques to synthetize data – to produce them. Combined together, these 16 sets provide 5-min granularity resulting in the blue curve in Figure 1. In fact here, the 5-min data was actually availble and represented by the red curve. I pretended it did no exist, but in the end I used it to assess the quality of the synthetization.

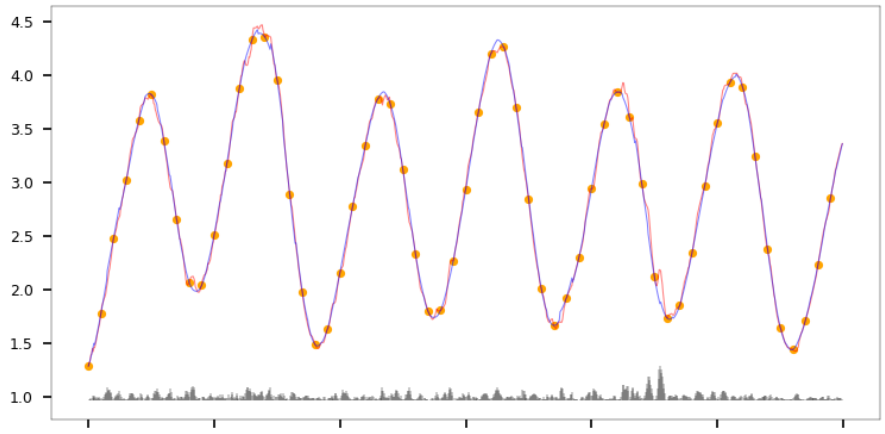

Figure 1: Tides at Dublin (5-min data), with 80 mins between interpolating nodes

The problem that we will work on is similar: getting data that measures the distances between the planets and the Sun, to see how frequently Earth, Venus (or Jupiter) are aligned on the same side of the Sun. For instance, in case of almost perfect alignment, the apparent locations of Jupiter and Mars are identical to the naked eye in the night sky. Is there a chance you might see that event in your lifetime? You'll get an answer to that curious question, but most importantly, the goal is to get you familiar with one aspect of data synthetization. Rather than 80-min observations, we will use monthly or quarterly observations. And we will reconstruct more granular data via interpolation, leading to a decent amount of synthetic data mimicking the observed time series. Then, we discuss assessing the quality of the synthetized data, and how more general modeling techniques could be used instead.

2 The problem

We first create a dataset with daily measurements of the distance between Earth and Venus, and interpolate the distance to test how little data is needed for good enough performance: can you reconstruct daily data from monthly observations? What about quarterly or yearly observations? Then, the purpose is to assess how

a specific class of models is good at synthetizing not only this type of data, but at the same time other types of datasets like the ocean tides in Figure 1 or the Riemann zeta function in Figure 2.

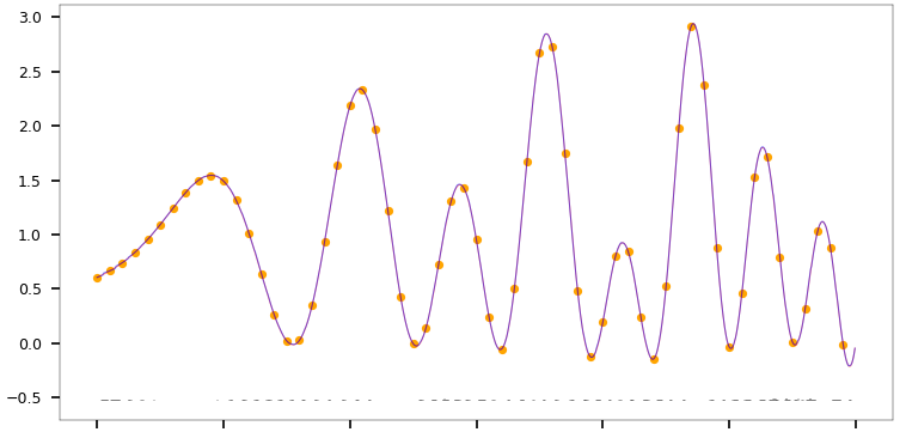

Figure 2: Interpolating the real part of $\zeta(\frac{1}{2}+it)$ based on orange points

The planetary fact sheet published by the NASA contains all the information needed. It is available here. I picked up Venus and Earth because they are among the planets with the lowest eccentricities in the solar system. For simplicity, assume that the two orbits are circular. Also assume that at a time denoted as $t = 0$, the Sun, Venus and Earth were aligned and on the same side (with Venus between Earth and the Sun).

Note that all the major planets revolve around the sun in the same direction. Let $\theta_V, \theta_E, R_V, R_E$ be respectively the orbital periods of Venus and Earth, and the distances from Sun for Venus and Earth. From the NASA table, these quantities are respectively 224.7 days, 365.2 days, 108.2×10^6 km, and 149.6×10^6 km. Let $d_V(t)$ be the distance at time t, between Earth and Venus. You first need to convert the orbitial periods into angular velocities $\omega_V = 2\pi/\theta_V$ and $\omega_E = 2\pi/\theta_E$ per day. Then elementary trigonometry leads to the formula

$$d_V^2(t) = R_E^2\left[1 + \left(\frac{R_V}{R_E}\right)^2 - 2\frac{R_V}{R_E}\cos\Big((\omega_V - \omega_E)t\Big)\right]. \tag{1}$$

The distance is thus periodic, and minimum and equal to $R_E - R_V$ when $(\omega_V - \omega_E)t$ is a multiple of 2π. This happens roughly every 584 days.

2.1 Steps to complete

Before starting the exercise, read the documentation about the interpolation method that I use. In particular, focus on section 2.1 about ocean tides, in my article "New Interpolation Methods for Synthetization and Prediction" available here, along with the details about the Python program `interpol_fourier.py`. The password to open the PDF document is `MLT12289058`. You can use a different interpolation technique, but one of the goals here is to learn how to use code written by a third party, and modify it for your own needs if necessary.

The exercise consists of the following steps:

Step 1: Use formula (1) to generate daily values of $d_V(t)$, for 10 consecutive years, starting at $t = 0$.

Step 2: Use the Python code `interpol_fourier.py` on my GitHub repository here. Interpolate daily data using one out of every 32 observations: using a power of 2 such as $32 = 2^5$ will let the code do everything nicely including producing the chart with the red dots, in the same way that I use one observation out of 16 for the ocean tides (80 minutes = 16×5 minutes). Conclude whether or not using one measurement every 32 days is good enough to reconstruct the daily observations. See how many nodes (the variable `n` in the code) you need to get a decent interpolation.

Step 3: Add planet Mars. The three planets (Venus, Earth, Mars) are aligned with the sun and on the same side when both $(\omega_V - \omega_E)t$ and $(\omega_M - \omega_E)t$ are almost exact multiples of 2π, that is, when both the distance $d_M(t)$ between Earth and Mars, and $d_V(t)$ between Earth and Venus, are minimum. In short, it happens when $g(t) = d_V(t) + d_M(t)$ is minimum. Assume it happened at $t = 0$. Plot the function $g(t)$, for a period of time long enough to see a global minimum (thus, corresponding to an alignment). Here ω_M is the orbital velocity of Mars, and its orbit is approximated by a circle.

Step 4: Repeat steps 1 and 2 but this time for $g(t)$. Unlike $d_V(t)$, the function $g(t)$ is not periodic. Alternatively, use Jupiter instead of Venus, as this leads to alignments visible to the naked eye in the night sky: the apparent locations of the two planets coincide.

Step 5: A possible general model for this type of time series is

$$f(t) = \sum_{k=1}^{m} A_k \sin(\omega_k t + \varphi_k) + \sum_{k=1}^{m} A'_k \cos(\omega'_k t + \varphi'_k) \tag{2}$$

where the $A_k, A'_k, \omega_k, \omega'_k, \varphi_k, \varphi'_k$ are the parameters, representing amplitudes, frequencies and phases. Show that this parameter configuration is redundant: you can simplify while keeping the full modeling capability, by setting $\varphi_k = \varphi'_k = 0$ and re-parameterize. Hint: use the angle sum formula (Google it).

Step 6: Try 10^6 parameter configurations of the simplified model based on formula (2) with $m = 2$ and $\varphi_k = \varphi'_k = 0$, to synthetize time series via Monte-Carlo simulations. For each simulated time series, measure how close it is to the ocean tide data (obtained by setting `mode='Data'` in the Python code), the functions $g(t)$ or $d_V(t)$ in this exercise, or the Riemann zeta function pictured in Figure 2 (obtained by setting `mode='Math.Zeta'` in the Python code). Use a basic proximity metric of your choice to asses the quality of the fit, and use it on the transformed time series obtained after normalization (to get zero mean and unit variance). A possible comparison metric is a combination of lag-1, lag-2 and lag-3 auto-correlations applied to the 32-day data (planets) or 16-min data (ocean tides), comparing simulated (synthetic) versus observed data. Also, autocorrelations don't require normalizing the data as they are already scale- and location-invariant.

Step 7: Because of the curse of dimensionality [Wiki], Monte-Carlo is a very poor technique here as we are dealing with 8 parameters. On the other hand, you can get very good approximations with just 4 parameters, with a lower risk of overfitting. Read section 1.3.3 in my book "Synthetic Data and Generative AI" about a better inference procedure, applied to ocean tides. Also read chapter 13 on synthetic universes featuring non-standard gravitation laws to generate different types of synthetic time series. Finally, read chapter 6 on shape generation and comparison: it features a different type of metric to measure the distance between two objects, in this case the time series (their shape: real versus synthetic version).

2.2 Notes

I will post a solution to this exercise, and discuss solutions offered by the participants in the classroom. Regarding the oscillations of the $g(t)$ function in Steps 3 and 4, a 10-year time period is not enough – by a long shot – to find when the next minimum (alignment) will occur. Looking at a 10-year time period is misleading.

Finally, one way to check if two time series (after normalization) are similar enough is typically done by comparing their estimated parameters (the ω_k, A_k and so on) to see how "close" they are. This will be further discussed during the course. This is also how you would measure the similarity between a synthetic time series (or any kind of data for that matter), and the observed data that it is supposed to mimic. You don't compare the shapes: they might be quite different yet represent the same mechanical effect at play. To the contrary, two extracts from different time series may look very similar visually, but may come from very different models: it just happens by coincidence that they look quite similar on some short intervals.

In some sense, what you want to measure is the *stochastic* proximity. Another example is comparing the numbers π and $3/7$. If you search long enough, you will find two sequences of 5000 digits that are identical in both numbers. Yet these two numbers are very different in nature. Now if you compare $5/11$ and $3/7$, you will never find such identical sequences, not even 2-digit long, and you may conclude that the two numbers are very different, while in fact they are of the same kind. Same if you compare π with a rational number very close to π.

There is caveat with comparing datasets based on their estimated parameters: if the parameter set is redundant as in Step 5, you can have two perfectly identical datasets that have very different parameters attached to them. This is known as model identifiability in statistics. I will discuss it in the classroom. Of course, there are many different ways to measure proximity, as we shall see.

Finally, I encourage you to upgrade my Python code `interpol_fourier.py` to include a sub-function that computes $g(t)$ defined in Step 3. And of course, you can test this program on your own data, not just the ocean tides or planet-related data. You should try `'mode=Math.Zeta'` and see if you find anything special about the time series generated. If you don't, feel free ask in the classroom or on Slack. There is something very special about it! Don't forget to install the MPmath Python library to make it work.

3 Solution

I split the solution into two parts. First the computation of daily distances $d_V(t), d_M(t), g(t)$ in million of km, and how they can be recovered via interpolation. Then, the simulations discussed in steps 5–7.

3.1 Synthetization: steps 1–4

The distances for $d_V(t), d_M(t), g(t)$ are respectively in blue, orange and green. They are represented by the vertical axis. The time is represented by the horizontal axis. In Figure 3, the time unit is a day (daily observations). In Figure 4, we are dealing with yearly observations instead. The blue curve shows a minimum about every 584 days, confirming that Venus is closest to Earth every 584 days. As for $g(t)$, there is no periodic minimum. Yet after 2400 days, you get another strong minimum, then minima get higher and higher and you have to wait over 400 years to reach a new low as low as the first one at $t = 0$, when perefct allignment occured by construction.

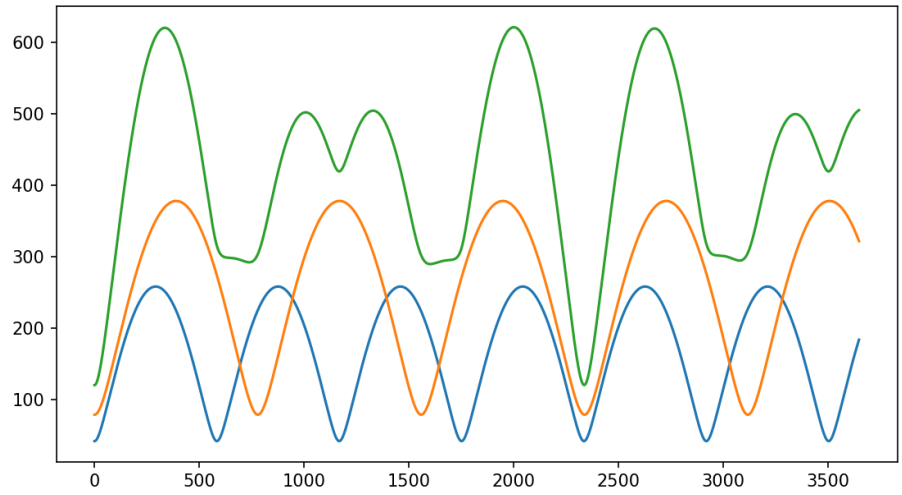

Figure 3: $d_V(t), d_M(t), g(t)$ in 10^6 km, first 10×365 days after alignment at $t = 0$

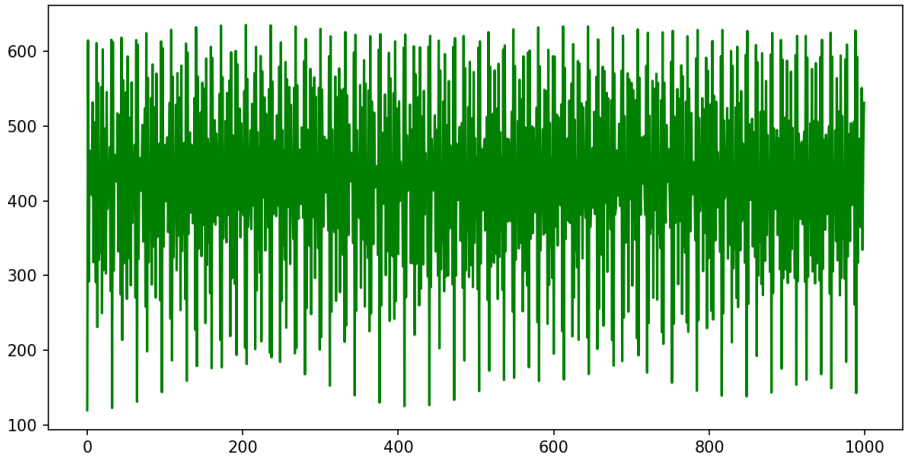

Figure 4: $g(t)$ over 1000 years, yearly measurements this time

Clearly, monthly data is well suited for interpolation, to recover daily data. But yearly data is not granular enough, and you can not expect to use (say) 50-year observations to recover or synthetize yearly observations.

3.1.1 Python code to compute distances

See below the code I created to compute the distances. It also produces an output file `gt_distances.txt` of daily observations for $g(t)$, used as input file for the interpolation program.

```
import numpy as np
```

```
import matplotlib.pyplot as plt

R_V = 108.2 # million km
R_E = 149.6 # million km
R_M = 228.0 # million km
ratio_V = R_V / R_E
ratio_M = R_M / R_E

pi2 = 2*np.pi
t_unit = 1 # (in number of days)

# time unit = 32 days, to interpolate daily values (1 obs ever 32 day)
omega_V = t_unit * pi2 / 224.7 # angular velocity per 32 days
omega_E = t_unit * pi2 / 365.2 # angular velocity per 32 days
omega_M = t_unit * pi2 / 687.0 # angular velocity per 32 days

time = []
d_V = []
d_M = []
d_sum = []
t_incr = 1 # (in number of days)
T = 365 * 10 # time period (in number of days)
OUT = open("gt_distances.txt","w")
for t in np.arange(0, T, t_incr):
   time.append(t)
   dist_V = R_E * np.sqrt(1 + ratio_V**2 - 2*ratio_V * np.cos((omega_V - omega_E)*t))
   dist_M = R_E * np.sqrt(1 + ratio_M**2 - 2*ratio_M * np.cos((omega_M - omega_E)*t))
   d_V.append(dist_V)
   d_M.append(dist_M)
   d_sum.append(dist_V + dist_M) # near absolute minimum every ~ 400 years
   OUT.write(str(dist_V + dist_M)+"\n")
OUT.close()
plt.plot(time,d_V)
plt.plot(time,d_M)
plt.plot(time,d_sum,c='green')
plt.show()
```

3.1.2 Interpolation / synthetization

I used `interpol_fourier.py` to interpolate the distances for $g(t)$, using `mode='Data'` and `n=8` (the number of interpolation nodes) as for the ocean tide dataset. Here the imput file was `gt_distances.txt` created by the Python code in the previous section.

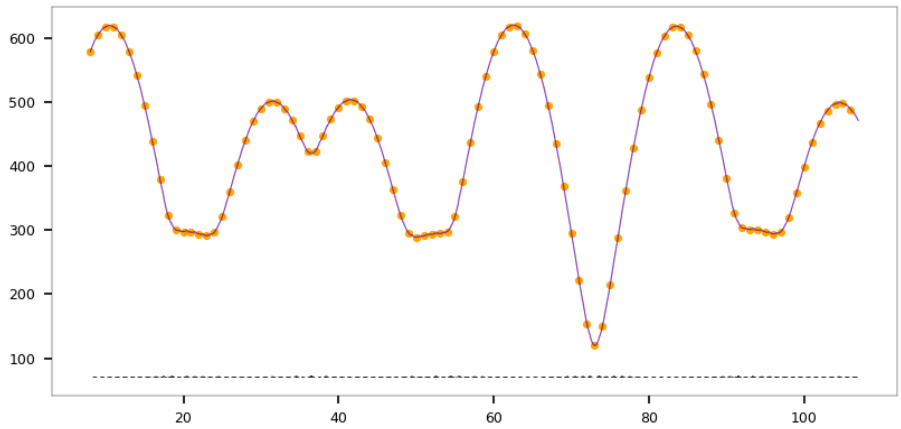

Figure 5: Daily interpolated values for $g(t)$, based on exact 32-day data in orange

Reducing the number of interpolation nodes from $n = 8$ to $n = 4$ starts showing a small error visible to the naked eye. With $n = 8$, you can't see the error as illustrated in Figure 5. I have no doubt that using one out every 64 days to reconstruct daily data (instead of one every 32) would still do a good job. In the

process I created 32 synthetic copies of the orange data to fill the gaps: not identical copies but instead different copies with the right distribution compatible with the orange data. Also keep in mind that $g(t)$ does not have any period, so any shifted version of it will be different. This is in contrast with the function $d_V(t)$. To run `interpol_fourier.py`, you need to change the input filename to `gt_distances.txt`, and set `t_unit=32`. The resulting plot in Figure 3 is truncated on the time (horizontal) axis, compared to Figure 5. Also the time unit on the horizontal axis is 32 days instead of one day as in Figure 3.

3.2 Simulations and comparison with real data

The answer to Step 5 is as follows. We have

$$A\cos(\omega t+\varphi)+A'\sin(w't+\varphi')=\alpha\sin\omega t+\beta\sin\omega' t+\alpha'\cos\omega t+\beta'\cos\omega' t,$$

with $\alpha=A\cos\varphi, \beta=-A'\sin\varphi', \alpha'=A\sin\varphi, \beta'=A'\cos\varphi'$. Thus the phase parameters φ, φ' are not necessary. However, removing them requires increasing m in Formula (2). Now, ignoring them, let's do the simulations with $m=2$. The Python code below deals with simulating $g(t)$ for the planet dataset. My conclusions follow after the code.

3.2.1 Python code

```
import numpy as np
import statsmodels.api as sm
import matplotlib as mpl
from matplotlib import pyplot as plt

# use statsmodel to compute autocorrel lag 1, 2, ..., nlags

#--- read data

IN = open("gt_distances.txt","r")
# IN = open("tides_Dublin.txt","r")
table = IN.readlines()
IN.close()

exact = []
t = 0
for string in table:
   string = string.replace('\n', '')
   fields = string.split('\t')
   value = float(fields[0])
   # value = np.cos(0.15*t) + np.sin(0.73*t)
   if t % 32 == 0: # 16 for ocean tides or Riemann zeta, 32 for planets (also try 64)
      exact.append(value)
   t = t + 1
nobs = len(exact)
time = np.arange(nobs)
exact = np.array(exact) # convert to numpy array

nlags = 8
acf_exact = sm.tsa.acf(exact, nlags=nlags)


#--- generate random params, one set per simulation

np.random.seed(104)
Nsimul = 10000

lim = 20
A1 = np.random.uniform(-lim, lim, Nsimul)
A2 = np.random.uniform(-lim, lim, Nsimul)
B1 = np.random.uniform(-lim, lim, Nsimul)
B2 = np.random.uniform(-lim, lim, Nsimul)
norm = np.sqrt(A1*A1 + A2*A2 + B1*B1 + B2*B2)
A1 = A1 / norm
A2 = A2 / norm
```

```
B1 = B1 / norm
B2 = B2 / norm
w1 = np.random.uniform(0, lim, Nsimul)
w2 = np.random.uniform(0, lim, Nsimul)
v1 = np.random.uniform(0, lim, Nsimul)
v2 = np.random.uniform(0, lim, Nsimul)

#--- generate Nsimul time series each with nobs according to model
#   measure fit between each realization and the real data
#   identify synthetized series with best fit

best_fit = 9999999.9
best_series_idx = 0
for i in range(Nsimul):
   if i % 5000 == 0:
      print("generating time series #",i)
   asimul = A1[i] * np.cos(w1[i]*time) + A2[i] * np.cos(w2[i]*time) + B1[i] *
       np.sin(v1[i]*time) + B2[i] * np.sin(v2[i]*time)
   acf = sm.tsa.acf(asimul, nlags=nlags)
   delta = acf - acf_exact
   metric1 = 0.5 * np.mean(np.abs(delta))
   corrm = np.corrcoef(exact,asimul)
   metric2 = 1 - abs(corrm[0,1])
   fit = metric1 # options: metric1 or metric2
   if fit < best_fit:
      best_fit = fit
      best_series_idx = i
      best_series = asimul
      acf_best = acf

print("best fit with series #",best_series_idx)
print("best fit:",best_fit)
print()

print("autocorrel lags, observed:")
print(acf_exact)
print()

print("autocorrel lags, best fit:")
print(acf_best)
print()

#--- plotting best fit

mu_exact = np.mean(exact)
stdev_exact = np.std(exact)
mu_best_series = np.mean(best_series)
stdev_best_series = np.std(best_series)
best_series = mu_exact + stdev_exact * (best_series - mu_best_series)/stdev_best_series
    # un-normalize

print("min: exact vs best series: %8.3f %8.3f" % (np.min(exact),np.min(best_series)))
print("max: exact vs best series: %8.3f %8.3f" % (np.max(exact),np.max(best_series)))
corrm = np.corrcoef(exact,best_series)
print("|correlation| between both %8.5f" % (abs(corrm[0,1])))

plt.scatter(exact, best_series, c='orange')
plt.show()
```

3.2.2 Conclusions

Before going into the details, I want to mention a few important facts. Monte-Carlo simulations are not a good solution with more than 3 or 4 parameters. Using 4 parameters sometimes lead to better results than the full model with 8 parameters in this case. Likewise using fewer simulations – say 10^4 instead of 10^6 – can lead to better results, especially in a case like this with multiple local minima. Also, using lag-1, lag-2, and lag-3

autorrelations is not enough to measure the "distance" (called `fit` in the Python code), between the real and synthetic data, to identify among 10^4 synthetized time series which one is the best representative of the type of data we are dealing with. Below is some other highlights and recommendations:

- All the discussion applies to *stationary* time series. It assumes that any non-stationary components (such as trends) have been removed, to apply the methods discussed here. The datasets in this exercise meet this requirement.
- To measure the quality of fit, it is tempting to use the correlation between synthetic and real data. However this approach favors synthetic data that is a replicate of the original data. To the contrary, comparing the two auto-correlation structures favors synthetic data of the same type as the real data, but not identical. It leads to a richer class of synthetic data, putting emphasis on structural and stochastic similarity, rather than being "the same". The latter results in overfitting.
- Try different seeds for the random generator, and see how the solution changes based on the seed. Also, rather than using the sum of absolute value of differences between various autocorrelation lags, try the max, median, or assign a different weight to each lag (such as decaying weights). Or use transformed auto-correlations using a logarithm transform.
- A classic metric to assess the quality of synthetic data is the Hellinger distance, popular because it yields a value between 0 and 1. It measures the proximity between the two marginal distributions – here, that of the synthetized and real time series. It is not useful for time series though, because you can have the same marginals and very different auto-correlation structures. Note that the metric I use also yields values between 0 and 1, with zero being best, and 1 being worst.
- The simulation was able to generate synthetic values outside the range of observed (real) values. Many synthetic data algorithms fail at that, because they use percentile-based methods (for instance, copulas) for data generation or to measure the quality (Hellinger is in that category). Empirical percentile distributions used for that purpose, including the Python version in the statsmodels library, have this limitation. It can be overcome though. I will discuss this in the classroom.