Generating 2D Datsets with similar statistical properties

Algorithm Engineering 2023 Project Paper

Veit Hucke
Friedrich Schiller University Jena
Germany
veit.hucke@uni-jena.de

Marius Wank
Friedrich Schiller University Jena
Germany
marius.wank@uni-jena.de

ABSTRACT

Teaching how to interpret statistical properties of data is an important task. Making assumptions about the data we handle purely based on their statistical properties is a pitfall that can easily be avoided by highlighting the importance of data visualization. In this paper we present an algorithm that tackles that task while also accommodating the need for hardware-optimized algorithms to handle today's large amounts of data. Our algorithm improves on the code created by Justin Matejka and George Fitzmaurice by implementing it in C++ and introducing parallelism as well as vectorization to it. The idea is to generate a scatter plot with a target shape while maintaining identical statistical properties to an input dataset through simulated annealing. Our experiments show that even a naive conversion from Python to C++ can drastically improve performance and allows access to several tools that can widen the gap even further. Using these tools, we successfully improved the runtime of the code by up to 105x. We conclude that, even though the implementation is harder, utilizing these tools is necessary to handle big data in reasonable amounts of time.

KEYWORDS

C++, parallelism, vectorization, statistics, data visualization

1 INTRODUCTION

1.1 Background

According to Francis John Anscombe graphs are an important tool to gain a better understanding during statistical analysis[1]. He argues that by only looking at the statistical properties of data and not visualizing it we will only get a limited understanding of that data. To support his claim he created Anscombe's quartet, a visualization of four statistically nearly identical datasets (see Figure 1). The graph shows that, even though similar in properties, the datasets are vastly different. To further underline his point, we wanted to expand upon his idea and improve an algorithm that could generate any given target shape from a given dataset while maintaining statistical properties similar to the original. We decided to use the work of Matejka and Fitzmaurice[6] as our starting point.

According to Moore's Law from 1965 the number of transistors on a computer chip (and therefore its power) will double every two years[7]. Nowadays we know that this law does not hold anymore as we approach physical limitations. With this in mind we also wanted to design our algorithm in a way that it utilizes modern techniques to improve runtime, meaning a hardware-specific implementation.

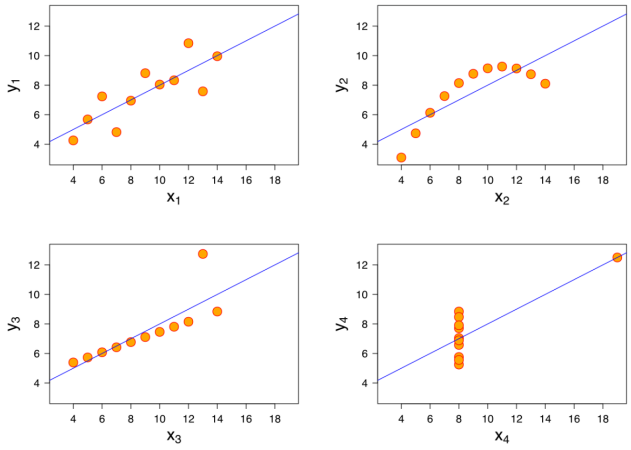

Figure 1: Anscombe's quartet. Each dataset contains 11 datapoints and has a similar mean of x and y (exact), sample variance of x and y (exact), correlation between x and y (to 2 decimal places), linear regression line (to 3 decimal places for a and to 2 decimal places for c) and coefficient of determination of the linear regression line (to 2 decimal places)[1].

1.2 Related Work

As mentioned above, the algorithm by Matejka and Fitzmaurice[6] generates any given target shape from a given dataset while maintaining statistical properties similar to the original. The similarity is limited to 2 decimal places. Their idea was that even though creating a fully new dataset with similar properties is difficult, iteratively changing an existing one marginally to conserve the properties is easy. This was achieved by performing a small, biased perturbation on the dataset every iteration, where a translocation of a point is only accepted when it moves towards the target shape. To prevent getting stuck on local optima, a small chance that decreases every iteration is given to accept perturbations that did not move a point towards the target shape. After each iteration a check is done to see if the change of statistical properties was within a given threshold. An example on how this process looks like can be found in Figure 2.

The algorithm accepting non-optimal solutions is called simulated annealing and was first introduced in 1995 by Kirpatrick et al.[5]. It uses the function $e^{-\frac{\Delta E}{kT}}$ as the probability to accept worse perturbations, where ΔE denotes the difference between the error functions of the current and previous step (in the case of Matejka and Fitzmaurice the distance to the closest point of the target shape), k the Boltzmann constant and T the temperature which is decreased every step to shrink the probability.

While this method proves to be successful, the performance leaves a lot of room for improvement. In the paper by Matejka and Fitzmaurice Python is used for the implementation causing the program to run for several minutes before finishing. Our implementation vastly improves upon that by using C++ as the programming language of choice.

Since Matejka and Fitzmaurice covered different generalized methods for creating datasets with similar statistics and dissimilar graphics we will not present them here to avoid repetition. We do however recommend them for a more in-depth understanding of the subject. These works include the publications by Chatterjee and Firat[3] as well as Govindaraju and Haslett[4].

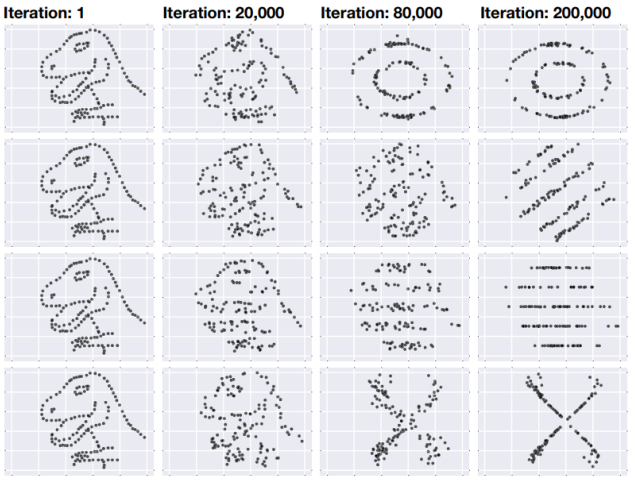

Figure 2: Example of the simulated annealing process by Matejka and Fitzmaurice. Each dataset has the same summary statistics to two decimal places: ($\bar{x}$=54.26, $\bar{y}$ = 47.83, sdx =16.76, sdy = 26.93, Pearson's r = -0.06). The image and description were taken directly from the original paper[6].

1.3 Our Contributions

The main goal of this project is to develop an application based on the work of Matejka and Fitzmaurice but with a stronger focus on performance and efficiency. We therefore chose the C++ programming language as a foundation, which is already the biggest difference to the original work.
Besides being inherently more performant than Python, it also allows to have significantly more control over the computer, making it possible to utilize its resources more efficiently. Which further specific improvements and techniques have been implemented will be described in detail in section 2.2.1.

1.4 Outline

In section 2 we are going to talk about the exact implementation of our algorithm. This includes specifics about how input and output data is processed as well as the C++ specific tools used to fabricate a hardware-optimized version of the simulated annealing approach.

In section 3 we share the results of our work and the setup that was used to compare our implementation with the one by Matejka and Fitzmaurice.

In section 4 we first discuss our implementation based on efficiency and portability. We also present ideas that can be used to improve our work in the future. In the end we judge if we achieved the goals of this study sufficiently.

2 THE ALGORITHM

2.1 Handling of input data

The basic approach we follow in our implementation is similar to the approach by Matejka and Fitzmaurice, but small adjustments have been made. In our implementation we only use the Datasaurus dataset as the input for demonstration purposes so the user only has to enter the preferred target shape as well as a path where the preferences.json is located. We use the preferences.json as a way to reduce the number of parameters the user has to enter in the command line. We did not implement support for all possible target shapes from the original paper as time was a limiting factor. After starting the program, the correctness of the input is checked and a dataframe is created from the Datasuarus dataset. After that the statistical properties of the input dataset are calculated and perturbations on the dataset are run through simulated annealing as long as defined by the user.

2.2 Simulated annealing

The algorithm that ensures both a perturbation towars a desired target shape and the conservation of the statistical properties of the original data is based on simulated annealing. There are other possibilities to achieve this goal, such as using genetic algorithms, but we chose to stick to the implementation of the original authors[6] because of its simplicity.

Algorithm 1

```
current_df ← initial_df
new_df ← initial_df
for n iterations do
    new_df ← Perturb(current_df, temp)
    if is_error_okay(new_df, current_df) then
        current_df ← new_df
function Perturb(df, temp):
    pos ← Random()
    orig_point ← df[pos]
    loop
        new_point ← MoveRandom(orig_point)
        if Dist(new_point) < Dist(orig_point)
    or temp > Random() then
            new_df ← df
            new_df[pos] ← new_point
    return new_df
```

In general, a *perturb_once()* function is called which accepts a DataFrame, containing the x- and y-coordinates of all points specified in the imported file. A random (x, y) pair from the input DataFrame is drawn and perturbed slightly. The value by which the original pair is modified is normal distributed which is itself modified by a factor *shake* specified by the user. The euclidean distance of both the original pair (x, y), as well as the modified pair (x_new, y_new) is calculated and compared. Now there are multiple condition that can be met in order to assert a successful perturbation of the data:
The most intuitive condition is that the distance of the modified point to the target shape is smaller than that of the original point. Alternatively, a worse state of the system can also be accepted, based on a temperature randomly drawn from the distribution unif[0.0, 1.0). This is part of the actual simulated annealing and prevents the system to eventually get stuck in a local optimum. As a cooling schedule, we adapted the smoothed monotonic function proposed by Matejka and Fitzmaurice to preserve comparability. The schedule starts at a temparature of 0.4 and terminates at a temperature of 0.01 per default, but these values may also be altered by the user. Finally, if the new distance is smaller than a lower bound *allowed_distance* defined by the user, the new state is accepted as well. This is because we consider a pair (x, y) being optimal and in its final position as soon as this threshold is reached.
This perturbation-comparison mechanism is repeated as long as none of these conditions are met. However, if one condition eventually becomes true, a copy of the initial DataFrame is made, the new points are overwritten and the copy is returned by the function.
The *perturb_once()* function itself is called for multiple iterations, where after each iteration a function *is_error_okay()* is called that ensures the conservation of the statistical properties. As in the original work from Matejka and Fitzmaurice, we consider the properties conserved if they are equal to two decimal points. If this is the case, the initial DataFrame is overwritten with the slightly modified one, the data of the new DataFrame is written to a buffer and the next iteration of *perturb_once()* is invoked. Otherwise, the initial DataFrame will not be overwritten and it goes into the next iteration of modification as well.

2.2.1 Improvements. As mentioned above, using the C++ programming language was not only the first major performance improvement that has been achieved, it also allowed having more control over the computer.
Vectorization. Keeping in mind that the data the algorithm works on is stored in large vectors, using vectorization is an obvious technique to implement. The statistical properties have to be calculated in every iteration to ensure that they do not change due to the perturbation of single points, so applying the same operations on huge chunks of data implies a significantly more efficient usage of the computers resources. The vectorization is achieved by guiding the compiler using the *omp simd* pragma. Furthermore, loops in the statistical methods are additionally accelerated by using the *reduction* clause as the body of the loops only contain simple summations. Finally, the used vectors are 64 byte aligned, supporting the compiler in vectorizing the code and resulting in a faster access and more efficient usage of the bandwidth between main memeory and the CPU.
Data handling. To further improve the internal processing of the data, we decided to keep the coordinates in separate x- and y-vectors that are part of a custom DataFrame struct. As the statistical properties are calculated independently for these x- and y-coordinates, the usage of a Structure of Arrays allows a strided 1 memory access, efficiently making use of the 64 byte of a cache line.
We also cosidered a more efficient way to output the data again: Compared to the original script, we did not save multiple .csv files during the iterations. Our algorithm opens an output file once and, everytime an acceptable change in the coordinates happens, the update is saved in a buffer, which is finally written to the file once after all iterations finished.
RNG. The standard library of C++ natively supports the generation of random numbers, however, considering performance, the built-in algorithms were still a major bottleneck as they have to be called in every iteration for multiple times. Our approach to improve this hotspot was to implement custom random number generators based on the xoroshiro128+ algorithm[2] and the Box-Muller transform to attain uniform and normal distributed numbers respectively.
Distances. The original implementation of the algorithm that finds the minimal distance between a point (x, y) and its closest line segment was not working properly for some shapes, such as the *Star*. Due to this circumastance, a new algorithm has been implemented that is based on extending the line segment and projecting the point (x, y) onto it. On the one hand, this algorithm resolved the issue with not finding a fitting optimum for some target shapes. On the other hand, as a positive side effect, the performance was slightly increased as well due to less if...else statements.

2.3 Processing of output data

To create the visualization of our output data we decided to stick to Python as it offers the easiest way for creating videos from multiple scatter plots via the libraries matplotlib. First we read the data from the output .csv file using the pandas library to create an iterable dataframe. We then simply iterate over the unique frames saved in the .csv, create a dataframe for the x- and y-coordinates corresponding to the current frame and create a scatter plot from these using the matplotlib library.

Each scatter plot is saved to a list which is used afterwards to create an animation using the animation class of matplotlib. As a result, we get a .gif file in witch we can see the time evolution of the scatter plot.

3 EXPERIMENTS

Table 1 shows the comparison between our code and the implementation by Matejka and Fitzmaurice for different target shapes. Our implementation achieves a 54x to 105x faster runtime depending on the concrete target shape. Independent of the target shape the generation of the output .csv file containing the x- and y-coordinates of the frames took about 0.2 seconds. This means that the visualization Python script we used was responsible for about 90 to 95% of the runtime in every case.

Table 1: Comparison of the running times of the simulated annealing process plus the visualization for different target shapes for 200,000 iterations. The input data was always the Datasaurus dataset from the paper by Matejka and Fitzmaurice. Measurements were taken on a desktop computer, running Windows 10 Educational with 16 GB RAM and one AMD Ryzen 5 2600X CPU. The underlying SSD was a 256 GB 850 Pro SATA III.

Team	Language	Target shape	Running time (s)
Wank and Hucke	C++	Circle	**4.16**
Matejka and Fitzmaurice	Python	Circle	238
Wank and Hucke	C++	Bullseye	**4.41**
Matejka and Fitzmaurice	Python	Bullseye	242
Wank and Hucke	C++	Dots	**2.68**
Matejka and Fitzmaurice	Python	Dots	283
Wank and Hucke	C++	X	**2.76**
Matejka and Fitzmaurice	Python	X	244

4 OUTLOOK & CONCLUSION

Even though our implementation achieves a vastly faster runtime it is far from perfect. A major bottleneck is the visualization which uses Python, further proving our point that Python should not be used for performance focused programming. In the future the visualization could be implemented using C++ graphics libraries like OpenGL or SDL. For the purpose of a more dynamic use of this code a future approach could include taking user input directly via a graphical user interface where the users can draw the input and target shape of the dataset by themselves. We experimented with the SFML library which proved to be a simple and effective tool. The main reason we did not include it in the final release is that portability was an issue since SFML requires the use of a version that matches the used compiler of the user. We did not want to include every possible version as we did not have the time to ensure sufficient portability for every OS and compiler. Also, since this work represents a proof-of-concept, the generated data will always be overwritten when running the code again. To make the program viable for applications outside of testing it would be a nice addition to have it save multiple different runs separately. Another thing to improve is the usage of our random number generator. The seeds we used for testing purposes are hardcoded and could be changed manually, but a more dynamic approach would be favorable. Finally, exception handling is a small but important task to ensure smooth usage of the program. This could include checking the contents of the input and output .csv files for correctness. However, we did not implement handling for all possible exceptions at this time as this was not necessary in the confined testing setup we used.

Even though there are many possibilities to improve our code even further, the here presented results achieved the goal to improve the approach by Matejka and Fitzmaurice and enables users to process big data in reasonable amounts of time, proving the value of hardware-optimized implementations.

REFERENCES

[1] F. J. Anscombe. 1973. Graphs in Statistical Analysis. *The American Statistician* 27, 1 (1973), 17–21. https://doi.org/10.1080/00031305.1973.10478966 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00031305.1973.10478966

[2] David Blackman and Sebastiano Vigna. 2021. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)* 47, 4 (2021), 1–32.

[3] Sangit Chatterjee and Aykut Firat. 2007. Generating Data with Identical Statistics but Dissimilar Graphics: A Follow up to the Anscombe Dataset. *The American Statistician* 61, 3 (2007), 248–254. http://www.jstor.org/stable/27643902

[4] S. J. Haslett and K. Govindaraju. 2009. CLONING DATA: GENERATING DATASETS WITH EXACTLY THE SAME MULTIPLE LINEAR REGRESSION FIT. *Australian & New Zealand Journal of Statistics* 51, 4 (2009), 499–503. https://doi.org/10.1111/j.1467-842X.2009.00560.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-842X.2009.00560.x

[5] Scott Kirkpatrick and Gregory B Sorkin. 1995. Simulated annealing. (1995).

[6] Justin Matejka and George Fitzmaurice. 2017. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1290–1294. https://doi.org/10.1145/3025453.3025912

[7] Gordon E Moore et al. 1965. Cramming more components onto integrated circuits.