applied sciences

MDPI

Article

Speeding up Statistical Tolerance Analysis to Real Time

Peter Grohmann and Michael S. J. Walter *

Faculty of Engineering, University of Applied Sciences Ansbach, 91522 Ansbach, Germany; peter.grohmann@hs-ansbach.de

* Correspondence: michael.walter@hs-ansbach.de; Tel.: +49-981-4877-559

Abstract: Statistical tolerance analysis based on Monte Carlo simulation can be applied to obtain a cost-optimized tolerance specification that satisfies both the cost and quality requirements associated with manufacturing. However, this process requires time-consuming computations. We found that an implementation that uses the graphics processing unit (GPU) for vector-chain-based statistical tolerance analysis scales better with increasing sample size than a similar implementation on the central processing unit (CPU). Furthermore, we identified a significant potential for reducing runtime by using array vectorization with NumPy, the proper selection of row- and column- major order, and the use of single precision floating-point numbers for the GPU implementation. In conclusion, we present open source statistical tolerance analysis and statistical tolerance synthesis approaches with Python that can be used to improve existing workflows to real time on regular desktop computers.

Keywords: computation time; statistical tolerance analysis; Monte Carlo simulation; sample size; statistical tolerance synthesis; tolerance engineering; Python

Citation: Grohmann, P.; Walter, M.S.J. Speeding up Statistical Tolerance Analysis to Real Time. *Appl. Sci.* **2021**, *11*, 4207. https://doi.org/10.3390/app11094207

Academic Editor: Maurizio Faccio

Received: 27 February 2021
Accepted: 23 April 2021
Published: 5 May 2021

1. Introduction

Manufactured components are subject to deviations that reduce the functional and aesthetic quality of the final product. For this reason, tolerances are specified based on the results of statistical tolerance analyses to minimize the degradation of geometrical accuracy and number of rejects in manufacturing. Additionally, economic aspects must also be considered. Needlessly tight tolerance specifications lead to higher expenses and an increase in production time due to the stricter demands on the manufacturing process. A common approach to satisfying both of these requirements is establishing statistical tolerance analyses based on Monte Carlo simulation to identify the cost-optimal manufacturing tolerance specification based on repeated random sampling and statistical analysis [1]. Monte Carlo simulation has consistently been the preferred choice of the tolerance community to evaluate the statistical implications of manufacturing deviations on the key characteristics of mechanical assemblies. Furthermore, Monte Carlo simulation is an established and common standard in a large variety of different scientific and industrial areas, because it allows simulating and tackling complex systems and processes. However, Monte Carlo simulation is computationally intensive, which means that the associated execution times can range from seconds to hours and even days. The ongoing digital transformation of manufacturing through the adoption of Industry 4.0 [2–4] requires that product characteristics be analyzed after each manufacturing process in order to establish a continuous information flow to enable a self-learning and self-adapting manufacturing process [5,6].

To support the digitization of manufacturing products with high geometric accuracy, it is therefore important to accelerate the statistical tolerance analysis. The runtime optimization of Monte Carlo simulation is nothing new to academic research. Various studies have discussed approaches to speed up Monte Carlo simulation, mainly focusing on either a proper distribution of the random sample generation among several local computers (such as [7]) or on carrying out the Monte Carlo simulation on a single GPU or on multiple GPUs (such as the application of NVIDIA's Compute Unified Device Architecture (CUDA)

in [7]). However, none of the existing work considers the specifics of tolerance engineering and investigates whether a Monte Carlo simulation using the GPU architecture has the potential to significantly speed up statistical tolerance analysis and statistical tolerance synthesis. The present paper attempts to investigate this using scientific open source computing libraries provided by the Python programming language.

The rest of this paper is structured as follows. Section 2 gives an overview of the software solutions used in tolerance engineering and provides a sufficient background concerning the basics of the programming language Python. Section 3 defines the research question and the main challenge of the paper. In Section 4, an overconstraint door hinge assembly is introduced, which serves as a case study. The implementations of a statistical tolerance analysis and statistical tolerance synthesis of the door hinge assembly are detailed in Sections 5.1–5.4. This is followed by a brief description of the runtime measurement procedures and details on the software and hardware used in Sections 5.5–5.7. The runtime results of the statistical tolerance analysis and statistical tolerance synthesis implementations are presented and discussed in Sections 6.1 and 6.2. In Section 6.2.2, the sample size required for credible statistical tolerance synthesis for the tolerance model is determined, followed by some conclusions in Section 7, in which recommendations for tolerance engineers and researchers are given to speed up statistical tolerance analysis to real time.

2. Numerical and Programming Foundations for This Study

For already about 30 years, commercial software tools for the analysis and optimization of tolerance specifications have been available. These so-called software tools for computer-aided tolerancing (CAT) enable the design engineer to carry out a statistical tolerance analysis of any given mechanical assembly in a virtual 3D-CAD environment. All of them use Monte Carlo simulation to obtain the statistical results. However, due to their high complexity and the high costs, very few companies employ commercial CAT tools. According to a survey from [8], 80% of the questioned companies in Germany use the spreadsheet program MS Excel to carry out vector-chain-based statistical tolerance analyses while the remaining respondents "relied on analytical calculation of tolerance stacking problems by hand, using established and simple approaches such as a worst-case tolerance analysis [9] or the root-sum-square approach" [8]. In research, however, the use of the commercial numeric computing environment MATLAB is the common standard.

In the present paper, the authors expand this portfolio of programming languages with Python. Python is an open-source dynamic scripting programming language which reduces the effort of implementing an initial prototype. It also provides a large number of libraries, already shipped, as well as external libraries, which are useful for a wide variety of scientific and engineering applications. The fact that Python is a dynamic programming language unfortunately is also one of its biggest weaknesses. This makes Python significantly slower than statically-typed programming languages, especially when a large number of operations needs to be performed repeatedly, as in the case of loops [10]. An efficient approach to improving the performance of Python code is to use NumPy Arrays, to replace explicit loop structures with "pre-compiled code written in a low-level language (e.g., C) to perform mathematical operations over a sequence of data" [11]. Figure 1a illustrates the general procedure and the runtime advantages for the Ishigami Function [12]:

$$y(x_1, x_2, x_3) = \sin(x_1) + 7 \cdot \sin^2(x_2) + 0.1 \cdot x_3^4 \cdot \sin(x_1). \quad (1)$$

Due to the drastic performance increase, NumPy Array programming has become the standard within the Python community, especially in the context of scientific computing. According to [13], NumPy played an important role in the software stack that led to the discovery of gravitational waves. For additional information on NumPy, see [14,15]. Another external open source Python library used in this paper is the CuPy library [16]. It can be implemented as a drop-in replacement for the NumPy library to execute computations on the GPU without having to apply NVIDIA's CUDA programming syntax (see Figure 1b).

```
[1]: import math as math
     import numpy as np

[2]: def ISHIGAMI_vectorized(samples):
         x1 = samples[:, 0]
         x2 = samples[:, 1]
         x3 = samples[:, 2]
         return np.sin(x1)+7*np.power(np.sin(x2),2)+0.1*np.power(x3,4)*np.sin(x1)

[3]: def ISHIGAMI(samples, sample_size):
         f = list()
         for i in range(sample_size):
             f.append(math.sin(samples[i, 0])+7*math.pow(math.sin(samples[i, 1]),2)+0.
      ↪1*math.pow(samples[i, 2],4)*math.sin(samples[i,0]))
         return f

[4]: samplesize = 1_000_000
     generated_samples = np.array(np.random.rand(samplesize,3), order='F')

[5]: %timeit ISHIGAMI_vectorized(generated_samples)

     76.8 ms ± 979 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

[6]: %timeit ISHIGAMI(generated_samples, samplesize)

     1.53 s ± 3.48 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

(**a**)

```
[7]: import cupy as cp

[8]: def ISHIGAMI_GPU(samples):
         x1 = samples[:, 0]
         x2 = samples[:, 1]
         x3 = samples[:, 2]
         return cp.sin(x1)+7*cp.power(cp.sin(x2),2)+0.1*cp.power(x3,4)*cp.sin(x1)

[9]: dev = cp.cuda.Device()
     samplesize = 1_000_000
     generated_samples = cp.array(cp.random.rand(samplesize,7), order='F')

[10]: %timeit ISHIGAMI_GPU(generated_samples)

      2.73 ms ± 53.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

(**b**)

Figure 1. (**a**) Vectorization with NumPy Arrays. (**b**) Use of the GPU architecture with CuPy.

Whenever one is working with larger array sizes with NumPy, as in the case of Monte Carlo simulation, it is necessary to be aware of row-major and column-major ordering. In order to have less overhead (due to memory access operations), the storage layout of the language needs to match the loop structure [17]. With row-major order, the elements are arranged consecutively along the row. With column-major order, they are arranged consecutively along the column [15,17]. The choice of the floating-point number precision defined in the IEEE/ISO/IEC 60559-2020 standard [18] is also of importance for both the performance and memory use of the statistical tolerance synthesis. For example, the GeForce 2060 RTX GPU used in this research, which is based on the NVIDIA Turing architecture, provides, on each of the installed 30 streaming multiprocessors, 64 FP32 ALUs (arithmetic logic units) designed for floating-point calculations. This allows fast floating-point arithmetic operations on the graphics units, see Table 1.

Table 1. Theoretical floating-point number performance of NVIDIA GEFORCE RTX 2060 [19].

Floating-Point Format	Theoretical Performance
FP16	12.90 TFLOPS [1]
FP32	6.451 TFLOPS
FP64	201.6 GFLOPS

[1] FLOPS (floating-point operations per second); T = Tera; G = Giga.

3. Research Question

A comparatively high number of samples is required to obtain statistically reliable results when performing statistical tolerance simulations (such as tolerance analysis and tolerance synthesis) [20–22]. However, with an increasing sample size, not only does the accuracy of the obtained results increase, but also the computational effort [23,24]. Recommendations range from 5000 [25] to 100,000 samples [24]. However, especially for more complex tolerance simulations, a size of 10,000 samples has become established among industrial experts and researchers [26]. The choice of the sampling procedure as well as the corresponding sample size have a significant influence on the reliability of a statistical tolerance simulation, and must therefore always be made with care. Hence, the key to speed up vector-chain-based statistical tolerance simulations is to accelerate the Monte Carlo simulation. The authors face the research question: Is it possible to realize an implementation of the Monte Carlo simulation in real time on a regular desktop computer with a sample size that satisfies the quality requirements in tolerance engineering?

In this paper, the definition of real time is based on response times of only a few milliseconds, which state-of-the-art automation solutions with a programmable logic controller or fieldbus-based production lines manage. In the following sections, statistical tolerance analyses and statistical tolerance syntheses are developed and executed with different programming approaches and varying sample sizes. Therefore, a non-trivial case study of an overconstraint door hinge assembly is used. We aim to quantify (i) how the computation times behave for different sample sizes with different implementations and (ii) whether, and if so how significantly, a tolerance simulation is sped up.

4. Case Study: Overconstraint Door Hinge Assembly

The overconstraint door hinge assembly is shown in Figure 2, following [27,28]. It should be noted in advance that this supposedly 'simple' assembly (the assumption of a one-dimensional chain with purely linear dependencies seems obvious) is, however, significantly influenced by interactions between the occurring deviations (due to the overconstraint design), and thus is non-trivial.

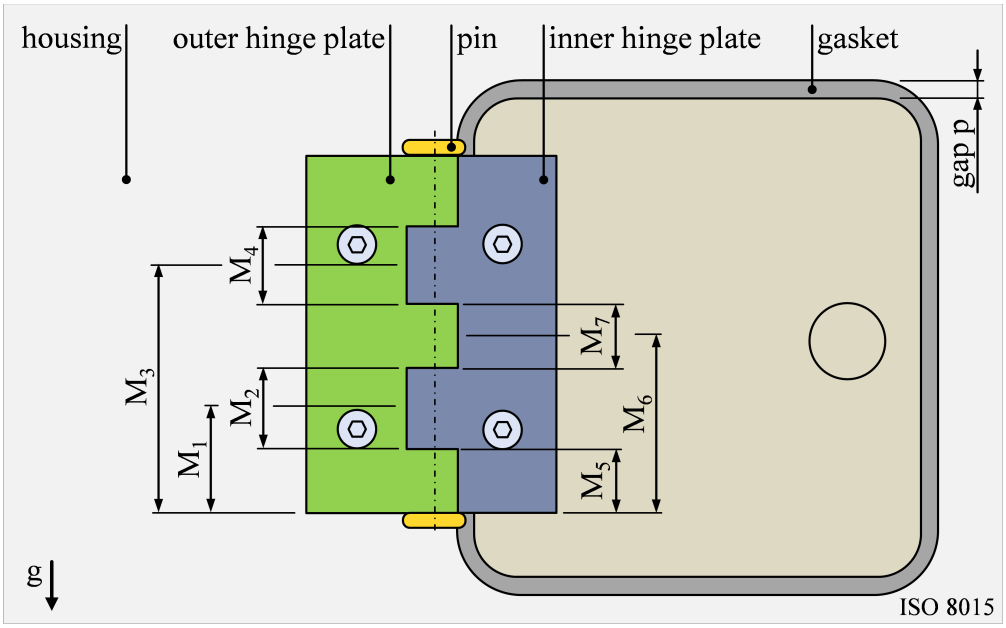


Figure 2. Overconstraint door hinge assembly.

The hinge consists of two main parts (the outer and inner hinge plates) connected by a pin. The hinge enables the opening and closing of a cover plate. Due to manufacturing deviations of the dimensions M_1 to M_7, a variation of the vertical position of the cover results. These deviations are limited by the tolerances specified in Table 2. The tolerances follow a Gaussian or a uniform distribution. In order to ensure a sufficient sealing gap, the standard deviation of the gap p should not exceed the limit of 0.1 mm. Hence, the functionally relevant key characteristic of the assembly is the vertical displacement of the inner hinge plate with respect to the outer hinge plate, which directly influences the width p of the gap for the gasket.

Table 2. Dimensions and corresponding tolerance specifications.

Dimension	Nominal (in mm)	Tolerance T_i (in mm)	Distribution
M_1	7.5	± 0.05	Gaussian ($\pm 3\sigma$)
M_2	5.1	± 0.1	Uniform
M_3	17.5	± 0.05	Gaussian ($\pm 3\sigma$)
M_4	5.1	± 0.1	Uniform
M_5	5.05	± 0.1	Gaussian ($\pm 3\sigma$)
M_6	12.5	± 0.05	Gaussian ($\pm 3\sigma$)
M_7	5.1	± 0.1	Uniform

First, the mathematical relation between the functional key characteristic p and the deviating dimensions M_1 to M_7 is required. The overconstraint design of the assembly already comes into play here: Due to the design of the hinge and the resulting interactions of the deviations, two assembly scenarios are possible (see Figure 3). The assembly scenarios differ in how the plates' surfaces are in contact. In scenario #1, the plates are in contact in the upper slot, whereas in scenario #2, there is contact in the lower slot. Thus, the distances A_1 (upper slot) and A_2 (lower slot) can be determined from M_1 to M_7.

$$A_1 = \left(M_6 + \frac{M_7}{2}\right) - \left(M_3 - \frac{M_4}{2}\right) \tag{2}$$

$$A_2 = M_5 - \left(M_1 - \frac{M_2}{2}\right). \tag{3}$$

If $A_1 \leq A_2$ the hinge will be assembled according to scenario #1. In this case, the resulting key characteristic p is the determined value of the distance A_1 and vice versa. This results in a piecewise-defined mathematical function of the gap p.

$$p = \begin{cases} A_1 \; if \; A_1 \leq A_2 \\ A_2 \; if \; A_1 > A_2 \end{cases} \tag{4}$$

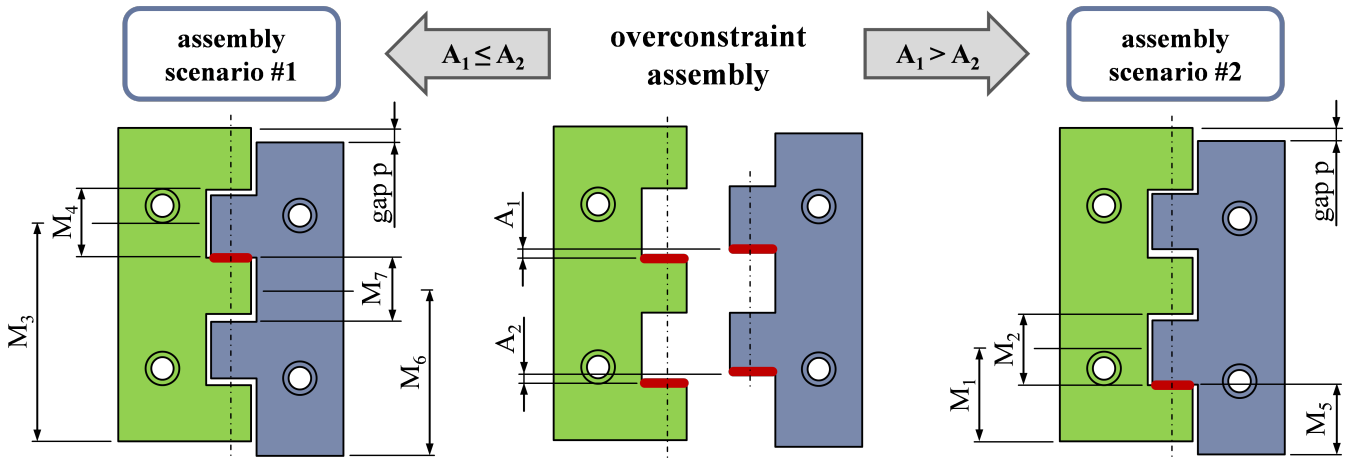


Figure 3. Two assembly scenarios occurring due to the overconstraint design of the door hinge assembly.

5. Methods: Detailing the Approach to Speeding Up Tolerance Simulations

The core of this work is the execution of the statistical tolerance analysis and the statistical tolerance synthesis considering different programming and architecture approaches as well as a varying sample size for the underlying Monte Carlo simulation. First, the procedure (Section 5.1) as well as the implementation (Section 5.2) of the statistical tolerance analysis are presented. The following sections focus on the procedure (Section 5.3) and the implementation of the statistical tolerance synthesis using numerical optimiza-

tion (Section 5.4). This is followed by details on the implementations and hardware used (Sections 5.5 and 5.6).

5.1. Procedure for the Statistical Tolerance Analysis

The application of Monte Carlo simulation [29] in statistical tolerance analysis carries out a virtual reproduction of the manufacturing, assembly and inspection of a large number of products. Figure 4 illustrates these three steps. First, a defined number n of variants of each individual part are generated virtually, which differ from each other in various properties (e.g., dimensions), which are subject to tolerances [30]. The value of this deviation is determined taking into account the associated probability distribution of each tolerance. This step reproduces virtually the production of n single parts. In the next step, the single parts generated are assembled into n final products. This corresponds to the n-fold virtual assembly of the final product and is based on the "pulling without adding back" of the individual parts according to Bernoulli's urn model [31]. Finally, the relevant key characteristic is determined using the vector-chain and documented for each of the n virtually assembled products. This corresponds to the inspection of the products after their assembly.

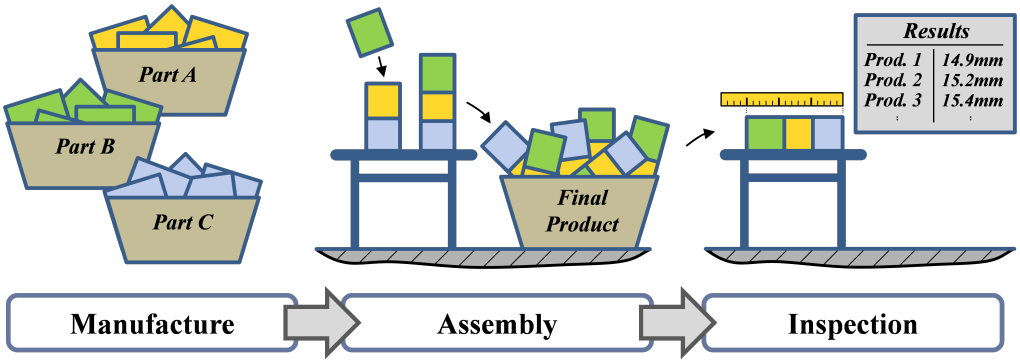


Figure 4. Virtual reproduction of the manufacturing, assembly and inspection of each part by statistical tolerance analysis using Monte Carlo simulation.

5.2. Implementation of the Monte Carlo Simulation

The sample generation and simulation are the most runtime-intensive steps of the tolerance optimization process, as a sufficient number of samples must first be generated and subsequently processed in the simulation step in order to obtain a trustworthy estimate for each tolerance configuration. Accelerating either of these steps would therefore provide the largest reduction of runtime of the overall process. Therefore the two steps 'sample generation' and 'simulation' have been implemented using the hardware architectures of the CPU and GPU of present-day desktop computers. To investigate the influence of different memory storage layouts and floating-point number configurations on the runtime, different implementations have been included in this study as well as an implementation in MATLAB (see Tables 3 and 4 for an overview of the CPU and GPU implementations).

Table 3. Implementations of statistical tolerance analysis using a CPU.

Label	Backend	Storage Layout	Float Format
NumPy Float32 C-Order	Python+NumPy	Row-major	Single-precision *
NumPy Float64 C-Order	Python+NumPy	Row-major	Double-precision
NumPy Float32 F-Order	Python+NumPy	Column-major	Single-precision *
NumPy Float64 F-Order	Python+NumPy	Column-major	Double-precision
MATLAB Float32	MATLAB	Column-major	Single-precision
MATLAB Float64	MATLAB	Column-major	Double-precision

* Since only double precision was available for the sample generation process in NumPy v.1.18.4, the double precision sample was cast to a single precision float after creation.

Table 4. Implementation of the statistical tolerance analysis implementations using a GPU.

Label	Backend	Storage Layout	Float Format
CuPy Float32 C-Order	Python+CuPy	Row-major	Single-precision
CuPy Float64 C-Order	Python+CuPy	Row-major	Double-precision
CuPy Float32 F-Order	Python+CuPy	Column-major	Single-precision
CuPy Float64 F-Order	Python+CuPy	Column-major	Double-precision
MATLAB Float32 GPU	MATLAB	Column-major	Single-precision
MATLAB Float64 GPU	MATLAB	Column-major	Double-precision

The generation of samples and the simulation is accomplished either on the CPU or with the use of the graphics unit. In more detail, the NumPy implementations use *np.random.normal* to generate a normal distribution pseudorandom sample and the *np.random.uniform* function for a uniform distribution. The simulation step was realized with NumPy arrays for fast vectorized array operations. The CuPy implementations use the same code as the NumPy implementations, but with CuPy as a drop-in replacement for NumPy to perform the sample generations and simulations on the GPU. For the MATLAB implementations, mlfg6331_64, *RandStream* is used to generate the samples using the CPU, but mrg32k3a, *parallel.gpu.RandStream* for the GPU implementation. The simulation is implemented using MATLAB matrix and vector operations. The Python program used is available on GitHub [32], accessed on 28 April 2021 (see page 17).

5.3. Procedure for Statistical Tolerance Synthesis

The purpose of statistical tolerance synthesis is to allocate the maximum tolerance of the functional key characteristic to the characteristics of the parts of the product [9]. It is thus the inverse or inversion of statistical tolerance analysis [9]. The goal of statistical tolerance synthesis is to identify the best compromise between (usually two) divergent requirements for a product and thus for tolerating the individual part. In the case of a cost-driven statical tolerance synthesis, this compromise is the tolerance that is associated with the lowest manufacturing costs, but at the same time ensures that a defined rejection rate of the final product is not exceeded [9]. If this compromise is identified by means of numerical optimization, the phrase statistical tolerance optimization has become established. Pseudocode for the implementation of a cost-driven statistical tolerance optimization algorithm is provided in Figure 5. Let T be a selected set of tolerances, which is chosen by a global optimization solver. The cost function is denoted by C. The total production cost is the sum of the production costs of each tolerance $C(T)$ on the basis of the tolerance cost model.

$$C(T) = C_{fix} + C_{var} = C_{fix} + \frac{C_{ind} \cdot e^{-m \cdot T}}{T^k} \tag{5}$$

and is usually specified by the coefficients C_{ind}, m, and k to approximate the dependencies between the costs and tolerances [33,34]. The manufacturing costs consist of the following components: The variable costs C_{var} and the fixed manufacturing costs C_{fix} for each tolerance T. Let S be the system model of the corresponding tolerance problem and D be the probability distributions of the considered tolerances T.

Algorithm 1: Tolerance optimization pseudocode

```
Result: Cost-optimal tolerance setting T_opt
Initialize C, S, D, minimal_cost = ∞
while minimal_cost not converged do
    select T;
    current_costs = C(T);
    if current_costs < minimal_cost then
        sample generation based on selected T and given D;
        input generated samples in S;
        if output of S holds specified boundary conditions
         then
            minimal_cost ← current_cost;
            T_opt ← T
        end
    end
end
```

Figure 5. Pseudocode of the implementation of the cost-driven statistical tolerance synthesis.

5.4. Implementation of Statistical Tolerance Synthesis Using Numerical Optimization

To investigate the impact on runtime, both open source CPU (NumPy) and GPU (CuPy) implementations of the statistical tolerance analysis have been integrated into the statistical tolerance synthesis process to calculate the cost-optimal tolerances using the differential evolutionary algorithm of [35]. The chosen parameter settings of the algorithm are listed in Table 5. For the CPU implementation with NumPy, the population is distributed among all CPU cores. For the GPU implementation with CuPy, a single Python process is sequentially executed.

Table 5. Settings for the differential evolutionary algorithm.

Parameter	Value
strategy	best1bin
bounds	$[0.0, 0.7] \times 7$
popsize	25
tol	0.01
mutation	(0.5, 1)
recombination	0.5
seed	1992
init	latinhypercube
workers	*,**

** workers = −1: Distributes the populations among all available CPU cores for parallel computation, * workers = 1: A single Python process is spawned.

The cost-driven tolerance optimization has been applied to the overconstraint door hinge assembly defined in Section 4, leading to the following optimization problem:

$$\min C(X) \tag{6}$$

under given constraints:

$$\sigma_p(X) \leq 0.1 \tag{7}$$

with the bounds of:

$$X_{(T_1-T_7)} \leq 0.7 \tag{8}$$

$$X_{(T_1-T_7)} > 0 \tag{9}$$

To ensure the functionality of the assembly, the requirements for p are defined as an inequality boundary condition. It demands that the standard deviation σ_p must be less

than or equal to 0.1 mm in order to guarantee a sufficient tightness. The tolerance-cost models are taken from [28]. Their coefficients are listed in Table 6.

Table 6. Parameters of tolerance-cost models.

Parameter	Value
$C_{fix,1-7}$	0 EUR
m_{1-7}	0
k_{1-7}	1
$C_{\mathrm{ind},1}$	1 EUR × mm
$C_{\mathrm{ind},2}$	9 EUR × mm
$C_{\mathrm{ind},3}$	5 EUR × mm
$C_{\mathrm{ind},4}$	15 EUR × mm
$C_{\mathrm{ind},5}$	2 EUR × mm
$C_{\mathrm{ind},6}$	11 EUR × mm
$C_{\mathrm{ind},7}$	18 EUR × mm

5.5. Runtime Measurement Notes

The Python built-in time function *time.time* is used to determine the runtime of each Python implementation. For the MATLAB implementations, the integrated stopwatch timer functions *tic toc* were used.

5.6. Software Details

The programming languages MATLAB 2018b and Python 3.8.2 (including the Python libraries NumPy 1.18.4, CuPy 8.1.0, and SciPy 1.5.4) were used.

5.7. Platform Details

The runs were conducted on a Windows 10 workstation with an AMD Ryzen 9 3900 12-core processor (3.8 GHz clock speed) equipped with 32 GB of DDR4 RAM (3200 MHz clock speed) and an Nvidia GeForce RTX 2060 graphics unit with 6.0 GB of VRAM that supports CUDA up to version 7.5.

6. Results and Discussion

In this section, we present and discuss the results of the study. First, we review the results of the statistical tolerance analysis implementations. Next, we discuss the results of the Statistical Tolerance Synthesis and determine, based on an appropriate sample size, which of the two best open source implementations is more appropriate for the underlying tolerance problem.

6.1. Statistical Tolerance Analysis

The statistical tolerance analysis was performed on the basis of the implementations described in Section 5.2. The results are presented for the CPU in Section 6.1.1 and the GPU in Section 6.1.2.

6.1.1. CPU Implementations

Figure 6 and Table 7 present the average runtime in seconds with its standard error of the mean of 25 executions of the steps of generating the samples and then simulating them, using CPU architecture.

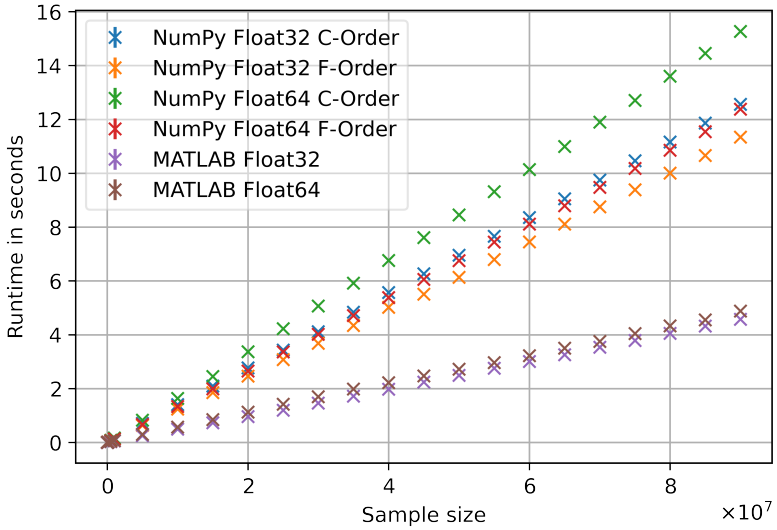


Figure 6. Statistical tolerance analysis using CPU: Mean runtime results with error bars.

Table 7. Statistical tolerance analysis using CPU: Mean runtime results in seconds with standard error of the mean.

Sample Size	NumPy Float32 C-Ordering	NumPy Float32 F-Ordering	NumPy Float64 C-Ordering	NumPy Float64 F-Ordering	MATLAB Float32	MATLAB Float64
1.0×10^4	0.001241 ± 0.000065	0.001181 ± 0.000049	0.001201 ± 0.000050	0.001181 ± 0.000052	0.003724 ± 0.000925	0.004503 ± 0.002241
5.0×10^4	0.005965 ± 0.000040	0.005705 ± 0.000050	0.006506 ± 0.000050	0.006325 ± 0.000064	0.004189 ± 0.000282	0.004814 ± 0.000486
1.0×10^5	0.012391 ± 0.000066	0.012491 ± 0.000061	0.013311 ± 0.000050	0.012851 ± 0.000048	0.006743 ± 0.000183	0.007997 ± 0.000464
5.0×10^5	0.069539 ± 0.000225	0.063134 ± 0.000286	0.083852 ± 0.000142	0.066497 ± 0.000124	0.031448 ± 0.000272	0.041315 ± 0.000856
1.0×10^6	0.142923 ± 0.000362	0.123606 ± 0.000173	0.168165 ± 0.000334	0.134275 ± 0.000332	0.058773 ± 0.000715	0.070176 ± 0.002249
5.0×10^6	0.712692 ± 0.004422	0.633685 ± 0.003192	0.832475 ± 0.001705	0.667433 ± 0.002543	0.254988 ± 0.001123	0.297843 ± 0.006478
1.0×10^7	1.404747 ± 0.002412	1.242067 ± 0.001219	1.637727 ± 0.002124	1.340692 ± 0.005562	0.493843 ± 0.002063	0.572002 ± 0.004212
1.5×10^7	2.099064 ± 0.002512	1.852131 ± 0.002354	2.453188 ± 0.002672	1.993793 ± 0.005053	0.732623 ± 0.002102	0.851755 ± 0.007814
2.0×10^7	2.772782 ± 0.006491	2.466719 ± 0.002730	3.368234 ± 0.004239	2.650779 ± 0.005898	0.959620 ± 0.001961	1.129460 ± 0.005319
2.5×10^7	3.436272 ± 0.003951	3.075642 ± 0.003229	4.223949 ± 0.005337	3.364491 ± 0.010650	1.199648 ± 0.001999	1.418932 ± 0.005626
3.0×10^7	4.117858 ± 0.004286	3.686407 ± 0.003401	5.073099 ± 0.006942	4.020194 ± 0.005144	1.461204 ± 0.003205	1.701652 ± 0.005182
3.5×10^7	4.842020 ± 0.005639	4.344172 ± 0.013985	5.916764 ± 0.005374	4.714091 ± 0.007223	1.719168 ± 0.002966	1.984968 ± 0.005454
4.0×10^7	5.573128 ± 0.004716	5.021194 ± 0.041395	6.765172 ± 0.006502	5.380363 ± 0.006298	1.981424 ± 0.003639	2.222212 ± 0.009649
4.5×10^7	6.269647 ± 0.005735	5.510913 ± 0.004690	7.609078 ± 0.007418	6.060067 ± 0.006725	2.243420 ± 0.003477	2.476140 ± 0.010313
5.0×10^7	6.959560 ± 0.006334	6.134791 ± 0.004377	8.455985 ± 0.008600	6.756005 ± 0.005361	2.491044 ± 0.004360	2.723836 ± 0.007641
5.5×10^7	7.660562 ± 0.005350	6.798842 ± 0.003901	9.315744 ± 0.010400	7.446538 ± 0.009688	2.755600 ± 0.007505	2.968508 ± 0.009055
6.0×10^7	8.359663 ± 0.008355	7.454204 ± 0.005153	10.138631 ± 0.010530	8.113391 ± 0.006551	3.006004 ± 0.004333	3.227844 ± 0.006248
6.5×10^7	9.055540 ± 0.006843	8.113711 ± 0.005656	10.995247 ± 0.007439	8.796638 ± 0.006242	3.260588 ± 0.004717	3.503080 ± 0.009699
7.0×10^7	9.749416 ± 0.005656	8.751739 ± 0.007032	11.900305 ± 0.018304	9.474140 ± 0.008674	3.538272 ± 0.004528	3.756784 ± 0.008264
7.5×10^7	10.464471 ± 0.006690	9.388546 ± 0.006888	12.709061 ± 0.006853	10.178886 ± 0.009369	3.792104 ± 0.004796	4.043528 ± 0.012855
8.0×10^7	11.163511 ± 0.006766	10.009300 ± 0.008521	13.604569 ± 0.009452	10.858189 ± 0.010415	4.052580 ± 0.007483	4.329744 ± 0.012126
8.5×10^7	11.865054 ± 0.008723	10.657933 ± 0.008216	14.453738 ± 0.010055	11.546921 ± 0.007045	4.323880 ± 0.007407	4.556864 ± 0.010179
9.0×10^7	12.560572 ± 0.010832	11.341762 ± 0.015154	15.269399 ± 0.025013	12.380537 ± 0.030369	4.578620 ± 0.009665	4.883124 ± 0.014633

It can be seen that both CPU MATLAB implementations perform better than the Python open source implementations with NumPy for nearly all sample sizes. This is due to MATLAB's default being able to work with all processor cores of the workstation, while the Python implementations are limited to one CPU core by the 'Global Interpreter Lock' (GIL) [36]. However, in Section 6.2.1, the GIL constraint is bypassed using the multiprocessing feature of the numerical solver in the tolerance synthesis. Multiprocessing is the creation of several new Python processes (each with its own GIL), in which each process takes a portion of the computation. A difference in runtime between the use of C-Order and F-Order as well as single precision and double precision can also be identified. For all CPU implementations, the use of F-Order as the NumPy array storage layout tends to be more advantageous than C-Order. On the other hand, runtime gains can also be achieved using single precision instead of double precision floating-point numbers for the CPU implementation. This, however, involves a loss of precision in the estimation.

6.1.2. GPU Implementations

Figure 7 and Table 8 show the average runtime in seconds with the standard error of the mean of 25 executions of the statistical tolerance analysis, subject to the sample size using the GPU architecture.

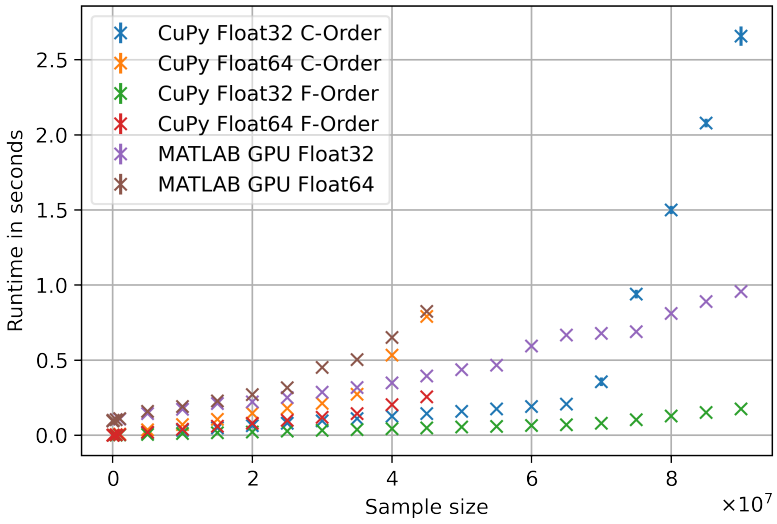


Figure 7. Statistical tolerance analysis using GPU: Mean runtime results with error bars.

Table 8. Statistical tolerance analysis using GPU: Mean runtime results in seconds with standard error of the mean.

Sample Size	CuPy Float32 C-Order	CuPy Float64 C-Order	CuPy Float32 F-Order	CuPy Float64 F-Order	MATLAB GPU Float32	MATLAB GPU Float64
1.0×10^4	0.000520 ± 0.000143	0.000629 ± 0.000140	0.000440 ± 0.000101	0.000440 ± 0.000098	0.096187 ± 0.000887	0.094625 ± 0.000577
5.0×10^4	0.000541 ± 0.000028	0.000721 ± 0.000051	0.000520 ± 0.000020	0.000600 ± 0.000041	0.098829 ± 0.001141	0.098982 ± 0.000765
1.0×10^5	0.000681 ± 0.000057	0.001041 ± 0.000028	0.000480 ± 0.000020	0.000641 ± 0.000046	0.096635 ± 0.000422	0.098395 ± 0.000749
5.0×10^5	0.001982 ± 0.000074	0.004324 ± 0.000091	0.000721 ± 0.000058	0.002867 ± 0.000067	0.104505 ± 0.001328	0.103663 ± 0.000813
1.0×10^6	0.003633 ± 0.000043	0.007451 ± 0.000065	0.001381 ± 0.000044	0.004314 ± 0.000058	0.106821 ± 0.000827	0.110193 ± 0.000804
5.0×10^6	0.016455 ± 0.000098	0.035831 ± 0.000150	0.005782 ± 0.000071	0.020137 ± 0.000098	0.144228 ± 0.002474	0.159218 ± 0.001141
1.0×10^7	0.032322 ± 0.000123	0.071453 ± 0.000602	0.011206 ± 0.000103	0.039910 ± 0.000108	0.172592 ± 0.001675	0.190616 ± 0.001785
1.5×10^7	0.048098 ± 0.000113	0.106091 ± 0.000126	0.016413 ± 0.000096	0.059422 ± 0.000080	0.210516 ± 0.004146	0.227843 ± 0.001579
2.0×10^7	0.063791 ± 0.000109	0.146295 ± 0.001609	0.021844 ± 0.000129	0.079300 ± 0.000127	0.222762 ± 0.001016	0.270121 ± 0.002696
2.5×10^7	0.079521 ± 0.000093	0.178616 ± 0.001031	0.027177 ± 0.000078	0.099269 ± 0.000239	0.250182 ± 0.000624	0.315998 ± 0.002046
3.0×10^7	0.095980 ± 0.000200	0.212295 ± 0.000292	0.032448 ± 0.000081	0.118662 ± 0.000272	0.286288 ± 0.000697	0.451989 ± 0.004776
3.5×10^7	0.111210 ± 0.000146	0.273510 ± 0.005553	0.037825 ± 0.000101	0.144299 ± 0.001217	0.317789 ± 0.000977	0.503447 ± 0.001281
4.0×10^7	0.127250 ± 0.000175	0.533831 ± 0.005671	0.043096 ± 0.000125	0.203151 ± 0.003404	0.349101 ± 0.002187	0.650528 ± 0.004721
4.5×10^7	0.143477 ± 0.000254	0.791146 ± 0.006204	0.048402 ± 0.000124	0.255648 ± 0.000926	0.394972 ± 0.001001	0.824722 ± 0.004174
5.0×10^7	0.158769 ± 0.000183	NaN	0.054847 ± 0.000372	NaN	0.436220 ± 0.001380	NaN
5.5×10^7	0.174816 ± 0.000258	NaN	0.059154 ± 0.000156	NaN	0.465871 ± 0.001071	NaN
6.0×10^7	0.191430 ± 0.000555	NaN	0.064545 ± 0.000144	NaN	0.594289 ± 0.004932	NaN
6.5×10^7	0.207075 ± 0.000245	NaN	0.070035 ± 0.000147	NaN	0.667256 ± 0.006966	NaN
7.0×10^7	0.355644 ± 0.028266	NaN	0.079949 ± 0.001041	NaN	0.678562 ± 0.003035	NaN
7.5×10^7	0.940129 ± 0.028603	NaN	0.103767 ± 0.001036	NaN	0.689129 ± 0.001856	NaN
8.0×10^7	1.499938 ± 0.027674	NaN	0.128240 ± 0.001316	NaN	0.810690 ± 0.002744	NaN
8.5×10^7	2.078780 ± 0.028271	NaN	0.151778 ± 0.001023	NaN	0.890743 ± 0.002536	NaN
9.0×10^7	2.658023 ± 0.064572	NaN	0.175046 ± 0.000983	NaN	0.957543 ± 0.003513	NaN

Since, for the GPU implementations, the generation and processing of the random numbers does not take place in the workstation's memory, but in the typically smaller video memory of the GPU, some distinctive results can be observed compared to the CPU implementations: (i) The video memory of the GPU in use reaches its capacity limit for double precision floating-point numbers for sample sizes larger than 4.5×10^7 (NaN results in Table 8). With single precision, which only requires half as much memory as double precision floating-point numbers, the memory limit is reached at sample sizes greater than 9.0×10^7. (ii) The choice between single and double precision also has a greater influence on the runtime, which is significantly lower in the case of single precision compared to using double precision. (iii) As in the case of CPU implementation, the use of the F-order storage layout seems to be more advantageous for the implementations. On the one hand, the run-

time is consistently lower, and on the other, the GPU implementations with C-order storage layout exhibit runtime spikes for higher sample sizes. We see the cause in the less efficient memory access operations compared to the implementations using the F-Order storage layout. (iv) In contrast to the CPU implementations, the GPU Python implementations perform significantly better than the GPU implementations with MATLAB.

A comparison of the CPU and GPU results shows that the statistical tolerance analysis that uses the GPU scales significantly better with an increase in sample size (apart from memory related effects). The most effective implementation (CuPy Float32 F-Order) is able to perform statistical tolerance analysis in real time up to a sample size of one million, which is sufficient to analyze even complex tolerance problems.

6.2. Statistical Tolerance Synthesis

In the following (Section 6.2.1), the fastest Python CPU and GPU implementations are incorporated into the statistical tolerance synthesis (described in Section 5.3) to determine how each implementation performs given various sample sizes. For this purpose, the average runtime of the differential evolution step is used as a metric for comparing the CPU and GPU implementations in the statistical tolerance synthesis. This will enable making a statement about which implementation achieves better runtime results, independently of how fast the solver converges to a solution in each individual case. In Section 6.2.2, we then analyze what sample size is actually required in order to obtain a reasonably good estimate from the statistical tolerance synthesis.

6.2.1. CPU vs. GPU Implementations

Figure 8 and Table 9 present the average runtime in seconds with its standard error of the mean of 10 executions of the differential evolution for each configured implementation, with the parameters shown in Table 5.

The results in Table 9 and Figure 8 show that the computational cost of the statistical tolerance synthesis increases linearly with the sample size, analogous to the results of the statistical tolerance analysis. CPU implementation (which, in the context of a statistical tolerance synthesis, is able to use all CPU cores) provides faster computation of the differential evolution for sample sizes smaller than 50,000. However for larger sample sizes, GPU implementation is faster. This indicates that reaching a certain complexity, the calculation using a GPU architecture is faster than a computation using only the CPU. Given a sample size of 10,000, which is a common standard in industrial applications, it therefore is possible to perform a faster tolerance synthesis based on CPU implementation due to a lower runtime for the differential evolution. To double check this assumed standard, we determine in Section 6.2.2 how large the sample size actually needs to be to obtain a reliable estimate for the given tolerance model.

Table 9. Average runtime of the differential evolution step in seconds with standard error vs. mean.

Sample Size	NumPy Float32 F-Order **	CuPy Float32 F-Order *
1.0×10^4	0.028884 ± 0.000035	0.116859 ± 0.000152
5.0×10^4	0.170581 ± 0.000301	0.169545 ± 0.000177
1.0×10^5	0.328820 ± 0.000453	0.247073 ± 0.000258
5.0×10^5	2.208728 ± 0.003886	1.466514 ± 0.001694
1.0×10^6	4.272736 ± 0.001130	3.268489 ± 0.000054
2.0×10^6	8.779391 ± 0.005388	6.490047 ± 0.000075
4.0×10^6	17.518838 ± 0.002645	12.932327 ± 0.000107
6.0×10^6	26.400617 ± 0.005473	19.379022 ± 0.000118
8.0×10^6	35.812385 ± 0.023610	25.812515 ± 0.007956
1.0×10^7	45.290421 ± 0.028137	34.823996 ± 0.129419

** workers = -1: Distributes the populations among all available CPU cores for parallel computation, * workers = 1: A single Python process is spawned.

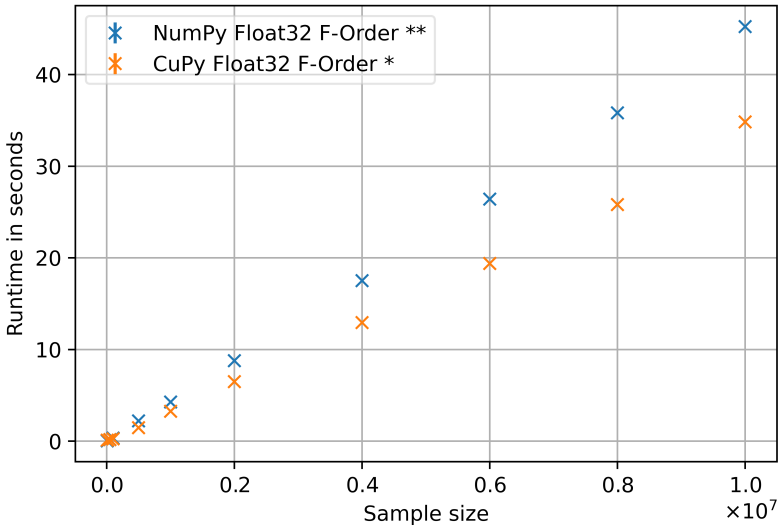
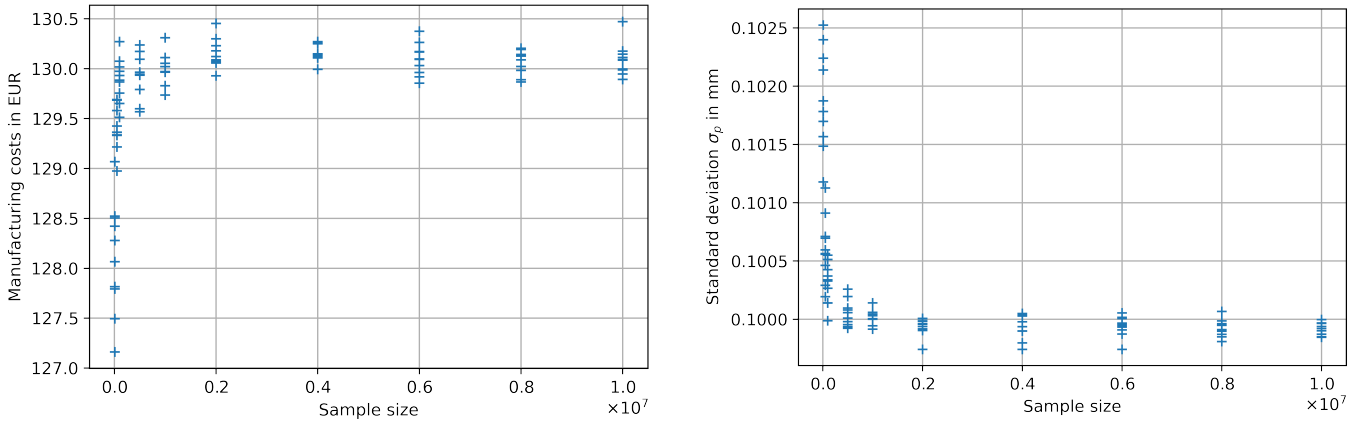


Figure 8. Performing Statistical Tolerance Synthesis on CPU (NumPy) vs. GPU (CuPy): Average runtime of the differential evolution step with error bars. ** workers = −1: Distributes the populations among all available CPU cores for parallel computation, * workers = 1: A single Python process is spawned.

6.2.2. Sample Size Required for a Reliable Estimate

Figure 9a displays the manufacturing costs calculated by the statistical tolerance synthesis (with CuPy Float 32 F-Order) vs. the sample size, from Section 6.2.1. Figure 9b displays, on the basis of a sample size of 1.0×10^7, the corresponding standard deviation σ_p, in order to determine if a sample size of 10,000 is sufficient for the tolerance optimization problem.



(**a**) (**b**)

Figure 9. Statistical tolerance synthesis using GPU (CuPy Float 32 F-Order): (**a**) Manufacturing costs vs. sample size for 10 runs, and (**b**) standard deviation σ_p (calculated with 1.0×10^7 sample size) vs. sample size.

The spread of the results of the statistical tolerance syntheses in Figure 9a,b show that for the given tolerance problem, a sample size of at least 50,000 samples should be used to satisfy the boundary condition ($\sigma_p < 0.1$ mm) with sufficient confidence. Smaller sample sizes tend to result in manufacturing costs, and thus tolerances, that do not comply with the required boundary conditions of the statistical tolerance synthesis. On this basis, computation using the GPU is preferable due to its lower runtime for the differential evolution. Figure 10 illustrates the reduction of manufacturing costs during optimization. The optimization is carried out using the parameter settings shown in Table 5 and a sample size of 100,000 with the CuPy single precision Float F-Order implementation. Hence, a cost-

optimal tolerance specification is obtained after 55 steps of the evolution (approximately 13.5 s).

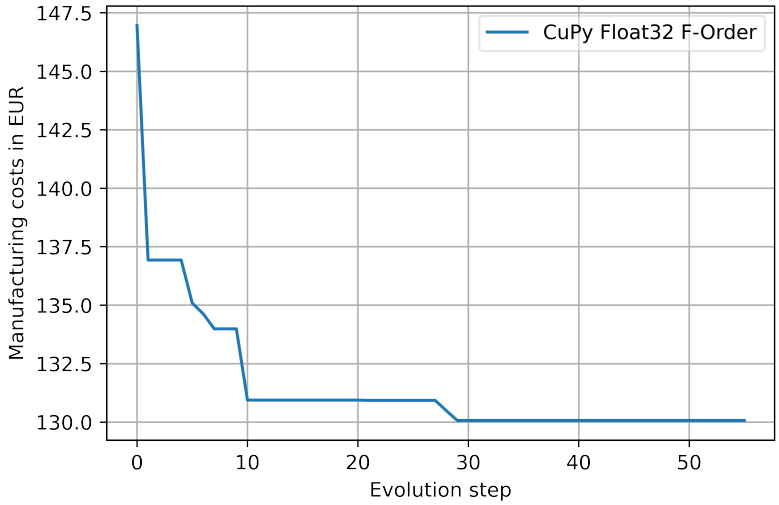


Figure 10. Manufacturing costs vs. differential evolution steps with 100,000 samples.

6.3. Additional Notes on Accuracy and Comparability of the Implementations

Although the NumPy and CuPy implementations are based on the same codebase (with the exception of imports), the question arises whether the different implementations provide the same level of accuracy. It might have been that the performance gain of the GPU implementation was partially obtained at the expense of the accuracy of the estimation, for example, by using a lower quality random sample generation process. A fair comparison between the Python and the MATLAB implementation is considerably more difficult than a comparison within only the Python implementations, due to the different programming languages, their quirks, and individual restrictions. For this reason, the claim in this paper is by no means meant to be a Python vs. MATLAB argument. Rather, the claim is that one can accomplish a statistical tolerance analysis and a statistical tolerance synthesis with open source software and methods.

6.4. Current Drawbacks of the Method and Future Research Demands

The tolerance optimization methodology using sampling methods with stochastic solution algorithms is approaching the limits of reasonable computation time according to [37]. With our approach to compute the Monte Carlo simulation on the GPU, we show a possible way to reduce the computation time that is expected to further increase in the future due to the growing complexity of future tolerance synthesis models [37]. However, several aspects require improvement and further research.

- The mathematical model used does not yet take into account geometric deviations of the manufacturing components. The consideration of these however is warranted because a majority of companies consider geometric deviations in the specification of tolerances [8]. The implementation in this paper is not limited only to deviations in size. Vector-chains with geometric tolerances can also be analyzed.
- The vector-chain needs an extension to fully account for 3D effects. This is due to the assumption that most problems can be traced back to 1D or 2D is usually overly optimistic [37]. However, the demonstrated approach using a derivative free optimization algorithm already lays a foundation to implement this in the future, since the problem does not have to be oversimplified [37]. The prerequisite is that a closed vector-chain is available.

- The approach has been tested on an Nvidia GPU. However, the CuPy library also has experimental support for RadeonOpenCompute (ROCm) [16], the non-proprietary open-source alternative to CUDA from AMD. Further work could investigate how applicable this is.
- Finally, we see a need for research concerning the support of multiple GPUs. This would allow the statistical tolerance synthesis to be distributed across several graphics units analogous to the already existing CPU implementation and thus to be accelerated. For this purpose, the applied solver or alternative approaches could be investigated.

7. Conclusions

With the approaches presented in this study, it is possible to reduce the runtime of vector-chain-based statistical tolerance analysis to real time and thus to drastically speed up statistical tolerance optimization. From the engineer's point of view, it can be concluded that the presented implementations of statistical tolerance analysis are capable of real time execution: The calculation times of the statistical tolerance analysis with a sample size of 10,000 were all well below 1 ms and only with a sample size of more than 5 million samples did runtimes (>5 ms) exceed the real time requirement set. It has therefore been proven that statistical tolerance analyses (with a sufficient sample size) could be accelerated to real time in order to establish a continuous flow of information for the implementation of self-awareness and self-adjustment in manufacturing and thus finally promote the implementation of Industry 4.0. Furthermore, the results provide information on how the CPU and GPU architectures could be effectively used in the context of tolerance analysis and tolerance synthesis and which of the considered parameters significantly influence the runtime of the implementations. From this, concrete recommendations for future work could be derived.

The general advice to improve the runtime of statistical tolerance analyses and statistical tolerance synthesis is to use NumPy Arrays for vectorization. The results clarify the significance of an adequate choice of memory layout, especially for implementations that use a GPU. We there recommend testing which memory layout option is more suited for the code used. Another finding is that the choice of architecture depends on the complexity of the considered tolerance problem. We recommend first evaluating how many samples are needed to obtain a reasonable estimate. If a correspondingly powerful GPU is available, it is highly recommended to carry out the calculations on a GPU instead of a CPU once a certain computational complexity has been reached. This is due to the better parallel processing options and the typically higher core count of a GPU. However, if the tolerance problem is not computationally complex, the additional overhead of copying the data into video memory may exceed the benefit generated by the faster computation on the GPU. In this case, an implementation that uses a CPU architecture can be more advantageous. If the calculation is carried out using the GPU, we recommend using single instead of double precision floating-point numbers.

We hope that this work will help further improve existing approaches to statistical tolerance analysis and statistical tolerance synthesis with respect to the ongoing transition to Industry 4.0 and provide a practical guide as well as an entry point for future tolerance engineering research that focuses more on tolerance engineering with heterogeneous computer architectures and parallel computing. Finally, efficient open source alternatives to the predominantly deployed software packages are available to speed up computer simulations to real time.

Author Contributions: Conceptualization, P.G. and M.S.J.W.; methodology, P.G. and M.S.J.W.; software, P.G. and M.S.J.W.; validation, P.G. and M.S.J.W.; data curation, P.G. and M.S.J.W.; writing—original draft preparation, M.S.J.W. and P.G.; writing—review and editing, M.S.J.W. and P.G.; visualization, M.S.J.W. and P.G.; supervision, M.S.J.W.; project administration, M.S.J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The Python implementation is available in GitHub in [32], accessed on 28 April 2021. The MATLAB implementation is available on request from the corresponding author.

Acknowledgments: We are very grateful to Rolands Kalvāns who provided valuable advice about writing the paper as well as hands-on programming advice in Python.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Harrison, R. Introduction To Monte Carlo Simulation. *AIP Conf. Proc.* **2010**, *1204*, 17–21. [CrossRef] [PubMed]
2. Cohen, Y.; Faccio, M.; Pilati, F.; Yao, X. Design and management of digital manufacturing and assembly systems in the Industry 4.0 era. *Int. J. Adv. Manufac. Technol.* **2019**, *105*, 3565–3577. [CrossRef]
3. Azzi, A.; Faccio, M.; Persona, A.; Sgarbossa, F. Lot splitting scheduling procedure for makespan reduction and machine capacity increase in a hybrid flow shop with batch production. *Int. J. Adv. Manufac. Technol.* **2011**, *59*, 775–786. [CrossRef]
4. Matheson, E.; Minto, R.; Zampieri, E.G.G.; Faccio, M.; Rosati, G. Human–Robot Collaboration in Manufacturing Applications: A Review. *Robotics* **2019**, *8*, 100. [CrossRef]
5. Boorla, S.M.; Bjarklev, K.; Eifler, T.; Howard, T.; McMahon, C.C.A. Industry 4.0-A challenge for variation simulation tools for mechanical assemblies. *Adv. Comput. Des.* **2019**, *4*, 43–52. [CrossRef]
6. Söderberg, R.; Wärmefjord, K.; Carlson, J.S.; Lindkvist, L. Toward a Digital Twin for real-time geometry assurance in individualized production. *CIRP Ann.* **2017**, *66*, 137–140. [CrossRef]
7. Alerstam, E.; Svensson, T.; Andersson-Engels, S. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *J. Biomed. Opt.* **2008**, *13*, 060504. [CrossRef] [PubMed]
8. Walter, M.S.J.; Klein, C.; Heling, B.; Wartzack, S. Statistical Tolerance Analysis—A Survey on Awareness, Use and Need in German Industry. *Appl. Sci.* **2021**, *11*. [CrossRef]
9. Chase, K.W.; Greenwood, H.W. Design Issues in Mechanical Tolerance Analysis. *Manufac. Rev.* **1988**, *1*, 50–59.
10. Boulle, A.; Kieffer, J. High-performance Python for crystallographic computing. *J. Appl. Crystallograph.* **2019**, *52*, 882–897. [CrossRef]
11. Soklaski, R. "Vectorized" Operations: Optimized Computationson NumPy Arrays. 2020. Available online: https://www.pythonlikeyoumeanit.com/ (accessed on 22 February 2021).
12. Ishigami, T.; Homma, T. An importance quantification technique in uncertainty analysis for computer models. In Proceedings of the First International Symposium on Uncertainty Modeling and Analysis, IEEE, College Park, MD, USA, 3–5 December 1990; pp. 398–403. [CrossRef]
13. Harris, C.R.; Millman, K.J.; van der Walt, S.J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N.J.; et al. Array programming with NumPy. *Nature* **2020**, *585*, 357–362. [CrossRef] [PubMed]
14. Van der Walt, S.; Colbert, S.C.; Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* **2011**, *13*, 22–30. [CrossRef]
15. The NumPy Team. NumPy-Open Source Scientific Computing Library for Python. 2020. Available online: https://numpy.org/doc/stable/ (accessed on 22 February 2021).
16. CuPy Team. CuPy—NumPy-Like API Accelerated with CUDA. 2020. Available online: https://docs-cupy.chainer.org/en/stable/ (accessed on 22 February 2021).
17. Thiyagalingam, J.; Beckmann, O.; Kelly, P. An Exhaustive Evaluation of Row-Major, Column-Major and Morton Layouts for Large Two-Dimensional Arrays. In Proceedings of the 16th International Workshop, LCPC, College Station, TX, USA, 2–4 October 2003.
18. ISO/IEC/IEEE. ISO/IEC/IEEE International Standard—Floating-Point Arithmetic. ISO/IEC 60559:2020(E) IEEE Std 754-2019. 2020; pp. 1–86. Available online: https://ieeexplore.ieee.org/document/9091348 (accessed on 5 May 2021). [CrossRef]
19. Techpowerup. NVIDIA GeForce RTX 2060. 2020. Available online: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060.c3310 (accessed on 22 February 2021).
20. Roy, U.; Liu, C.; Woo, T. Review of dimensioning and tolerancing: Representation and processing. *Comput. Aided Des.* **1991**, *23*, 466–483. [CrossRef]
21. Nigam, S.D.; Turner, J.U. Review of statistical approaches to tolerance analysis. *Comput. Aided Des.* **1995**, *27*, 6–15. [CrossRef]
22. Matala, A. *Sample Size Requierement for Monte Carlo—Simulations Using Latin Hypercube Sampling*; Helsinki University of Technology: Helsinki, Finland, 2008.
23. Stuppy, J. *Methodische und Rechnerunterstützte Toleranzanalyse für Bewegte Technische Systeme*; VDI-Verl: Düsseldorf, Germany, 2011.
24. Chase, K.; Parkinson, A. A survey of research in the application of tolerance analysis to the design of mechanical assemblies. *Res. Eng. Des.* **1991**, *3*, 23–37. [CrossRef]

25. Glancy, C.; Stoddard, J.; Law, M. Automating the Tolerancing Process. In *Dimensioning and Tolerancing Handbook*; McGraw-Hill Education: New York, NY, USA, 1999.
26. Cvetko, R.; Chase, K.; Magleby, S. *New Metrics for Evaluating Monte Carlo Tolerance Analysis of Assemblies*; American Society of Mechanical Engineers: New York, NY, USA, 1998.
27. Altschul, R.; Scholz, F. Case study in statistical tolerancing. *Manufac. Rev.* **1994**, *7*, 52–56.
28. Walter, M.; Spruegel, T.; Ziegler, P.; Wartzack, S. Berücksichtigung von Wechselwirkungen zwischen Abweichungen in der statistischen Toleranzanalyse. *Konstruktion* **2015**, *67*, 88–92.
29. Sobol, I.M. *The Monte Carlo Method*; Little Mathematics Library, Mir Publishers: Moscow, Russia, 1975.
30. Grossman, D.D. *Monte Carlo Simulation of Tolerancing in Discrete Parts Manufacturing and Assembly*; Technical Report; Stanford University: Stanford, CA, USA, 1976.
31. Ostwald, W.; Thaer, C.; Heiberg, J. *Ostwalds Klassiker der exakten Wissenschaften*; Die Elemente von Euklid; Akademische Verlagsgesellschaft m. b. h.: Leipzig, Germany, 1937.
32. Grohmann, P.; Kalvāns, R. Statistical-Tolerance-Analysis-and-Synthesis-with-Python. 2021. Available online: https://github.com/EinmalmitProfis/Statistical-Tolerance-Analysis-and-Synthesis-with-Python (accessed on 28 April 2021).
33. Andolfatto, L.; Thiebaut, F.; Lartigue, C.; Douilly, M. Quality- and cost-driven assembly technique selection and geometrical tolerance allocation for mechanical structure assembly. *J. Manufact. Syst.* **2014**, *33*, 103–115. [CrossRef]
34. Walter, M.S.J. *Toleranzanalyse und Toleranzsynthese Abweichungsbehafteter Mechanismen*; VDI Verlag: Düsseldorf, Germany, 2016.
35. The SciPy Community. Documentation. 2020. Available online: https://docs.scipy.org/doc/ (accessed on 22 February 2021).
36. The Python Software Foundation. Python Documentation. 2020. Available online: https://docs.python.org/3/ (accessed on 22 February 2021).
37. Hallmann, M.; Schleich, B.; Wartzack, S. From tolerance allocation to tolerance-cost optimization: A comprehensive literature review. *Int. J. Adv. Manufac. Technol.* **2020**, *107*, 4859–4912. [CrossRef]