Monte Carlo method for evaluating the uncertainty of roundness measurements on a CMM - a Python implementation

A. Egidi, A. Balsamo, D. Corona

Monte Carlo method for evaluating the uncertainty of roundness measurements on a CMM – a Python implementation

R.T. 14/2021 April 2021

I.N.RI.M. TECHNICAL REPORT

Summary

Abstract

Coordinate measuring machines (CMMs) are leading instruments in production metrology, due to their accuracy, precision, flexibility and level of automation. They are essentially based on sampling of coordinates of points on manufactured pieces, and can virtually measure any tolerance indicated on the drawing. Among different form tolerances, roundness tolerance is often present in technical documentation, when in the presence of centrosymmetric objects. CMMs usually show high quality metrological characteristics, but, as well as every other instruments, they are affected by a certain amount of measuring uncertainties, and it is generally a good practice relying on the Monte Carlo technique to numerically estimate the uncertainty in a measuring process based on the use of a CMM. This report presents a developed model for the evaluation of measurement uncertainty in the process of measuring roundness on a CMM. The described model is used with concrete measuring machine and concrete workpiece.

Introduction

By equipping CMMs with adequate tools, they can be used for measuring any macro-tolerances indicated in the reference technical drawing of a mechanical piece, since only the coordinates of points which are on the physical surface of the measured object can be contacted by the probe; after that the coordinates of interest have been acquired, an independent software implements the associative geometry of the workpiece by means of primitive geometries like lines, circles, planes, etc. From the sampled points, dedicated algorithms extract the sought associative elements. Roundness [1-2] is one of the most investigated parameters, and the deviation of the scanned profile from a perfect circle is defined as *roundness error* or *local roundness deviation* [3]. In coordinate metrology, four are the different strategies to evaluate this error, after sampling the surface profile with an adequate number of points; directly quoting from [4]:

1. **Least Squares Circles (*LSC*)**:
   '*The least squares circle (LSC) is fitted inside the profile such that the sum of the squares of radial coordinates between the circle and profile is minimized as illustrated in Figure 1. The center of the LSC is then used to draw a circumscribed and an inscribed circle on the polar profile and the out-of-roundness value is the radial separation of these two circles. The least squares circle and its center are unique because there is only one that meets the definition and its accuracy depends on the number of points taken.*'

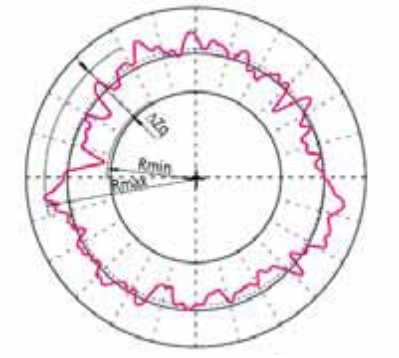

Figure 1: The Least Squares Circle (LSC) method

2. **Minimum Circumscribed Circle (*MCC*)**:
   '*A center is found by drawing a circle that has the smallest possible radius but still contains the polar plot profile in this method as illustrated in Figure 2. An inscribed circle is then drawn inside the profile based on the center of the minimum circumscribed circle. The out-of-roundness value is the difference between the radii of the inscribed and circumscribed circle.*'

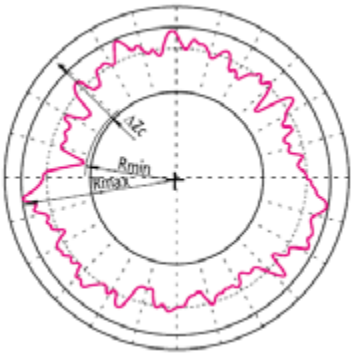

Figure 2: The Maximum Circumscribed Circle (MCC) method

3. **Maximum Inscribed Circle (*MIC*)**:
   '*This method fits the largest possible circle inside the profile figure as shown in Figure 3. The circle can be determined by trial and error with a compass or with a template. After the circle has been drawn, the out-of-roundness value is the maximum distance between the profile and the inscribed circle.*'

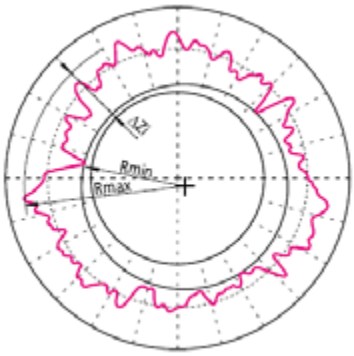

Figure 3: The Maximum Inscribed (MIC) method

4. **Minimum Zone circle (*MZC*)**:
   '*In this method, two circles are used as reference for measuring the roundness error. One circle is drawn outside the roundness profile just as to enclose the whole of it and the other circle is drawn inside the roundness profile so that it just inscribes the profile. The roundness error here is the difference between the radius of the two circles. This method is shown in Figure 4.*'

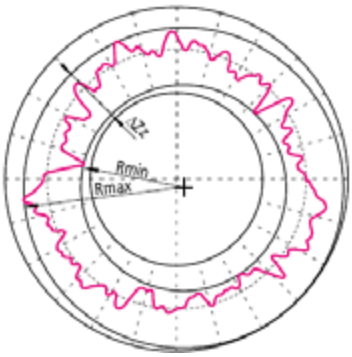

Figure 4: The Minimum Zone Circle (MZC) method

The determination of *LSC*, *MZC*, *MIC*, and *MCC* boundaries for any geometry (*MIC* and *MCC* for closed geometries only) is an optimization problem and can be solved using constrained and/or unconstrained optimization techniques. Known solution methodologies exist for different LSC geometries (Forbes 1989; Shakarji 1998) [5]. Determining MI, MC, and MZ boundaries for different geometries is a challenging task and an active topic of research.

Without going into detail of the mathematical foundations of the reported strategies, it must be stated that every measurement, in general, has its own uncertainty; more remarkably, a measurement result has its dignity only if it is accompanied by a numerical uncertainty. GUM [6] associates to the word "uncertainty" the concept of a parameter related to the result of a measurement characterizing the dispersion of the values that could reasonably be attributed to the measurand. Uncertainty can be described as a probability distribution of the values of the measured quantity, and its evaluation allows to determine the limits of the same measurand with a certain level of confidence; these limits encompass the so-called *expanded uncertainty*. The quantification of the uncertainties in the CMMs world is a really complex task, due to the number of factors and their reciprocal interactions affecting them, but, in this context, it is universally believed that Monte Carlo simulation is the best approach for the quantitative estimation of uncertainty. For this reason, the application of this method is described with respect to a common task in coordinate metrology: measuring roundness by a CMM.

Methodology

The use of Monte Carlo simulations to estimate uncertainties in CMM measurements is nowadays a robust technique, and is still under investigation since the PTB published a pioneering paper on the subject some years ago [7-8]; here, the concept of VCMM (*Virtual Coordinate Measuring Machine*) is defined: it relies on the principle of imitating the measuring strategy and the physical behavior of a CMM, generating clouds of points by means of a virtual probe in contact with the workpiece (Figure 5); the Monte Carlo method is engaged here in randomly sampling the probability density functions that describe the possible scenarios of stochastically interacting parameters that affect the output of a measurement and the evaluation of its uncertainty. The application of this technique to input parameters (in turn allocated in realistic probability density functions) produces a collection of virtual measurements of specific features of the object under examination whose 95 % confidence interval represents the expanded measurement uncertainty. Similar strategy was applied in the development of 'Expert CMM' [9].

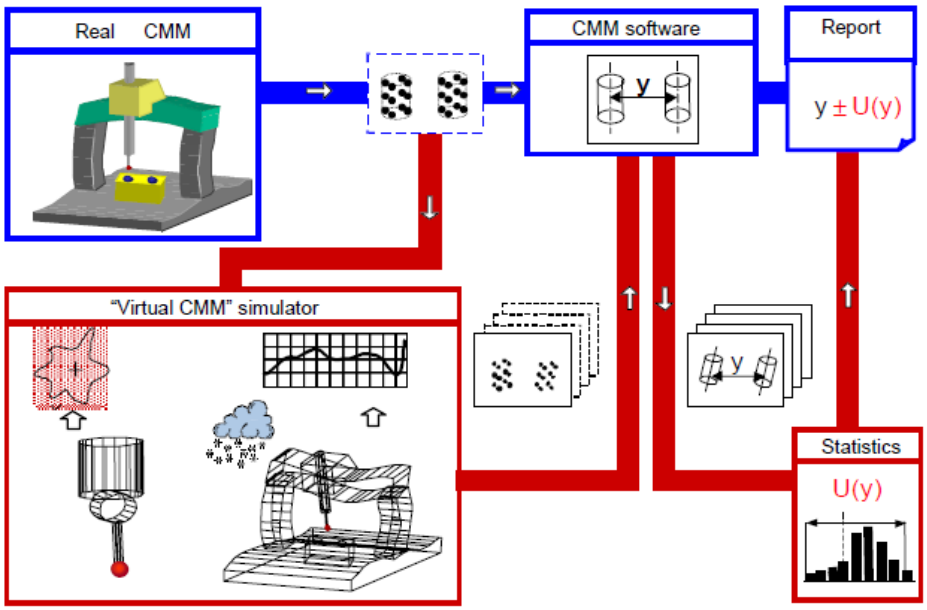

Figure 5: Data flow of the integrated VCMM (extracted from [5])

Two main steps are involved in the simulation strategy: the first relies on determining the uncertainties of the sampled points, the second is aimed to propagate them through the part program driving the machine computations and to define the criterion to obtain the associative geometry. Big efforts need to be carried on to quantify the hardware errors of the CMM, and the literature is rich of such (often expensive and time-consuming) studies [10-13]. In the present work, a simulation model for evaluating the uncertainties in measuring roundness was developed, based on the repeatability of input quantities (*x*, *y* coordinates of the sampled points). For the sake of simplicity, the uncertainty of coordinate points is assumed expressible only by one contribution, in order to define suitable probability density functions to be taken into account in the simulations: the statistical variation of the same sampled points over the sensed surface; the other contributions (like the Form measurement error - ***RONt*** *according to ISO 12181* - as reported in the CMM datasheet, temperature, etc.) are compensated by the CMM software during the probing process.

Shape errors, in combination with the number and position of the measured points, embody the primary factors for measurement uncertainties; the inputs in the model defined in such a way contain the CMM hardware errors interacting with the environment. This study is focused on a specific measurement task, where the object under investigation was a masterball, or "datum sphere" (Figure 6), whose data points were sampled in just twelve equally spaced points of its equator. In these conditions, some assumptions were made:

1. the distributions of the coordinates of each point were assumed to be Gaussian, even with a low (= 10) number of repetitions; this assertion was supported by performing normality tests on the coordinates of the probed points through the software Originlab Origin Pro 2021.
2. No conditions about the "real" shape error of the datum sphere were taken into account to be used in the model, because of the lack of a reference certificate: the datum sphere was accordingly considered "perfect" with respect to the CMM capabilities[1], so that no specific distribution describing its form error could be added to perturbate the *x*, *y* coordinates of the probed points.

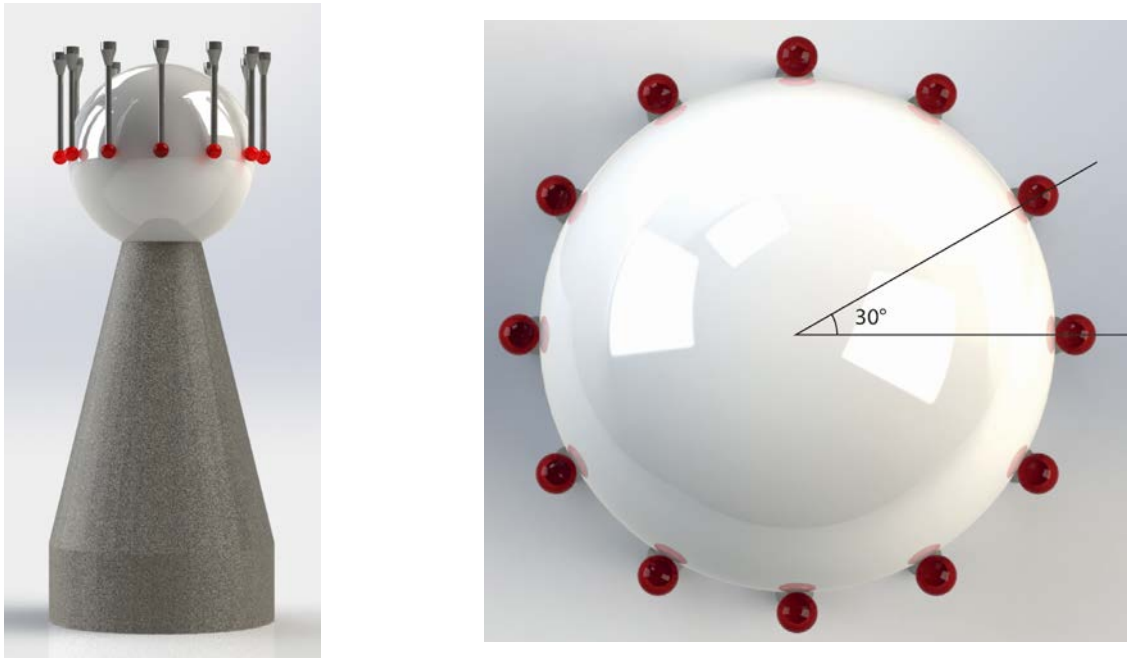

Figure 6: A "datum sphere" (with a diameter of 30 mm) was sampled in 12 different positions, with an angular spacing of 30°

[1] The CMM used for sampling the surface of the sphere was a Leitz PMM-C, currently equipping one of the laboratories for Dimensional Metrology at INRiM, with a B4 controller and TRX probe head.

From the mentioned distributions, *N* Monte Carlo samplings were generated by means of an originally developed Python script (working in 2.7.18 release), in order to obtain different *N* roundness errors based on the four definitions cited in the previous paragraph, and their respective standard deviations. Starting from m = 10 repeated measurements of coordinates for each of the 12 representative points, normal distributions for x_i and y_i (i = 1, 2, 3, …, 12) were generated, according to their mean values and standard deviations σ_{xi} and σ_{yi}. For each distribution, a sample of N (N = 10000) elements was simulated, and for each set of new clouds of coordinates points the four types of roundness errors (*LSC*, *MCC*, *MIC* and *MZC*) were calculated, looping 4 different dedicated scripts. The next step was to calculate the average values and the expanded uncertainties of the roundness, and to plot the histograms of the distributions of the parameters. The final objective was to compare the "simulated" roundness errors with the ones calculated through the CMM software by analyzing the points effectively probed by the CMM itself on the sphere.

Python scripts

The codes implemented are reported (in their almost integral forms) in "Annex" sections; they are diffusely commented and nearly self-explicative. It must be remarked that the portions of the codes in charge of calculating the smallest enclosing circle from a set of points were taken from literature [14], as well as for the largest inscribed circle [15] and for least square circle [16]; also for the minimum zone circle strategy literature was consulted [17]. Efforts were made to integrate the sample codes to a Monte Carlo approach for generating the distributions of points needed for roundness error determination.

Results and discussion

From Figure 7 to Figure 10 the distributions of the four types of roundness errors can be appreciated; they show a substantial similarity, and also a slight asymmetry. In Table 1 and Table 2 the results of the calculations of the four errors are shown, with their relative uncertainties.

	Monte Carlo simulations			
	LSC	***MCC***	***MIC***	***MZC***
ε / µm	0.432	0.457	0.463	0.409
σ / µm	0.098	0.124	0.114	0.104

Table 1: : simulated roundness errors results, with respective standard deviations

	CMM software			
	LSC	***MCC***	***MIC***	***MZC***
ε / µm	0.303	0.345	0.406	0.263
σ / µm	0.022	0.040	0.035	0.013

Table 2: roundness error values obtained through CMM software

The average values obtained by the python scripts seem to overestimate the reference measurements obtained through CMM software. Moreover, in accordance with ISO 15530-4:2011 recommendation which refers to the method of testing software for evaluation of uncertainty, these results appear not to be mutually compatible in all cases[2]; anyway, it must be pointed out that the algorithms used by CMM software are not accessible, so an exhaustive comparison on the basis of the mathematical implementations cannot be

[2] The compatibility indexes $E_n = \frac{|\varepsilon_{calc} - \varepsilon_{CMM}|}{\sqrt{U_{calc}^2 - U_{CMM}^2}}$ are > 1 for LSC and MZC (respectively 1.3 and 1.4), < 1 for MCC and MIC (respectively 0.9 and 0.5)

performed. This work is aimed to propose an alternative strategy to calculate the roundness errors of CMM-probed points taking advantage of the Monte Carlo method, a robust and diffusely used technique for evaluating the statistical uncertainty.

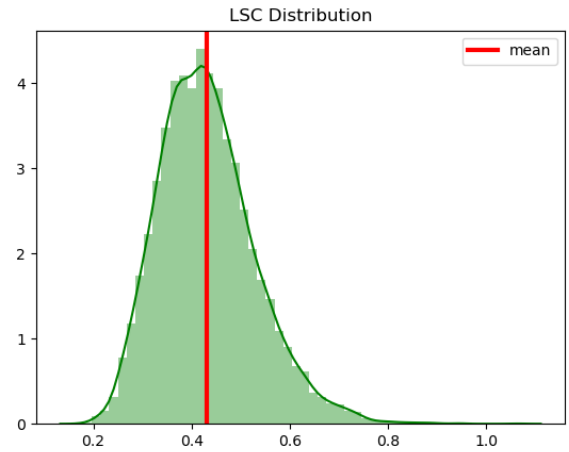

Figure 7: LSC distribution

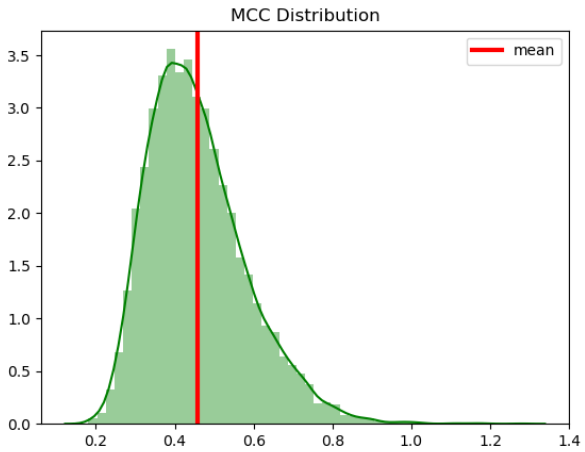

Figure 8: MCC distribution

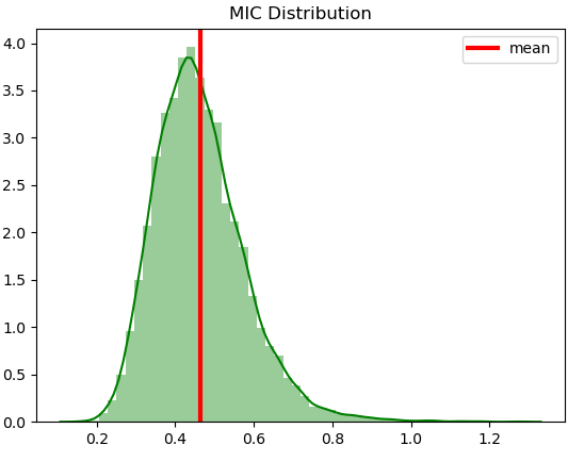

Figure 9: MIC distribution

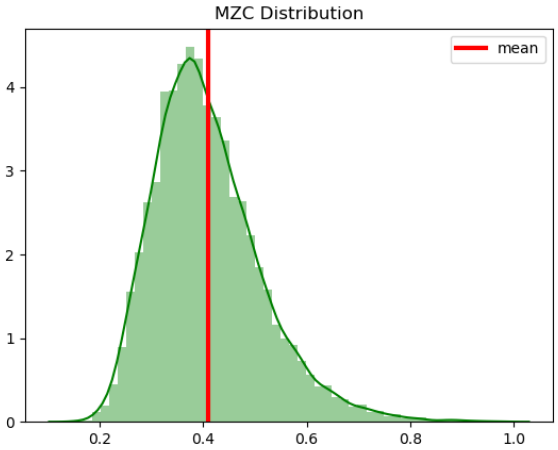

Figure 10: MZC distribution

References

[1] '*A guide to the Measurement of Roundness*'

[2] Roundness error definitions (Wikipedia)

[3] ISO 12181-1:2011 Geometrical product specifications (GPS) — Roundness — Part 1: Vocabulary and parameters of roundness

[4] *Four Methods for Roundness Evaluation* - Wentao Sui, Dan Zhang, 2012 International Conference on Applied Physics and Industrial Engineering https://doi.org/10.1016/j.phpro.2012.02.317

[5] *Computational Surface and Roundness Metrology* - Muralikrishnan, Balasubramanian, Raja, Jayaraman, Springer Science & Business Media, 11 set 2008

[6] GUM: Guide to the Expression of Uncertainty in Measurement

[7] *The "Virtual CMM" a software tool for uncertainty evaluation – practical application in an accredited calibration lab* - Michael Trenk, Matthias Franke, Dr. Heinrich Schwenke, in ASPE Summer Topical Meeting on Uncertainty Analysis in Measurement and Design. State College, Pennsylvania, USA, 2004

[8] Virtual Coordinate Measuring Machine (VCMM): https://tinyurl.com/yxe629rm

[9] *Evaluation of CM Uncertainty Through Monte Carlo Simulations* - Balsamo, A., Di Ciommo, M., Mugno, R., Rebaglia B.I., Ricci, E., Grella, R., CIRP Annals - Manufacturing Technology Vol. 48, No. 1, pp. 425-428, 1999

[10] *Evaluation of Coordinate Measurement Uncertainty with Use of Virtual Machine model based on Monte Carlo Method* - Sladek, J., Gaska, A., Measurement, Vol.45, pp. 1565-1575, 2012

[11] *Uncertainty determination for CMMs by Monte Carlo simulation integrating feature form deviations* - Kruth, J.P., Van Gestel, N., Bleys, P., Welkenhuyzen, F., CIRP – Manufacturing Technology, Vol. 58, pp. 463-466, 2009

[12] *Effect of CMM Point Coordinate Uncertainty on Uncertainties in Determination of Circular Features* - Dhanish, P.B., Mathew, J., Measurement, Vol. 39, p.p. 522-531, 2006.

[13] *Flatness error evaluation an verification based on new generation geometrical product specification (GPS)* - Wen, X-L., Zhu, X-C., Zhao, B-Y., Wang, D-X, Wang, F-L., Precision Engineering, Vol. 36, p.p. 70-76, 2012

[14] Project Nayuki's smallest enclosing circle

[15] Python Package Index (PyPI) MaximumInscribedCircle 0.1.6

[16] Python Package Index (PyPI) circle-fit 0.1.3

[17] *The min–max problem for evaluating the form error of a circle* - Wen-Yuh Jywea, Chien-Hong Liub, Cha'o-Kuang Chenb - Measurement, Volume 26, Issue 4, p. 273-282 (1999).

Annex A: LSC

```
# -*- coding: utf-8 -*-
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# definisco gli array contenenti le coordinate dei 12 punti toccati
sull'equatore della sfera di qualifica;
# ciascun vettore dell'array contiene le coordinate X e Y ripetute 10 volte:
P1 = [[16.49985, 16.49986, 16.49986, 16.49988, 16.49989, 16.49988, 16.49988,
16.49987, 16.49986, 16.49983],
    [-0.00192,-0.00193, -0.00204, -0.00193, -0.00209, -0.00209, -0.00194, -
0.00216, -0.00195, -0.00219]]
P2 = [[14.29004, 14.29018 , 14.29011, 14.29003, 14.28988, 14.29012, 14.29013,
14.28993, 14.28986, 14.29001],
    [8.249, 8.24876, 8.2489, 8.24919, 8.24932, 8.24895, 8.24897, 8.24922,
8.24935, 8.24903]]
P3 = [[8.24718, 8.2471, 8.24707, 8.24707, 8.24718, 8.24693, 8.24709, 8.24676,
8.24678, 8.24686],
    [14.2913, 14.29128, 14.29128, 14.2914, 14.29124, 14.2914, 14.29133,
14.29153, 14.29155, 14.29145]]
P4 = [[-5E-4, -3E-4, -4.9E-4, -4.4E-4, -2.7E-4, -3.4E-4, -1.7E-4, -5.2E-4, -5E-
4, -3.2E-4],
    [16.50016, 16.50014, 16.50013, 16.50021, 16.50013, 16.50013, 16.50016,
16.50018, 16.50014, 16.50016]]
P5 = [[-8.24714, -8.24706, -8.24694, -8.24718, -8.24717, -8.24695, -8.24705, -
8.24685, -8.24665, -8.24671],
    [14.2912, 14.29123, 14.29134, 14.29119, 14.29114, 14.29131, 14.29124,
14.29137, 14.29144, 14.29143]]
P6 = [[-14.28781, -14.2881, -14.28799, -14.28803, -14.28831, -14.28816, -
14.28782, -14.28782, -14.28777, -14.28796],
    [8.25302, 8.25253, 8.25268, 8.25253, 8.25206, 8.25236, 8.25289, 8.2529,
8.25299, 8.25271]]
P7 = [[-16.4999, -16.49991, -16.49991, -16.49988, -16.49987, -16.49989, -
16.49991, -16.49993, -16.49992, -16.4999],
    [9.5E-4, 0.00115, 9.7E-4, 8.5E-4, 7.9E-4, 9.3E-4, 8.7E-4, 7.7E-4, 8.6E-4]]
P8 = [[-14.28836, -14.28838, -14.28843, -14.28834, -14.28821, -14.28829, -
14.2883, -14.28835, -14.28834, -14.28829],
    [-8.25156, -8.25164, -8.25151, -8.25156, -8.25186, -8.25166, -8.25166, -
8.25156, -8.2516, -8.2517]]
P9 = [[-8.24631, -8.24626, -8.24624, -8.2465, -8.24625, -8.24647, -8.24655, -
8.24636, -8.24617, -8.24612],
    [-14.29165, -14.29169, -14.29175, -14.29153, -14.2917, -14.29157, -14.29152,
-14.29159, -14.29173, -14.29173]]
P10 = [[3.1E-4, 1.6E-4, 3E-5, 9E-5, 2.1E-4, 3.1E-4, 1.4E-4, 1.8E-4, 2.5E-4,
2.8E-4],
    [-16.49996, -16.50001, -16.49998, -16.49994, -16.49999, -16.5, -16.49997, -
16.49997, -16.49999, -16.50001]]
P11 = [[8.24753, 8.24709, 8.24739, 8.24715, 8.2473, 8.24765, 8.24717, 8.24716,
8.24703, 8.24715],
    [-14.29091, -14.29115, -14.29097, -14.29107, -14.29104, -14.29083, -
14.29104, -14.29107, -14.29117, -14.29106]]
P12 = [[14.28773, 14.2875, 14.28795, 14.28767, 14.28788, 14.28795, 14.28762,
14.28762, 14.28757, 14.28769],
    [-8.25278, -8.25321, -8.25246, -8.25298, -8.25259, -8.25252, -8.25295, -
8.25296, -8.25305, -8.25285]]

# calcolo medie e deviazioni standard di ciascun set di coordinate:
P1_medio_x = np.average(P1[0]); P1_sigma_x = np.std(P1[0])
P1_medio_y = np.average(P1[1]); P1_sigma_y = np.std(P1[1])
P2_medio_x = np.average(P2[0]); P2_sigma_x = np.std(P2[0])
P2_medio_y = np.average(P2[1]); P2_sigma_y = np.std(P2[1])
```

```
P3_medio_x = np.average(P3[0]); P3_sigma_x = np.std(P3[0])
P3_medio_y = np.average(P3[1]); P3_sigma_y = np.std(P3[1])
P4_medio_x = np.average(P4[0]); P4_sigma_x = np.std(P4[0])
P4_medio_y = np.average(P4[1]); P4_sigma_y = np.std(P4[1])
P5_medio_x = np.average(P5[0]); P5_sigma_x = np.std(P5[0])
P5_medio_y = np.average(P5[1]); P5_sigma_y = np.std(P5[1])
P6_medio_x = np.average(P6[0]); P6_sigma_x = np.std(P6[0])
P6_medio_y = np.average(P6[1]); P6_sigma_y = np.std(P6[1])
P7_medio_x = np.average(P7[0]); P7_sigma_x = np.std(P7[0])
P7_medio_y = np.average(P7[1]); P7_sigma_y = np.std(P7[1])
P8_medio_x = np.average(P8[0]); P8_sigma_x = np.std(P8[0])
P8_medio_y = np.average(P8[1]); P8_sigma_y = np.std(P8[1])
P9_medio_x = np.average(P9[0]); P9_sigma_x = np.std(P9[0])
P9_medio_y = np.average(P9[1]); P9_sigma_y = np.std(P9[1])
P10_medio_x = np.average(P10[0]); P10_sigma_x = np.std(P10[0])
P10_medio_y = np.average(P10[1]); P10_sigma_y = np.std(P10[1])
P11_medio_x = np.average(P11[0]); P11_sigma_x = np.std(P11[0])
P11_medio_y = np.average(P11[1]); P11_sigma_y = np.std(P11[1])
P12_medio_x = np.average(P12[0]); P12_sigma_x = np.std(P12[0])
P12_medio_y = np.average(P12[1]); P12_sigma_y = np.std(P12[1])

num_val = 10000  # sarà la dimensione dei vettori di ascisse e coordinate di
ciascun punto
# la larghezza di cui tener conto nella distribuzione gaussiana delle coordinate
sarà data dalla sigma statistica
form_measurement_error = 0.00045  # in mm . "Single stylus form error" (PFTU)
della CMM Leitz PMMC 12.10.7, come da datasheet di CMM Leitz (
https://snipurl.im/bRAnD )

# le distribuzioni finali saranno date dalle distribuzioni statistiche dei 12
punti di partenza.
# generazione delle distribuzioni gaussiane sulla base di valori medi e dev.st.
calcolati sopra:

gauss_x1 = np.random.normal(P1_medio_x, np.std(P1[0]), num_val)
gauss_y1 = np.random.normal(P1_medio_y, np.std(P1[1]), num_val)
gauss_x2 = np.random.normal(P2_medio_x, np.std(P2[0]), num_val)
gauss_y2 = np.random.normal(P2_medio_y, np.std(P2[1]), num_val)
gauss_x3 = np.random.normal(P3_medio_x, np.std(P3[0]), num_val)
gauss_y3 = np.random.normal(P3_medio_y, np.std(P3[1]), num_val)
gauss_x4 = np.random.normal(P4_medio_x, np.std(P4[0]), num_val)
gauss_y4 = np.random.normal(P4_medio_y, np.std(P4[1]), num_val)
gauss_x5 = np.random.normal(P5_medio_x, np.std(P5[0]), num_val)
gauss_y5 = np.random.normal(P5_medio_y, np.std(P5[1]), num_val)
gauss_x6 = np.random.normal(P6_medio_x, np.std(P6[0]), num_val)
gauss_y6 = np.random.normal(P6_medio_y, np.std(P6[1]), num_val)
gauss_x7 = np.random.normal(P7_medio_x, np.std(P7[0]), num_val)
gauss_y7 = np.random.normal(P7_medio_y, np.std(P7[1]), num_val)
gauss_x8 = np.random.normal(P8_medio_x, np.std(P8[0]), num_val)
gauss_y8 = np.random.normal(P8_medio_y, np.std(P8[1]), num_val)
gauss_x9 = np.random.normal(P9_medio_x, np.std(P9[0]), num_val)
gauss_y9 = np.random.normal(P9_medio_y, np.std(P9[1]), num_val)
gauss_x10 = np.random.normal(P10_medio_x, np.std(P10[0]), num_val)
gauss_y10 = np.random.normal(P10_medio_y, np.std(P10[1]), num_val)
gauss_x11 = np.random.normal(P11_medio_x, np.std(P11[0]), num_val)
gauss_y11 = np.random.normal(P11_medio_y, np.std(P11[1]), num_val)
gauss_x12 = np.random.normal(P12_medio_x, np.std(P12[0]), num_val)
gauss_y12 = np.random.normal(P12_medio_y, np.std(P12[1]), num_val)

# separo le X e le Y delle distribuzioni appena trovate in due array 2D (con 12
righe e num_val colonne)
P_X = np.array([gauss_x1, gauss_x2, gauss_x3, gauss_x4, gauss_x5, gauss_x6,
gauss_x7, gauss_x8, gauss_x9, gauss_x10, gauss_x11, gauss_x12])
```

```
P_Y = np.array([gauss_y1, gauss_y2, gauss_y3, gauss_y4, gauss_y5, gauss_y6,
gauss_y7, gauss_y8, gauss_y9, gauss_y10, gauss_y11, gauss_y12])

# traspongo, così da avere num_val righe e 12 colonne
P_X_new = np.transpose(P_X)
P_Y_new = np.transpose(P_Y)


results_x = []
for i in range(0, num_val):
    i = P_X_new[i, :len(P_X)]
    results_x.append(i)

results_y = []
for j in range(0, num_val):
    j = P_Y_new[j, :len(P_Y)]
    results_y.append(j)


import circle_fit as cf

res = []
for i in range(num_val):
    punti = np.transpose([results_x[i], results_y[i]])
    xc,yc,r,_ = cf.least_squares_circle(punti)
    x,y = (punti[:,0], punti[:,1])
    x_0, y_0 = (xc, yc)
    d = np.sqrt((x-x_0)**2 + (y-y_0)**2)
    max_dist = np.max(d)
    min_dist = np.min(d)
    LSC = (max_dist - min_dist) * 1000 # Least square circle (LSC): It is a
circle which separates the roundness profile of an object by separating the sum
of total areas of the inside and outside it in equal amounts. The roundness
error then can be estimated as the difference between the maximum and minimum
distance from this reference circle
    print 'errore di rotondità n°', i+1, 'secondo metodo LSC:', round(LSC, 3) ,
'µm'
    res.append(LSC)
    np.savetxt('LSC.txt', res)


LSC = np.loadtxt('LSC.txt')

aver_LSC = np.mean(res); sigma_LSC = np.std(res)
print 'LSC medio:', round(aver_LSC, 3), 'µm'
print 'sigma LSC:', round(sigma_LSC, 3), 'µm'

istogramma = sns.distplot(LSC, color = 'green')
plt.axvline(LSC.mean(), color = 'red', linewidth = 3, label = 'mean')
plt.legend()
plt.title('LSC Distribution')
plt.show()
```

Annex B: MCC

```
# -*- coding: utf-8 -*-
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math, random
```

```
# stesso blocco di script per LSC, tagliato da riga 6 a riga 108

                        # MINIMO CERCHIO CIRCOSCRITTO:

    def make_circle(points):
        # Convert to float and randomize order
        shuffled = [(float(x), float(y)) for (x, y) in points]
        random.shuffle(shuffled)
        # Progressively add points to circle or recompute circle
        c = None
        for (i, p) in enumerate(shuffled):
            if c is None or not is_in_circle(c, p):
                c = _make_circle_one_point(shuffled[: i + 1], p)
        return c


    # One boundary point known
    def _make_circle_one_point(points, p):
        c = (p[0], p[1], 0.0)
        for (i, q) in enumerate(points):
            if not is_in_circle(c, q):
                if c[2] == 0.0:
                    c = make_diameter(p, q)
                else:
                    c = _make_circle_two_points(points[: i + 1], p, q)
        return c


    # Two boundary points known
    def _make_circle_two_points(points, p, q):
        circ = make_diameter(p, q)
        left = None
        right = None
        px, py = p
        qx, qy = q
        # For each point not in the two-point circle
        for r in points:
            if is_in_circle(circ, r):
                continue

            # Form a circumcircle and classify it on left or right side
            cross = _cross_product(px, py, qx, qy, r[0], r[1])
            c = make_circumcircle(p, q, r)
            if c is None:
                continue
            elif cross > 0.0 and (
                    left is None or _cross_product(px, py, qx, qy, c[0], c[1]) >
_cross_product(px, py, qx, qy, left[0],

left[1])):
                left = c
            elif cross < 0.0 and (
                    right is None or _cross_product(px, py, qx, qy, c[0], c[1])
< _cross_product(px, py, qx, qy,

right[0],

right[1])):
                right = c

        # Select which circle to return
        if left is None and right is None:
```

```
            return circ
        elif left is None:
            return right
        elif right is None:
            return left
        else:
            return left if (left[2] <= right[2]) else right


    def make_diameter(a, b):
        cx = (a[0] + b[0]) / 2
        cy = (a[1] + b[1]) / 2
        r0 = math.hypot(cx - a[0], cy - a[1])
        r1 = math.hypot(cx - b[0], cy - b[1])
        return (cx, cy, max(r0, r1))


    def make_circumcircle(a, b, c):
        # Mathematical algorithm from Wikipedia: Circumscribed circle
        ox = (min(a[0], b[0], c[0]) + max(a[0], b[0], c[0])) / 2
        oy = (min(a[1], b[1], c[1]) + max(a[1], b[1], c[1])) / 2
        ax = a[0] - ox;
        ay = a[1] - oy
        bx = b[0] - ox;
        by = b[1] - oy
        cx = c[0] - ox;
        cy = c[1] - oy
        d = (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by)) * 2.0
        if d == 0.0:
            return None
        x = ox + ((ax * ax + ay * ay) * (by - cy) + (bx * bx + by * by) * (cy -
ay) + (cx * cx + cy * cy) * (
                      ay - by)) / d
        y = oy + ((ax * ax + ay * ay) * (cx - bx) + (bx * bx + by * by) * (ax -
cx) + (cx * cx + cy * cy) * (
                      bx - ax)) / d
        ra = math.hypot(x - a[0], y - a[1])
        rb = math.hypot(x - b[0], y - b[1])
        rc = math.hypot(x - c[0], y - c[1])
        return (x, y, max(ra, rb, rc))


    _MULTIPLICATIVE_EPSILON = 1 + 1e-14


    def is_in_circle(c, p):
        return c is not None and math.hypot(p[0] - c[0], p[1] - c[1]) <= c[2] *
_MULTIPLICATIVE_EPSILON


    # Returns twice the signed area of the triangle defined by (x0, y0), (x1,
y1), (x2, y2).
    def _cross_product(x0, y0, x1, y1, x2, y2):
        return (x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0)


    # print make_circle(points)

    centro_x = make_circle(points)[0]
    centro_y = make_circle(points)[1]
    raggio_circ = make_circle(points)[2]
    # print 'raggio minimo cerchio circoscritto =', raggio_circ, 'mm'
```

```
    x, y = (points[:, 0], points[:, 1])
    x_0, y_0 = (centro_x, centro_y)
    dist = raggio_circ - np.sqrt((x - x_0) ** 2 + (y - y_0) ** 2)

    roundness_error_MCC = np.max(dist) # Minimum circumscribed circle (MCC): It
is defined as the smallest circle which encloses whole of the roundness profile.
Here the error is the largest deviation from this circle

    MCC = 1000 * roundness_error_MCC
    print 'errore di rotondità n°', i+1, 'secondo metodo MCC:', round(MCC, 3),
'µm'


    res.append(MCC)
    np.savetxt('MCC.txt', res)

MZC_new = np.loadtxt('MCC.txt')

aver_MCC = np.average(res); sigma_MCC = np.std(res)
print 'MCC medio:', round(aver_MCC,3), 'µm'
print 'sigma MCC:', round(sigma_MCC, 3), 'µm'

istogramma = sns.distplot(MCC, color = 'green')
plt.axvline(MCC.mean(), color = 'red', linewidth = 3, label = 'mean')
plt.legend()
plt.title('MCC Distribution')
plt.show()
```

Annex C: MIC

```
# -*- coding: utf-8 -*-
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math, random
from numpy import linalg

# stesso blocco di script per LSC, tagliato da riga 6 a riga 108

                        # MASSIMO CERCHIO INSCRITTO:

    epsilon = 1e-6

    class Point():
        def __init__(self, x, y):
            self.x = x
            self.y = y

        @classmethod
        def fromList(cls, p):
            return cls(p[0], p[1])

        def __cmp__(self, other):
            return cmp(self.x, other.x) and cmp(self.y, other.y)

        def __str__(self):
            return 'x : {}, y : {}'.format(self.x, self.y)

        def toTuple(self):
```

```
            return (self.x, self.y)


    def listToPoint(p):
        return Point(p[0], p[1]);


    class Vector():

        def __init__(self, p1, p2):
            self.__construtor(p1, p2)

        def __construtor(self, p1, p2):
            self.fromPoint = p1
            self.toPoint = p2
            self.diffPoint = Point(p2.x - p1.x, p2.y - p1.y)

        def __str__(self):
            return 'cor : {}'.format(self.diffPoint)

        def setToPoint(self, p):
            self.__construtor(self.fromPoint, p)

        def setNorm(self, newNorm):
            if abs(newNorm) < epsilon:
                self.__construtor(self.p1, self.p1)
            else:
                factor = newNorm / self.norm()
                x = self.diffPoint.x * factor
                y = self.diffPoint.y * factor
                self.diffPoint = Point(x, y)
                self.toPoint = Point(self.fromPoint.x + x,
                                     self.fromPoint.y + y)
            return self

        def norm(self):
            xsquare = self.diffPoint.x * self.diffPoint.x
            ysquare = self.diffPoint.y * self.diffPoint.y
            return math.sqrt(xsquare + ysquare)

        def dotProduct(self, other):
            return self.diffPoint.x * other.diffPoint.x + self.diffPoint.y *
other.diffPoint.y

        def includedAngleIndegrees(self, other):
            rad = self.includedAngleRad(other)
            return math.degrees(rad)

        def includedAngle(self, other):
            return self.includedAngleIndegrees(other)

        def includedAngleCos(self, other):

            return self.dotProduct(other) / (self.norm() * other.norm())

        def includedAngleRad(self, other):
            return math.acos(self.includedAngleCos(other))


    class MaximumInscribledCircle():
        """docstring for MaximumInscribledCircle"""

        def __init__(self, points):
```

```
            if not isinstance(points[0], Point):
                xs, ys = np.mean(points, axis=0)
                self.points = [listToPoint(p) for p in points]
            else:
                self.points = points
                xs = np.mean((p.x for p in points))
                ys = np.mean((p.y for p in points))
            self.center = Point(xs, ys)

        def __getNorms(self, tmpVector, p):
            other = Vector(tmpVector.fromPoint, p)
            if abs(other.norm()) < epsilon:
                return tmpVector.norm() / epsilon
            costheta = tmpVector.dotProduct(other) / (tmpVector.norm() *
other.norm())
            if costheta < epsilon:
                return tmpVector.norm() / epsilon
            # print ('other norm', other.norm(), costheta,
tmpVector.dotProduct(other))
            return other.norm() / costheta / 2

        def getIntersection(self, tmpVector, newNorm):
            newCenter = Vector(tmpVector.fromPoint,
tmpVector.toPoint).setNorm(newNorm).toPoint
            return newCenter

        def getCenter(self):
            originalPoint = Point(0, 0)
            firstNorm = self.first.x * self.first.x + self.first.y *
self.first.y
            secondNorm = self.second.x * self.second.x + self.second.y *
self.second.y
            thirdNorm = self.third.x * self.third.x + self.third.y *
self.third.y

            xdelta1 = 2 * (self.first.x - self.second.x)
            ydelta1 = 2 * (self.first.y - self.second.y)
            xdelta2 = 2 * (self.third.x - self.second.x)
            ydelta2 = 2 * (self.third.y - self.second.y)
            coefficient = np.array([[xdelta1, ydelta1], [xdelta2, ydelta2]])
            ordinate = np.array([firstNorm - secondNorm, thirdNorm -
secondNorm])
            center = linalg.solve(coefficient, ordinate)
            return center

        def getThirdPoint(self, points):
            lineVector = Vector(self.first, self.second)
            lineVectorNorm = lineVector.norm() / 2

            def getCosTheta(p):
                other = Vector(self.directionVector.fromPoint, p)
                if other.norm() < epsilon:
                    return -epsilon
                tmp = self.directionVector.includedAngleCos(other)
                return - epsilon if tmp < epsilon else tmp

            def getSinTheta(p):
                tmpVector = Vector(p, self.first)
                other = Vector(p, self.second)
                x = tmpVector.includedAngleCos(other)
                return math.sqrt(1 - x * x)
```

```
            costhetas = [getCosTheta(p) for p in points]
            sinThetas = [epsilon] * len(costhetas)
            for i in range(len(sinThetas)):
                if costhetas[i] > epsilon:
                    sinThetas[i] = getSinTheta(points[i])
            radiusApproximate = [lineVectorNorm / s for s in sinThetas]
            pidx = np.argmin(radiusApproximate)
            self.third = points[pidx]
            self.final_radius = radiusApproximate[pidx]

        def getSecondPoint(self, points):

            norms = [self.__getNorms(self.directionVector, p) for p in points]
            pidx = np.argmin(norms)
            intersection = self.getIntersection(self.directionVector,
norms[pidx])
            self.center = intersection
            return points[pidx]

        def getMinimalDistancePoint(self):
            center = self.center
            points = self.points
            getDist = lambda p: np.sqrt((center.x - p.x) * (center.x - p.x) +
                                        (center.y - p.y) * (center.y - p.y))
            dist = [getDist(p) for p in points]
            return points[np.argmin(dist)]

        def verifyAngle(self):
            if Vector(self.first, self.second).dotProduct(Vector(self.first,
self.third)) < epsilon:
                return -1
            if Vector(self.second, self.first).dotProduct(Vector(self.second,
self.third)) < epsilon:
                return -2
            return 0

        def getRadius(self):
            self.first = self.getMinimalDistancePoint()
            points = self.points
            self.directionVector = Vector(self.first, self.center)
            self.second = self.getSecondPoint(points)
            middle = Point((self.first.x + self.second.x) / 2,
                           (self.first.y + self.second.y) / 2)
            self.directionVector = Vector(middle, self.center)
            self.getThirdPoint(points)
            code = self.verifyAngle()
            while code != 0:
                self.center = self.getCenter()
                self.center = listToPoint(self.center)
                if code == -1:
                    self.first = self.third
                else:
                    self.second = self.third
                middle = Point((self.first.x + self.second.x) / 2, (self.first.y
+ self.second.y) / 2)
                self.directionVector = Vector(middle, self.center)
                self.getThirdPoint(points)
                code = self.verifyAngle()
                center = self.getCenter()
            return self.final_radius


    def PointsInCircum(r, n=100):
```

```
        data = [(math.cos(2 * math.pi / n * x) * r, math.sin(2 * math.pi / n *
x) * r) for x in range(n + 1)]
        data = [(p[0] + random.random(), p[1] + random.random()) for p in data]
        return data


    def testFunc():
        global max_r_inscr
        global centre
        circle = MaximumInscribledCircle(points)
        max_r_inscr = circle.getRadius()
        centre = circle.getCenter()


    def main():
        testFunc()

    if __name__ == '__main__':
        main()


    # print 'raggio massimo cerchio inscritto =', max_r_inscr, 'mm'

    x, y = (points[:, 0], points[:, 1])
    x_0, y_0 = (centre[0], centre[1])
    dist = np.sqrt((x - x_0) ** 2 + (y - y_0) ** 2) - max_r_inscr
    MIC = np.max(dist) * 1000 # Maximum inscribed circle (MIC): It is defined as
the largest circle that can be inscribed inside the roundness profile. The
roundness error here again is the maximum deviation of the profile from this
inscribed circle.
    print 'errore di rotondità n°', i+1, 'secondo metodo MIC:', (round(MIC, 3)),
'µm'

    res.append(MIC)
    np.savetxt('MIC.txt', res)

MIC_new = np.loadtxt('MIC.txt')

aver_MIC = np.average(res);
sigma_MIC = np.std(res)
print 'MIC medio:', round(aver_MIC, 3), 'µm'
print 'sigma MIC:', round(sigma_MIC, 3), 'µm'

istogramma = sns.distplot(MIC, color = 'green')
plt.axvline(MIC.mean(), color = 'red', linewidth = 3, label = 'mean')
plt.legend()
plt.title('MIC Distribution')
plt.show()
```

Annex D: MZC

```
# -*- coding: utf-8 -*-
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math, random
from numpy import linalg

# stesso blocco di script per LSC, tagliato da riga 6 a riga 108
```

```
                    # MINIMO CERCHIO CIRCOSCRITTO:

    def make_circle(points):
        # Convert to float and randomize order
        shuffled = [(float(x), float(y)) for (x, y) in points]
        random.shuffle(shuffled)
        # Progressively add points to circle or recompute circle
        c = None
        for (i, p) in enumerate(shuffled):
            if c is None or not is_in_circle(c, p):
                c = _make_circle_one_point(shuffled[: i + 1], p)
        return c


    # One boundary point known
    def _make_circle_one_point(points, p):
        c = (p[0], p[1], 0.0)
        for (i, q) in enumerate(points):
            if not is_in_circle(c, q):
                if c[2] == 0.0:
                    c = make_diameter(p, q)
                else:
                    c = _make_circle_two_points(points[: i + 1], p, q)
        return c


    # Two boundary points known
    def _make_circle_two_points(points, p, q):
        circ = make_diameter(p, q)
        left = None
        right = None
        px, py = p
        qx, qy = q
        # For each point not in the two-point circle
        for r in points:
            if is_in_circle(circ, r):
                continue

            # Form a circumcircle and classify it on left or right side
            cross = _cross_product(px, py, qx, qy, r[0], r[1])
            c = make_circumcircle(p, q, r)
            if c is None:
                continue
            elif cross > 0.0 and (
                    left is None or _cross_product(px, py, qx, qy, c[0], c[1]) >
_cross_product(px, py, qx, qy, left[0],

left[1])):
                left = c
            elif cross < 0.0 and (
                    right is None or _cross_product(px, py, qx, qy, c[0], c[1])
< _cross_product(px, py, qx, qy,

right[0],

right[1])):
                right = c

        # Select which circle to return
        if left is None and right is None:
            return circ
        elif left is None:
```

```
            return right
        elif right is None:
            return left
        else:
            return left if (left[2] <= right[2]) else right


    def make_diameter(a, b):
        cx = (a[0] + b[0]) / 2
        cy = (a[1] + b[1]) / 2
        r0 = math.hypot(cx - a[0], cy - a[1])
        r1 = math.hypot(cx - b[0], cy - b[1])
        return (cx, cy, max(r0, r1))


    def make_circumcircle(a, b, c):
        # Mathematical algorithm from Wikipedia: Circumscribed circle
        ox = (min(a[0], b[0], c[0]) + max(a[0], b[0], c[0])) / 2
        oy = (min(a[1], b[1], c[1]) + max(a[1], b[1], c[1])) / 2
        ax = a[0] - ox;
        ay = a[1] - oy
        bx = b[0] - ox;
        by = b[1] - oy
        cx = c[0] - ox;
        cy = c[1] - oy
        d = (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by)) * 2.0
        if d == 0.0:
            return None
        x = ox + ((ax * ax + ay * ay) * (by - cy) + (bx * bx + by * by) * (cy -
ay) + (cx * cx + cy * cy) * (
                ay - by)) / d
        y = oy + ((ax * ax + ay * ay) * (cx - bx) + (bx * bx + by * by) * (ax -
cx) + (cx * cx + cy * cy) * (
                bx - ax)) / d
        ra = math.hypot(x - a[0], y - a[1])
        rb = math.hypot(x - b[0], y - b[1])
        rc = math.hypot(x - c[0], y - c[1])
        return (x, y, max(ra, rb, rc))


    _MULTIPLICATIVE_EPSILON = 1 + 1e-14


    def is_in_circle(c, p):
        return c is not None and math.hypot(p[0] - c[0], p[1] - c[1]) <= c[2] *
_MULTIPLICATIVE_EPSILON


    # Returns twice the signed area of the triangle defined by (x0, y0), (x1,
y1), (x2, y2).
    def _cross_product(x0, y0, x1, y1, x2, y2):
        return (x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0)


    centro_x = make_circle(points)[0]
    centro_y = make_circle(points)[1]
    raggio_circ = make_circle(points)[2]
    # print 'X centro cerchio circoscritto =', centro_x
    # print 'Y centro cerchio circoscritto =', centro_y
    # print 'raggio minimo cerchio circoscritto =', raggio_circ, 'mm'


# MASSIMO CERCHIO INSCRITTO:
```

```
    epsilon = 1e-6


    class Point():
        def __init__(self, x, y):
            self.x = x
            self.y = y

        @classmethod
        def fromList(cls, p):
            return cls(p[0], p[1])

        def __cmp__(self, other):
            return cmp(self.x, other.x) and cmp(self.y, other.y)

        def __str__(self):
            return 'x : {}, y : {}'.format(self.x, self.y)

        def toTuple(self):
            return (self.x, self.y)


    def listToPoint(p):
        return Point(p[0], p[1]);


    class Vector():

        def __init__(self, p1, p2):
            self.__construtor(p1, p2)

        def __construtor(self, p1, p2):
            self.fromPoint = p1
            self.toPoint = p2
            self.diffPoint = Point(p2.x - p1.x, p2.y - p1.y)

        def __str__(self):
            return 'cor : {}'.format(self.diffPoint)

        def setToPoint(self, p):
            self.__construtor(self.fromPoint, p)

        def setNorm(self, newNorm):
            if abs(newNorm) < epsilon:
                self.__construtor(self.p1, self.p1)
            else:
                factor = newNorm / self.norm()
                x = self.diffPoint.x * factor
                y = self.diffPoint.y * factor
                self.diffPoint = Point(x, y)
                self.toPoint = Point(self.fromPoint.x + x,
                                     self.fromPoint.y + y)
            return self

        def norm(self):
            xsquare = self.diffPoint.x * self.diffPoint.x
            ysquare = self.diffPoint.y * self.diffPoint.y
            return math.sqrt(xsquare + ysquare)

        def dotProduct(self, other):
            return self.diffPoint.x * other.diffPoint.x + self.diffPoint.y *
other.diffPoint.y
```

```
        def includedAngleIndegrees(self, other):
            rad = self.includedAngleRad(other)
            return math.degrees(rad)

        def includedAngle(self, other):
            return self.includedAngleIndegrees(other)

        def includedAngleCos(self, other):

            return self.dotProduct(other) / (self.norm() * other.norm())

        def includedAngleRad(self, other):
            return math.acos(self.includedAngleCos(other))


    class MaximumInscribledCircle():
        """docstring for MaximumInscribledCircle"""

        def __init__(self, points):

            if not isinstance(points[0], Point):
                xs, ys = np.mean(points, axis=0)
                self.points = [listToPoint(p) for p in points]
            else:
                self.points = points
                xs = np.mean((p.x for p in points))
                ys = np.mean((p.y for p in points))
            self.center = Point(xs, ys)

        def __getNorms(self, tmpVector, p):
            other = Vector(tmpVector.fromPoint, p)
            if abs(other.norm()) < epsilon:
                return tmpVector.norm() / epsilon
            costheta = tmpVector.dotProduct(other) / (tmpVector.norm() *
other.norm())
            if costheta < epsilon:
                return tmpVector.norm() / epsilon
            # print ('other norm', other.norm(), costheta,
tmpVector.dotProduct(other))
            return other.norm() / costheta / 2

        def getIntersection(self, tmpVector, newNorm):
            newCenter = Vector(tmpVector.fromPoint,
tmpVector.toPoint).setNorm(newNorm).toPoint
            return newCenter

        def getCenter(self):
            originalPoint = Point(0, 0)
            firstNorm = self.first.x * self.first.x + self.first.y *
self.first.y
            secondNorm = self.second.x * self.second.x + self.second.y *
self.second.y
            thirdNorm = self.third.x * self.third.x + self.third.y *
self.third.y

            xdelta1 = 2 * (self.first.x - self.second.x)
            ydelta1 = 2 * (self.first.y - self.second.y)
            xdelta2 = 2 * (self.third.x - self.second.x)
            ydelta2 = 2 * (self.third.y - self.second.y)
            coefficient = np.array([[xdelta1, ydelta1], [xdelta2, ydelta2]])
            ordinate = np.array([firstNorm - secondNorm, thirdNorm -
secondNorm])
```

```
            center = linalg.solve(coefficient, ordinate)

            return center

        def getThirdPoint(self, points):
            lineVector = Vector(self.first, self.second)
            lineVectorNorm = lineVector.norm() / 2

            def getCosTheta(p):
                other = Vector(self.directionVector.fromPoint, p)
                if other.norm() < epsilon:
                    return -epsilon
                tmp = self.directionVector.includedAngleCos(other)
                return - epsilon if tmp < epsilon else tmp

            def getSinTheta(p):
                tmpVector = Vector(p, self.first)
                other = Vector(p, self.second)
                x = tmpVector.includedAngleCos(other)
                return math.sqrt(1 - x * x)

            costhetas = [getCosTheta(p) for p in points]
            sinThetas = [epsilon] * len(costhetas)
            for i in range(len(sinThetas)):
                if costhetas[i] > epsilon:
                    sinThetas[i] = getSinTheta(points[i])
            radiusApproximate = [lineVectorNorm / s for s in sinThetas]
            # print ('radius approximate', radiusApproximate, lineVectorNorm)
            pidx = np.argmin(radiusApproximate)
            self.third = points[pidx]
            self.final_radius = radiusApproximate[pidx]

        def getSecondPoint(self, points):

            norms = [self.__getNorms(self.directionVector, p) for p in points]
            pidx = np.argmin(norms)
            intersection = self.getIntersection(self.directionVector,
norms[pidx])
            self.center = intersection
            # print (Vector(intersection, self.first).norm(), 'first to center')
            # print (Vector(intersection, points[pidx]).norm(), 'second to
center')

            return points[pidx]

        def getMinimalDistancePoint(self):
            center = self.center
            points = self.points
            getDist = lambda p: np.sqrt((center.x - p.x) * (center.x - p.x) +
                                        (center.y - p.y) * (center.y - p.y))
            dist = [getDist(p) for p in points]

            return points[np.argmin(dist)]

        def verifyAngle(self):
            if Vector(self.first, self.second).dotProduct(Vector(self.first,
self.third)) < epsilon:
                return -1
            if Vector(self.second, self.first).dotProduct(Vector(self.second,
self.third)) < epsilon:
                return -2
            return 0
```

```
        def getRadius(self):

            self.first = self.getMinimalDistancePoint()

            points = self.points
            self.directionVector = Vector(self.first, self.center)
            self.second = self.getSecondPoint(points)

            middle = Point((self.first.x + self.second.x) / 2,
                           (self.first.y + self.second.y) / 2)
            self.directionVector = Vector(middle, self.center)
            self.getThirdPoint(points)
            code = self.verifyAngle()
            while code != 0:
                self.center = self.getCenter()
                self.center = listToPoint(self.center)
                if code == -1:
                    self.first = self.third
                else:
                    self.second = self.third
                middle = Point((self.first.x + self.second.x) / 2, (self.first.y
+ self.second.y) / 2)
                self.directionVector = Vector(middle, self.center)
                self.getThirdPoint(points)
                code = self.verifyAngle()
            return self.final_radius


    def PointsInCircum(r, n=100):
        data = [(math.cos(2 * math.pi / n * x) * r, math.sin(2 * math.pi / n *
x) * r) for x in range(n + 1)]
        data = [(p[0] + random.random(), p[1] + random.random()) for p in data]
        return data


    def testFunc():
        global r_inscr
        global centre
        circle = MaximumInscribledCircle(points)
        r_inscr = circle.getRadius()
        centre = circle.getCenter()


    def main():
        testFunc()


    if __name__ == '__main__':
        main()

    # print 'raggio massimo cerchio inscritto =', r_inscr, 'mm'

    #  articolo di riferimento: 'The min-max problem for evaluating the form
error of a circle':

    x, y = (points[:, 0], points[:, 1])

    # 3-1 model:
    x_0_circ, y_0_circ = (centro_x, centro_y)
    dist_circ_max = np.max(np.sqrt((x - x_0_circ) ** 2 + (y - y_0_circ) ** 2))
    dist_circ_min = np.min(np.sqrt((x - x_0_circ) ** 2 + (y - y_0_circ) ** 2))
    MZC_three_one = (dist_circ_max - dist_circ_min) * 1000
```

```
    # 1-3 model:
    x_0_insc, y_0_insc = (centre[0], centre[1])
    dist_insc_max = np.max(np.sqrt((x - x_0_insc) ** 2 + (y - y_0_insc) ** 2))
    dist_insc_min = np.min(np.sqrt((x - x_0_insc) ** 2 + (y - y_0_insc) ** 2))
    MZC_one_three = (dist_insc_max - dist_insc_min) * 1000

    # 2-2 model:
    # faccio variare i centri candidati nel rettangolo individuato dalle
coordinate x e y dei centri inscritto e circoscritto; assumo che il centro della
corona circolare sia lì dentro
    range_x_centro = np.linspace(centre[0], centro_x, len(P_X))
    range_y_centro = np.linspace(centre[1],centro_y, len(P_Y))

    diff_x = np.subtract(x, range_x_centro)
    diff_y = np.subtract(y, range_y_centro)
    r = np.sqrt(diff_x ** 2 + diff_y ** 2)
    MZC_two_two = 1000 * np.min(np.max(r) - np.min(r))

    # l'MZC finale sarà il valore minimo fra quelli calcolati secondo i 3
modelli
    MZC = min(MZC_three_one, MZC_one_three, MZC_two_two) # Here two circles are
used as reference for measuring the roundness error. One circle is drawn outside
the roundness profile just as to enclose the whole of it and the other circle is
drawn inside the roundness profile so that it just inscribes the profile. Both
circles, however, have the same center point. The roundness error here is the
difference between the radius of the two circles
    print 'errore di rotondità n°', i+1, 'secondo metodo MZC:', round(MZC, 3),
'µm'

    res.append(MZC)
    np.savetxt('MZC.txt', res)

MZC_new = np.loadtxt('MZC.txt')

aver_MZC = np.average(res);
sigma_MZC = np.std(res)
print 'MZC medio:', round(aver_MZC, 3), 'µm'
print 'sigma MZC:', round(sigma_MZC, 3), 'µm'

istogramma = sns.distplot(MZC, color = 'green')
plt.axvline(MZC.mean(), color = 'red', linewidth = 3, label = 'mean')
plt.legend()
plt.title('MZC Distribution')
plt.show()
```