21.08.2025 Statistische Kennzahlen

Donnerstag, 21. August 2025 09:36

Grundlegende Kennzahlen

- **Gesamt**: Anzahl aller Distanzwerte (inkl. NaN). Kontext für Datensatzgröße
  - `total_count = len(distances)`
- **NaN**: Anzahl ohne gültiges Ergebnis (z. B. keine Nachbarn). Weniger = besser.
  - `nan_count = int(np.isnan(distances).sum())`
    - `Np.isnan() gibt ein Boolean Array zurück - True is NaN`
    - `.sum() zählt die NaN Werte`
- **% NaN**: Anteil ungültiger Werte. Hoch = viele Abdeckungsprobleme.
  - `(nan_count / total_count) if total_count > 0 else np.nan,`
    - `If total_count > 0 -> Schutz vor Div0`
- **% Valid**: Anteil gültiger Werte (= 1 − %NaN). Hoch = robuste Abdeckung.
  - `(1 - nan_count / total_count) if total_count > 0 else np.nan,`
- **Valid Count**: Anzahl gültiger Werte (nicht NaN). Vergleichbar zu %Valid.
  - `Valid count = int(clipped.size) #.size gibt Anzahl der Elemente im Array zurück`
    - `range_override: Optionales Tupel zur expliziten Festlegung des Wertebereichs -> grundsätzlich nicht geclipped`
    - `if range_override is None: data_min, data_max = float(np.min(valid)), float(np.max(valid))`
    - `else: data_min, data_max = map(float, range_override)`
    - `clipped = valid[(valid >= data_min) & (valid <= data_max)]`
    - `valid = distances[~np.isnan(distances)]`
- **Valid Sum**: Summe aller gültigen Distanzen.
  - **Nahe 0** → die Abweichungen heben sich gegenseitig auf → es gibt keinen systematischen Versatz (Bias) zwischen den Punktwolken.
  - **Deutlich positiv** → die Vergleichsoberfläche liegt im Schnitt weiter „oben“/„außen“ als die Referenzoberfläche.
  - **Deutlich negativ** → die Vergleichsoberfläche liegt im Schnitt weiter „unten“/„innen“ als die Referenzoberfläche.
  - `valid_sum = float(np.sum(clipped))`
- **Valid Squared Sum**: Summe der Quadrate aller gültigen Distanzwerte.
  - Jeder Distanzwert d_i wird quadriert: d_i^2.
  - Danach wird alles aufsummiert.
  - Ergebnis ist immer **≥ 0** (weil Quadrate nie negativ sind).
  - Empfindlich auf große Ausreißer
  - `valid_squared_sum = float(np.sum(clipped ** 2))`

Parameter

- **Normal Scale** -> bestimmt *Richtung* der Messung (Normale).
  - **Definition:** Radius (in den Einheiten der Punktwolke), mit dem die **Normalenrichtung** an jedem Kernpunkt berechnet wird.
  - **Zu klein:** Oberflächenrauschen wird dominant → Normale flattert, Distanzwerte werden instabil.
  - **Zu groß:** Strukturen werden überglättet → lokale Details (z. B. Steine, kleine Erosionsformen) verschwinden.
  - `normal_scale,`
  - `StatisticsService._load_params(params_path)`
- **Search Scale** -> bestimmt *Ausdehnung* des Messzylinders entlang dieser Normale.
  - **Definition:** Projektions- oder Zylinder-Radius, in dem die Distanzpunkte gesammelt werden, um die Distanz entlang der Normale zu messen.
  - **Faustregel:** etwa **2× Normal Scale** → genug Punkte, aber nicht zu starker Glättung.
  - **Zu klein:** wenig oder gar keine Treffer → viele NaNs.
  - **Zu groß:** Werte werden stark geglättet, Details verschwimmen, Übergänge werden zu weich.
  - `search_scale,`
  - `StatisticsService._load_params(params_path)`

Lage- und Streuungsmaße

- **Min**: Kleinster Distanzwert. Hinweis auf Ausreißer.
  - `float(np.nanmin(distances)),`
- **Max**: Größter Distanzwert. Hinweis auf Ausreißer/große Änderungen.
  - `float(np.nanmax(distances))`
- **Mean (Bias)**: Arithmetisches Mittel (Bias). Ideal nahe 0.
  - `avg = float(np.mean(clipped))`
- **Median**: Robuster Lagewert. Weniger anfällig für Ausreißer.
  - `med = float(np.median(clipped))`
- **Std Empirical:** Standardabweichung. Maß für Streuung, empfindlich gegenüber Ausreißern.
  - `std_empirical = float(np.std(clipped))`
- **RMS**: Root Mean Square. Wurzel aus dem Mittelwert der quadrierten Distanzen.
  - $$x_{\mathrm{RMS}} = \sqrt{\frac{1}{n}\left({x_1}^2 + {x_2}^2 + \cdots + {x_n}^2\right)}.$$

- Misst die typische Abweichung unabhängig vom Vorzeichen.
- Je größer der RMS, desto „rauer“ oder stärker unterschiedlich sind die beiden Oberflächen.
- Enthält sowohl Streuung (Std) als auch systematischen Bias (Mean).

```
rms = float(np.sqrt(np.mean(clipped ** 2)))
```

- **MAE**: Mean Absolute Error
  - Statt quadrieren (wie bei RMS) wird der Betrag genommen.
  - Damit zählt jede Abweichung linear, egal ob positiv oder negativ.
  - **Robuster als Std/RMS**, weil Ausreißer nicht so stark gewichtet werden (kein Quadrat).
  - Gibt den **typischen Betrag der Abweichung** an, also die mittlere Entfernung zwischen den Oberflächen.
    - MAE = 0 → perfekte Übereinstimmung.
    - MAE = 0.01 → im Mittel weichen die Wolken um 1 cm ab (bei Einheiten in Metern).
  - $$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$
  - `mae = float(np.mean(np.abs(clipped)))`

- **NMAD**: Robuste Schätzung von σ (basierend auf MAD).
  - Weniger empfindlich gegenüber Ausreißern als Standardabweichung.
  - NMAD ~ Std, wenn die Daten annähernd normalverteilt sind.
  - `Normalized MAD: nmad = float(1.4826 * mad)`
    - (Faktor 1.4826 skaliert MAD so, dass es für eine Normalverteilung eine unverzerrte Schätzung der Standardabweichung liefert).
    - $\mathrm{NMAD} = 1.4826 \times \mathrm{MAD}$
  - `Median Absolute Deviation: mad = float(np.median(np.abs(clipped - med)))`
    - $\mathrm{MAD} = \mathrm{median}(|X_i - \tilde{X}|)$
  - `Median: med = float(np.median(clipped))`
    - $\tilde{X} = \mathrm{median}(X)$

Inlier-/Outlier-Analysen

- **MAE Inlier**: MAE ohne Ausreißer (|x| > 3·RMSE).
  - `mae_in = float(np.mean(np.abs(inliers))) if inliers.size > 0 else np.nan`
    - mittlerer absoluter Fehler **ohne** Einfluss dieser extremen Abweichungen.
  - `inliers = clipped[~outlier_mask]`
    - alle „normalen“ Punkte innerhalb des Toleranzbereichs.
  - `outlier_mask = np.abs(clipped) > (3 * rms)`
    - markiert alle Punkte, die weiter als 3 * RMS vom Nullwert entfernt liegen.
  - **Typischerweise ist MAE_in < MAE.**

- **NMAD Inlier**: NMAD ohne Ausreißer. Vergleichbar zwischen Runs.
  - Macht Ergebnisse zwischen verschiedenen Runs besser vergleichbar, weil extreme Werte keinen Einfluss haben.
  - In Kombination mit **NMAD (alle Werte)** kannst man sehen, wie stark Ausreißer die Statistik beeinflussen.

```
nmad_in = (
    float(1.4826 * np.median(np.abs(inliers - median)))
    if inliers.size > 0
    else np.nan)
```

- **Outlier Count**: Anzahl der Distanzwerte, die **außerhalb des Toleranzintervalls** |x|>3·RMS liegen.
  - Hoher Wert → viele stark abweichende Punkte (z. B. Messfehler, Vegetation, bewegte Objekte).
  - Niedriger Wert → stabile, homogene Punktwolken.

```
outlier_count,
int(outlier_mask.sum()),
outlier_mask = np.abs(clipped) > (3 * rms)
```

- **Inlier Count**: Anzahl Werte |x| ≤ 3·RMSE.
  - Anzahl der Punkte, die **innerhalb des Toleranzintervalls** |x|≤3·RMS liegen.
  - Inlier Count+Outlier Count=Valid Count

```
inlier_count,
int((~outlier_mask).sum()),
outlier_mask = np.abs(clipped) > (3 * rms)
```

- **Mean/Std Inlier**: Mittelwert bzw. Streuung ohne Ausreißer.

```
mean_in = float(np.mean(inliers)) if inliers.size else np.nan
std_in = float(np.std(inliers)) if inliers.size > 0 else np.nan
inliers = clipped[~outlier_mask]
outlier_mask = np.abs(clipped) > (3 * rms)
```

- **Mean/Std Outlier**: Mittelwert bzw. Streuung der Ausreißer.

```
mean_out = float(np.mean(outliers)) if outliers.size else np.nan
std_out = float(np.std(outliers)) if outliers.size > 0 else np.nan
outliers = clipped[outlier_mask]
outlier_mask = np.abs(clipped) > (3 * rms)
```

- **Pos/Neg Outlier**: Anzahl positiver bzw. negativer Ausreißer.
  - `pos_out = int(np.sum(outliers > 0))`

- `neg_out = int(np.sum(outliers < 0))`

- **Pos/Neg Inlier**: Verhältnis positiver/negativer Inlier.
  ```
  pos_in = int(np.sum(inliers > 0))
  neg_in = int(np.sum(inliers < 0))
  ```

Quantile

- **Q05 / Q95**:
  - Unter- und Obergrenze, innerhalb derer **90 % der Werte** liegen.
  - Robust gegen Ausreißer, da extreme 5 % abgeschnitten werden.
    ```
    "Q05": float(np.percentile(clipped, 5)),
    "Q95": float(np.percentile(clipped, 95)),
    ```

- **Q25 / Q75**: Unteres und oberes Quartil.
  - Definieren den Bereich, in dem die mittleren **50 % der Werte** liegen.
  - Differenz Q75—Q25Q75 - Q25Q75—Q25 = **Interquartilsabstand (IQR)** → robustes Streuungsmaß.
    ```
    "Q25": float(np.percentile(clipped, 25)),
    "Q75": float(np.percentile(clipped, 75)),
    ```

- `TODO: IQR!`

Verteilungs-Fits

- **Gauss Mean / Std**:
  - Nützlich, um deine Distanzverteilung mit einer **theoretischen Gauß-Verteilung** zu vergleichen.
  - `float(mu),`
    - Gauss Mean: Lageparameter der angepassten Normalverteilung.
    - Entspricht dem geschätzten „Zentrum“ der Daten.
    - Kann leicht vom empirischen Mittelwert abweichen, weil es aus einem Fit stammt.
  - `float(std),`
    - Gauss Std: Streuungsparameter der Normalverteilung.
    - Entspricht der geschätzten Standardabweichung der Daten.
    - Glättet die tatsächliche Verteilung → weniger empfindlich gegenüber kleinen Unregelmäßigkeiten.
  - `mu, std = norm.fit(clipped)`
  - `Library:` from scipy.stats import norm

- **Gauss Chi²**:
  - **Niedriger Wert** → Histogramm stimmt gut mit der Normalverteilung überein → „guter Fit“.
  - **Hoher Wert** → deutliche Abweichungen (z. B. Schiefe, Mehrgipfligkeit, fette Tails).
  - Absolute Größenordnung hängt von **Anzahl Bins** und **Datenmenge** ab → eher zum **Vergleich zwischen verschiedenen Runs** geeignet.

  - # Histogramm-Fit vorbereiten
    - `cdfL = norm.cdf(bin_edges[:-1], mu, std)`
    - `cdfR = norm.cdf(bin_edges[1:], mu, std)`
  - # Erwartete Häufigkeiten unter der Gauß-Verteilung
    - wie viele Punkte laut angepasster Gauß-Verteilung in jedem Histogramm-Bin liegen sollten.
    - `expected_gauss = N * (cdfR - cdfL)`
  - # Kleine erwartete Werte aussortieren, um Division durch 0 zu vermeiden
    - `eps = 1e-12`
    - `thr = min_expected if min_expected is not None else eps`
    - `maskG = expected_gauss > thr`
      - **Maskierung (maskG)** → Bins mit sehr kleiner erwarteter Häufigkeit (< thr) werden ignoriert, weil sie sonst das Ergebnis instabil machen.
  - # Pearson-Chi²
    - `pearson_gauss = float(np.sum((hist[maskG] - expected_gauss[maskG]) ** 2 / expected_gauss[maskG]))`
      `beobachtete Häufigkeiten N (`Hist = die tatsächlichen gezählten Punkte pro Bin)
    - `erwartete Häufigkeiten n (expected_gauss[maskG])`
    - $$X^2 = \sum_{j=1}^{m} \frac{(N_j - n_{0j})^2}{n_{0j}}$$

  Weibull Verteilung:
  https://en.wikipedia.org/wiki/Weibull_distribution
  Library: scipy.stats.weibull_min (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.weibull_min.html)

- Die **Weibull-Anpassung** ist oft besser als eine Gauß-Anpassung für stark schiefe Fehlerverteilungen (z. B. wenn Distanzen nicht normalverteilt sind).

- **Weibull a (Form)**:
  - Formparameter.
  - a<1 → schwere Tails, stark rechtsschief.
  - a≈2 → ähnlich einer Rayleigh-Verteilung.
  - a>3 → wird symmetrischer, nähert sich Normalverteilung.

  - `float(a),`
  - `a, loc, b = weibull_min.fit(clipped)`

- **Weibull b (Skala)**:
  - Skalenparameter.
  - Größer = breitere Verteilung.
  - Entspricht ungefähr einer „Streckung“ der Distanzverteilung.
  -
  - `float(b),`

    - `a, loc, b = weibull_min.fit(clipped)`

- **Weibull shift**:
    - Verschiebung der Verteilung entlang der Achse.
    - In CloudCompare oft ≈ Median/Min, je nachdem wie die Daten liegen.

    - `float(loc),`
    - `a, loc, b = weibull_min.fit(clipped)`

- **Weibull mode**:
    - Stelle des Maximums der Dichtefunktion.
    - `mode_weibull = float(loc + b * ((a - 1) / a) ** (1 / a)) if a > 1 else float(loc)`

- **Weibull skewness**: Schiefe. Positiv = Rechts-Tail; negativ = Links-Tail.
    - `skew_weibull = float(weibull_min(a, loc=loc, scale=b).stats(moments="s"))`

- **Weibull Chi²**: Pearson-Chi² zum Weibull-Fit. Nur relativ interpretieren.
    - # Erwartete Häufigkeiten unter der Weibull-Verteilung
        - `cdfL = weibull_min.cdf(bin_edges[:-1], a, loc=loc, scale=b)`
        - `cdfR = weibull_min.cdf(bin_edges[1:], a, loc=loc, scale=b)`

    - `expected_weib = N * (cdfR - cdfL)`

    - # Kleine erwartete Klassen ausschließen
        - `maskW = expected_weib > thr`

    - # Pearson-Chi² für Weibull
        - `pearson_weib = float(np.sum((hist[maskW] - expected_weib[maskW]) ** 2 / expected_weib[maskW]))`

Verteilungscharakteristika

- **Library:** import pandas as pd

- **Skewness**: Schiefe.
    - Maß für die **Asymmetrie** der Verteilung.
    - ~0 → symmetrisch;
    - >0 → rechtsschief (lange rechte Flanke)
    - <0 → linksschief (lange linke Flanke)
    - |Skew|>1|→ starke Asymmetrie.
    - `float(pd.Series(clipped).skew()),`

- **Kurtosis**: Exzess-Kurtosis. .
    - Maß für die „Spitzigkeit“ / „Tails“ einer Verteilung
    - 0 = normal
    - > 0 = schwere Tails
    - < 0 = leichte Tails
    - `float(pd.Series(clipped).kurt()),`

Toleranz-/Abdeckungsmaße

- **Anteil |Distanz| > 0.01**:
    - Prozentsatz der Punkte, die mehr als 1 cm Abweichung haben.
    - Robust und leicht interpretierbar.
    - Niedriger = bessere Übereinstimmung.

    - `float(np.mean(np.abs(clipped) > 0.01)),`

- **Anteil [-2Std, 2Std]**: Anteil innerhalb ±2σ um 0. Ideal ~95% (bei normalverteilten, biasfreien Daten).
    - Zeigt, wie viele Werte innerhalb des 95%-Intervalls einer Normalverteilung liegen.
    - Wenn ≈95 % → Daten sind fast normalverteilt und ohne Bias.
    - Deutlich kleiner → viele Ausreißer oder Schiefe.

    - `float(np.mean((clipped > -2*std) & (clipped < 2*std))),`

- **Max |Distanz|**: Größter Absolutwert. Hinweis auf Extreme.
    - Extremwert der Abweichungen.
    - Nützlich für „worst case“-Betrachtung, kann aber durch Ausreißer stark verzerrt sein.

    - `float(np.max(np.abs(clipped))),`

- **Within-Tolerance**: Anteil innerhalb einer definierten Toleranz (z. B. ±1 cm).
    - Anteil der Punkte, die in der zulässigen Toleranz liegen.
    - Praktisches Qualitätsmaß (gerade für Ingenieur- oder Bauanwendungen).
    - Flexibel, da tolerance frei wählbar (z. B. 0.005 = 5 mm, 0.02 = 2 cm).

    - `within_tolerance = float(np.mean(np.abs(clipped) <= tolerance))`
    - `Default: tolerance: float = 0.01`

Vergleichs- und Übereinstimmungsmaße

- **ICC**: Intraklassen-Korrelation (hier nur Platzhalter).
    - `icc = np.nan  # Placeholder`

- **CCC**: Concordance Correlation Coefficient.
- Das ist *nicht der echte Lin's CCC*, sondern eine pragmatische Ersatzkennzahl → also gut für **relative Vergleiche zwischen Runs**, aber keine echte statistische Maßzahl.
  - `ccc = (2 * mean_dist * std_dist) / (mean_dist**2 + std_dist**2) if mean_dist != 0 else np.nan`
  - `mean_dist = float(np.mean(clipped))`
  - `std_dist = float(np.std(clipped))`
  - `TODO: Distanzwerte zwischen ref und ref_ai Output vergleichen!`
  - `Bzw. Tunnel Outputs vergleichen`

- **Bland-Altman Lower/Upper**: Untere und obere Übereinstimmungsgrenzen (Bias ± 1.96·σ).
  - https://en.wikipedia.org/wiki/Bland%E2%80%93Altman_plot
  - https://pmc.ncbi.nlm.nih.gov/articles/PMC4470095/
  - Limits of Agreement (LoA): Bias±1.96·Std
  - **Schmale LoA** → hohe Übereinstimmung / geringe Streuung.
  - **Breite LoA** → viele Abweichungen / unpräzise Übereinstimmung.
  - Vergleichbar über verschiedene Runs (z. B. Python vs. CloudCompare).

  - `bland_altman_lower = bias - 1.96 * std_dist`
  - `bland_altman_upper = bias + 1.96 * std_dist`

- **Jaccard Index / Dice Coefficient**: In aktueller Implementierung identisch zu Within-Tolerance.
  - `jaccard_index = intersection / union if union > 0 else np.nan`
    - Anteil innerhalb Toleranz = identisch zu within_tolerance.
    - `intersection = np.sum((clipped > -tolerance) & (clipped < tolerance))`
      - Anzahl Punkte, die innerhalb der Toleranz liegen.
    - `union = len(clipped)`
      - Gesamtanzahl der Punkte.

  - `dice_coefficient = (2 * intersection) / (2 * union) if union > 0 else np.nan`
    - gleiche Formel, da hier keine zwei unterschiedlichen Mengen kombiniert werden.

  - Jaccard & Dice sind **Set-Similarity-Maße** zwischen zwei Mengen A,BA, BA,B:
    - $$J(A,B) = \frac{|A \cap B|}{|A \cup B|}, \quad D(A,B) = \frac{2|A \cap B|}{|A| + |B|}$$