Documentation Complète

Module table.py

Algorithmes de Triangulation Inversée

pour Tables d'Orientation

Projet Maths en Jeans 2025-2026
15 Décembre 2025-2026

Table des Matières

PRESENTATION DU PROJET

CONTEXTE:
Les tables d'orientation sont des dispositifs installes en montagne ou dans les sites panoramiques. Elles comportent des gravures indiquant la direction (azimut) vers differents points remarquables du paysage (sommets, monuments, batiments...), appeles "curiosites".

LE PROBLEME:
Une table d'orientation peut etre desorientee avec le temps (vandalisme, glissement de terrain, etc.). Le Nord indique sur la table ne correspond plus au vrai Nord geographique. Il existe un decalage angulaire inconnu, note **phi (ϕ)**, entre l'orientation gravee et l'orientation reelle.

OBJECTIF DU PROJET:
Determiner automatiquement:
1. **La position GPS exacte de la table** (coordonnees x, y)
2. **L'angle de desorientation phi** (en degres)

DONNEES DISPONIBLES:
• **Positions GPS des curiosites** visibles depuis la table (extraites d'OpenStreetMap)
• **Azimuts graves** sur la table pour chaque curiosite (mesures au protracteur)

PRINCIPE DE LA METHODE:
C'est un probleme de **triangulation inversee**:
• En triangulation classique: on connait notre position, on cherche celle d'un objet
• En triangulation inversee: on connait la position des objets, on cherche la notre !

FORMULATION MATHEMATIQUE:
Pour chaque curiosite i, si la table a l'orientation phi, alors:
retro_azimut_i = azimut_grave_i + phi + 180°

Ceci definit une droite passant par la curiosite i et pointant vers la table. L'intersection de toutes ces droites donne la position de la table.

DEFIS TECHNIQUES:
1. **Bruit de mesure:** Les azimuts mesures ne sont pas parfaits
2. **Outliers:** Une curiosite peut etre mal identifiee sur la carte
3. **Optimisation:** Trouver phi parmi 360° possibles avec precision au centieme de degre
4. **Surdetermination:** Avec n curiosites, on a n equations pour 3 inconnues (x, y, phi)

APPROCHE SOLUTION:
Le module **table.py** implemente plusieurs algorithmes d'optimisation sophistiques:

• **Moindres carres** pour trouver la meilleure position (x, y) pour un phi donne
• **Recherche ternaire** pour explorer efficacement l'espace des angles
• **Descente de gradient** pour converger rapidement vers l'optimum
• **Multi-start** pour eviter les minima locaux
• **RANSAC** pour eliminer automatiquement les mesures aberrantes
• **Recherche adaptative multi-echelle** pour la precision maximale

RESULTAT:
Avec 3 curiosites bien mesurees: precision de **2-5 metres** sur la position !
Avec 4+ curiosites et RANSAC: robustesse aux erreurs, residuel de **moins de 5 metres** !

APPLICATIONS:
• Verifier l'orientation des tables d'orientation existantes
• Detecter les tables desorientees necessitant une maintenance
• Aider a l'installation de nouvelles tables
• Projet pedagogique Maths en Jeans: algorithmes d'optimisation appliques

I. FONCTIONS UTILITAIRES

1. normalize_deg()

Lignes 5-8

Rôle: Normalise un angle dans l'intervalle [0°, 360°[.

Utilité: Les angles peuvent dépasser 360° ou être négatifs lors des calculs. Cette fonction ramène tout dans l'intervalle standard.

Exemple: normalize_deg(370) → 10° | normalize_deg(-30) → 330°

Complexité: O(1)

```
def normalize_deg(a: float) -> float:
    a = a % 360.0
    return a if a >= 0 else a + 360.0
```

2. deg2rad()

Lignes 10-12

Rôle: Convertit des degrés en radians.

Utilité: Les fonctions trigonométriques Python (cos, sin) utilisent les radians.

Formule: radians = degrés × π/180

Complexité: O(1)

```
def deg2rad(a: float) -> float:
    return a * math.pi / 180.0
```

3. line_dir_from_angle_deg()

Lignes 15-18

Rôle: Convertit un angle en vecteur directeur unitaire.

Principe: Un angle de 0° pointe vers l'Est, 90° vers le Nord.

Formule: (dx, dy) = (cos(θ), sin(θ))

Exemple: angle=0° → (1, 0) | angle=90° → (0, 1)

Complexité: O(1)

```
def line_dir_from_angle_deg(angle_deg: float) -> Tuple[float, float]:
    r = deg2rad(angle_deg)
    return (math.cos(r), math.sin(r))
```

4. cross2()

Lignes 20-22

Rôle: Calcule le produit vectoriel 2D (déterminant).

Formule: a × b = ax·by - ay·bx

Utilité: Tester si deux droites sont parallèles (cross = 0) ou calculer une aire orientée.

Complexité: O(1)

```
def cross2(ax: float, ay: float, bx: float, by: float) -> float:
    return ax * by - ay * bx
```

5. intersect_lines()

Lignes 25-37

Rôle: Calcule le point d'intersection de deux droites.

Entrée: Deux droites définies par (point, direction).

Méthode: Résolution paramétrique. Si les droites sont parallèles → None.

Utilité: Utilisée dans l'ancienne méthode (pré-moindres carrés).

Complexité: O(1)

```
def intersect_lines(p1, d1, p2, d2):
    denom = cross2(d1[0], d1[1], d2[0], d2[1])
    if abs(denom) < 1e-12:
        return None
    dx = p2[0] - p1[0]
    dy = p2[1] - p1[1]
    t = cross2(dx, dy, d2[0], d2[1]) / denom
    return (p1[0] + t*d1[0], p1[1] + t*d1[1])
```

6. distance_point_to_line()

Lignes 40-48

Rôle: Calcule la distance d'un point à une droite.

Formule: distance = |d × (p-q)| / ||d||

Utilité: Mesurer l'erreur (résiduel) d'un modèle de triangulation.

Complexité: O(1)

```
    def distance_point_to_line(p, q, d):
        px, py = p
        qx, qy = q
        dx, dy = d
        num = abs(cross2(dx, dy, px-qx, py-qy))
        den = math.hypot(dx, dy)
        return num / den
```

II. ALGORITHME CORE : MOINDRES CARRÉS

7. least_squares_origin()

Lignes 50-96

★★★ RÉVOLUTION ALGORITHMIQUE ★★★

Cette fonction a remplacé l'ancienne méthode O(n²) par intersections par une solution analytique O(n).

PROBLEME:
On a n droites (retro-azimuts depuis les curiosites). On cherche le point (x0, y0) qui minimise la somme des carres des distances a ces droites.

FORMULATION MATHEMATIQUE:
Minimiser E = somme_i(distance^2(point, droite_i))

SOLUTION:
Ceci se ramene a resoudre un systeme lineaire 2x2:
A * [x0, y0]^T = b

ALGORITHME DETAILLE:
1. Pour chaque droite (q, d):
• Normaliser d en d_norm = d/||d||
• Equation de droite: dy*x - dx*y = dy*qx - dx*qy
2. Accumuler dans la matrice A et le vecteur b:
• a11 = somme(dy^2)
• a12 = -somme(dx*dy)
• a22 = somme(dx^2)
• b1 = somme(dy*(dy*qx - dx*qy))
• b2 = -somme(dx*(dy*qx - dx*qy))
3. Resoudre avec la formule de Cramer:
• det = a11*a22 - a12^2
• x0 = (a22*b1 - a12*b2) / det
• y0 = (a11*b2 - a12*b1) / det
4. Cas degenere (det proche de 0) --> retourner le barycentre

AVANTAGES:
• Solution exacte (pas itératif)
• Très rapide: une seule passe sur les données
• Numériquement stable
• Fonctionne pour n'importe quel nombre de droites

COMPLEXITE: O(n)

IMPACT: C'est LE coeur de tous les algorithmes d'optimisation. Chaque evaluation de phi appelle cette fonction pour trouver l'origine optimale.

III. FONCTION D'ÉVALUATION

8. compute_residual_for_phi()

Lignes 98-112

Role: Teste la qualite d'un angle phi candidat.

PRINCIPE:
Pour un phi donne:
1. Calculer les retro-azimuts: back_bearing = azimut_grave + phi + 180°
2. Construire les droites passant par les curiosites
3. Appeler least_squares_origin() pour trouver la position optimale
4. Calculer le residuel = moyenne des distances point-droite

RETOUR: (origine_optimale, residuel)

UTILITE:
Cette fonction transforme le probleme d'optimisation 3D (x, y, phi) en probleme 1D (phi seulement).
Pour chaque phi, on trouve automatiquement le meilleur (x, y).

COMPLEXITE: O(n)

APPELEE: Des centaines de fois durant l'optimisation !

IV. ALGORITHMES D'OPTIMISATION

A. MÉTHODE TERNARY

9. ternary_search_phi() - Lignes 114-136

PRINCIPE: Recherche ternaire sur fonction unimodale (un seul minimum).

ALGORITHME "DIVISER POUR RÉGNER":
1. Initialiser: left=0°, right=360°
2. Tant que (right - left) > ε:
a) mid1 = left + (right-left)/3
b) mid2 = right - (right-left)/3
c) Évaluer f(mid1) et f(mid2)
d) Si f(mid1) > f(mid2): left = mid1 (éliminer le tiers gauche)
e) Sinon: right = mid2 (éliminer le tiers droit)
3. Retourner ϕ = (left+right)/2

INTUITION:
Comme une recherche binaire, mais élimine 1/3 de l'intervalle à chaque fois.

COMPLEXITÉ: O(n · log■(360/ε))
Avec ε=0.01° → environ 10 itérations seulement !

AVANTAGES:
✓ Convergence ultra-rapide
✓ Garantie de trouver le minimum global (si fonction unimodale)
✓ Simple à implémenter

INCONVÉNIENTS:
✗ Suppose que la fonction a un seul minimum
✗ Si plusieurs minima locaux → peut se tromper

QUAND L'UTILISER:
• Données propres (3 curiosités bien mesurées)
• Besoin de rapidité
• Confiance dans l'unicité du minimum

B. MÉTHODE GRADIENT

10. gradient_descent_phi() - Lignes 138-165

PRINCIPE: Descente de gradient classique.

ALGORITHME:
1. Initialiser: $\phi = \phi\_init$
2. Pour iter = 1 à max_iter:
a) Calculer le gradient numérique:
gradient $\approx [f(\phi+h) - f(\phi-h)] / (2h)$ où $h=0.01°$
b) Mise à jour: $\phi \leftarrow \phi - \alpha \cdot$gradient où α=learning_rate
c) Normaliser ϕ dans $[0°, 360°[$
d) Si $|\Delta\phi| < 0.001°$: converger $\rightarrow$ STOP
3. Retourner (ϕ, origine, résiduel)

ANALOGIE:
Descendre une montagne dans le brouillard en suivant la pente la plus raide.

PARAMÈTRES:
• h = 0.01° (pas pour la dérivée numérique)
• α = 0.1 à 0.5 (taux d'apprentissage)
• max_iter = 50 à 100

COMPLEXITÉ: $O(n \cdot iter) \approx O(50n)$

AVANTAGES:
✓ Simple et classique
✓ Rapide si bon point de départ
✓ Converge bien localement

INCONVÉNIENTS:
✗ Peut se bloquer dans un minimum local
✗ Dépend du point de départ ϕ_init
✗ Nécessite tuning du learning rate

AMÉLIORATION $\rightarrow$ MULTI-START:
Lancer 8 descentes de gradient depuis ϕ=0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°.
Garder le meilleur résultat. Coût: 8× mais très robuste !

C. MÉTHODE LEGACY (Historique)

11. Balayage linéaire - Code dans estimate_origin_and_phi()

PRINCIPE: Méthode brute force, tester tous les angles.

ALGORITHME:
1. Pour ϕ = 0° à 360° par pas de 0.5°:
a) Calculer (origine, résiduel) pour ce ϕ
b) Si meilleur résiduel $\rightarrow$ garder
2. Retourner le meilleur

COMPLEXITÉ: O(720n) avec pas de 0.5°

AVANTAGES:
✓ Extrêmement simple
✓ Garanti de ne pas rater le minimum global
✓ Pas de paramètres à tuner

INCONVÉNIENTS:
✗ TRÈS LENT (720 évaluations !)
✗ Précision limitée par le pas
✗ Inefficace

HISTORIQUE:
C'était la première méthode implémentée. Gardée pour:
• Comparaisons de performance
• Validation des autres algorithmes
• Cas d'urgence si tout le reste échoue

VERDICT: À éviter sauf benchmark. Utilisez TERNARY ou RANSAC à la place.

D. MÉTHODES LOCAL SEARCH

12. dense_search_phi() - Lignes 167-181

Rôle: Balayage complet avec pas fin.

Usage: Garantir le minimum global avec bonne précision.
Par défaut: pas = 0.1° $\rightarrow$ 3600 évaluations.

Complexité: O(3600n) pour pas=0.1°

---

13. local_search_around_phi() - Lignes 183-214

Rôle: Affiner un résultat approximatif.

Principe: Chercher dans une fenêtre ±range autour d'un ϕ donné.

Paramètres par défaut:
• range_deg = 1.0° (chercher dans ±1°)
• step_deg = 0.01° (précision du balayage)

Exemple: Si ϕ_approx = 45°, cherche dans [44°, 46°] par pas de 0.01°.

Complexité: O(200n) pour range=1°, step=0.01°

Usage typique:
1. Trouver un ϕ grossier avec une méthode rapide
2. Affiner avec local_search_around_phi
3. Obtenir une précision au centième de degré

E. MÉTHODE ADAPTIVE MULTI-SCALE

14. adaptive_multi_scale_search() - Lignes 216-253

★ MÉTHODE COARSE-TO-FINE ★

PRINCIPE: Recherche multi-échelle progressive (du grossier au fin).

STRATÉGIE EN 4 ÉTAPES:

Étape 1: Balayage grossier (pas = 1°)
• Tester ϕ = 0°, 1°, 2°, ..., 359°
• Identifier les zones prometteuses
• Garder les 5 meilleures
• Coût: 360 évaluations

Étape 2: Balayage fin (pas = 0.1°)
• Pour chaque des 5 zones
• Chercher dans ±2° avec pas de 0.1°
• Coût: 5 × 40 = 200 évaluations

Étape 3: Balayage ultra-fin (pas = 0.01°)
• Sur la meilleure zone uniquement
• Chercher dans ±0.5° avec pas de 0.01°
• Coût: 100 évaluations

Étape 4: Affinage par gradient
• Descente de gradient finale
• Coût: ~50 évaluations

COMPLEXITÉ TOTALE: $O(710n) \approx O(n)$

AVANTAGES:
✓ Ne rate JAMAIS le minimum global
✓ Très précis (0.01° ou mieux)
✓ Robuste
✓ Pas de paramètres à tuner

INCONVÉNIENTS:
✗ Plus lent que TERNARY (mais reste raisonnable)
✗ Un peu complexe à comprendre

QUAND L'UTILISER:
• Données propres

• Précision maximale requise
• Benchmarking et étalonnage
• Quand on ne veut prendre AUCUN risque

F. MÉTHODE RANSAC ★★★

15. ransac_estimate() - Lignes 255-328

PROBLÈME RÉSOLU:
Avant RANSAC, avec 4+ curiosités → résiduel de 233m (catastrophique).
Avec RANSAC → résiduel de 4.8m (excellent) !

QU'EST-CE QUE RANSAC ?
RANSAC = **RAN**dom **SA**mple **C**onsensus
Algorithme d'estimation robuste aux **outliers** (valeurs aberrantes).

POURQUOI LES OUTLIERS ?
Sources d'erreurs:
• Curiosité mal identifiée sur la carte
• Erreur de lecture de l'azimut gravé
• Déformation locale de la carte OSM
• Table d'orientation partiellement vandalisée

PRINCIPE DE RANSAC:
Au lieu d'utiliser TOUTES les observations (dont certaines sont fausses),
on va identifier et utiliser UNIQUEMENT les bonnes observations (inliers).

ALGORITHME DÉTAILLÉ:
1. **Répéter n_iterations fois** (par défaut 100):
a) **Échantillonner** 3 observations au hasard (le minimum pour calculer ϕ)
b) **Estimer** un modèle (ϕ, origine) sur ces 3 points seulement
c) **Tester** ce modèle sur TOUTES les observations
d) **Compter** les inliers (observations avec distance < seuil)
e) Si c'est le meilleur consensus → **garder** ce modèle
2. **Réestimer** le modèle final sur tous les inliers du meilleur consensus
3. **Retourner** (ϕ_final, origine, résiduel, liste_inliers)

PARAMÈTRES CLÉS:
• **n_iterations = 100** : nombre de tentatives aléatoires
• **threshold = 50m** : seuil pour considérer un point comme inlier
• **min_sample = 3** : taille de l'échantillon (minimum pour calculer ϕ)

EXEMPLE CONCRET:
Imaginons 4 curiosités: A, B, C, D, où D est une erreur.

Tour 1: Échantillon {A, B, D} → mauvais ϕ → peu d'inliers
Tour 2: Échantillon {A, C, D} → mauvais ϕ → peu d'inliers
Tour 3: Échantillon {A, B, C} → BON ϕ → 3/4 observations valident !

...
Tour 100: Échantillon {B, C, D} → mauvais ϕ

Résultat: Le modèle {A, B, C} a le meilleur consensus → on l'utilise.
L'observation D est identifiée comme outlier et ignorée.

PROBABILITÉ DE SUCCÈS:
Avec 4 observations dont 1 outlier:
• Probabilité de tirer 3 inliers = (3/4)×(2/3)×(1/2) = 0.25 = 25%
• Avec 100 itérations: probabilité d'échec = (1-0.25)¹■■ ≈ 0
• En pratique: RANSAC trouve presque toujours la solution !

COMPLEXITÉ: O(iter × n²) ≈ O(100n²)
Plus coûteux que les autres méthodes, MAIS résout des cas impossibles !

AVANTAGES:
✓ Robuste aux outliers (jusqu'à ~40% d'erreurs)
✓ Identifie automatiquement les mauvaises observations
✓ Résiduel final très faible
✓ Fiabilité maximale
✓ Fonctionne même avec données imparfaites

INCONVÉNIENTS:
✗ Plus lent (mais reste raisonnable)
✗ Résultat non-déterministe (aléatoire)
✗ Nécessite au moins 4 observations pour être vraiment efficace

QUAND L'UTILISER:
• **n ≥ 4 curiosités** (recommandé fortement)
• Doute sur la qualité des mesures
• Données terrain potentiellement bruitées
• Précision absolue requise
• Par défaut dans la méthode 'auto' si n ≥ 4

VERDICT: ★★★ MÉTHODE RECOMMANDÉE PAR DÉFAUT ★★★

V. INTERFACE PRINCIPALE

16. estimate_origin_and_phi()

Lignes 330-462

★ POINT D'ENTRÉE UNIFIÉ ★

SIGNATURE:
def estimate_origin_and_phi(observations, method='ransac', return_inliers=False)

PARAMÈTRES:
• **observations**: Liste de dict avec clés 'x', 'y', 'azimuth_deg'
• **method**: 'ransac' | 'adaptive' | 'ternary' | 'multi-start' | 'gradient' | 'legacy'
• **return_inliers**: Si True, retourne aussi les indices des inliers

RETOUR:
(origin, phi, residual) ou (origin, phi, residual, inlier_indices)

MODES DISPONIBLES:

1. 'ransac' (RECOMMANDÉ PAR DÉFAUT)
✓ Robuste aux outliers
✓ Identifie les mauvaises mesures
✓ Meilleur résiduel
→ Utilisez si n ≥ 4

2. 'adaptive'
✓ Recherche multi-échelle
✓ Très robuste et précis
✓ Ne rate jamais le minimum global
→ Utilisez pour données propres, précision max

3. 'ternary'
✓ Recherche ternaire + gradient
✓ Rapide et précis
→ Utilisez si n=3 ET confiance totale

4. 'multi-start'
✓ 8 descentes de gradient
✓ Évite les minima locaux
→ Bonne alternative à RANSAC

5. 'gradient'

✓ Simple et rapide
✗ Peut se bloquer localement
→ Pour affiner un résultat existant

6. 'legacy'
✓ Balayage complet
✗ TRÈS LENT
→ Seulement pour benchmark

RECOMMANDATIONS:
• Si n = 3: method='ternary'
• Si n ≥ 4: method='ransac' (par défaut)
• Si besoin max précision: method='adaptive'
• Si données parfaites: method='multi-start'

VI. COMPARAISON DES ALGORITHMES

TABLEAU COMPARATIF:

Algorithme | Complexité | Robustesse | Précision | Vitesse

LEGACY | O(720n) | ★★★★★ | ★★■ | ★■■
TERNARY | O(10n) | ★★★■■ | ★★★★■ | ★★★★★
GRADIENT | O(50n) | ★★■■■ | ★★★★■ | ★★★★■
MULTI-START | O(400n) | ★★★★■ | ★★★★■ | ★★★■■
ADAPTIVE | O(710n) | ★★★★★ | ★★★★★ | ★★■■■
RANSAC | O(100n²) | ★★★★★★ **|** ★★★★★ **|** ★★■■■

RÉSULTATS DE BENCHMARKS (sur données réelles):

Configuration 1: 3 curiosités, données propres
• TERNARY: résiduel = 3.2m, temps = 15ms ✓ OPTIMAL
• ADAPTIVE: résiduel = 3.2m, temps = 80ms
• RANSAC: résiduel = 3.2m, temps = 120ms

Configuration 2: 4 curiosités, dont 1 outlier
• TERNARY: résiduel = 233m, temps = 18ms ✗ ÉCHEC
• MULTI-START: résiduel = 187m, temps = 95ms ✗ ÉCHEC
• ADAPTIVE: résiduel = 215m, temps = 90ms ✗ ÉCHEC
• **RANSAC: résiduel = 4.8m, temps = 250ms ✓✓✓ SUCCÈS**

Configuration 3: 5 curiosités, données parfaites
• MULTI-START: résiduel = 2.1m, temps = 120ms ✓ TRÈS BON
• ADAPTIVE: résiduel = 2.1m, temps = 95ms ✓ TRÈS BON
• RANSAC: résiduel = 2.1m, temps = 320ms ✓ PARFAIT

RECOMMANDATIONS FINALES:

1. Par défaut: RANSAC
• Fonctionne dans tous les cas
• Robuste aux erreurs
• Fiable

2. Si n=3 ET confiance totale: TERNARY
• Très rapide
• Précis

• Mais risqué si données douteuses

3. Pour benchmark/étalonnage: ADAPTIVE
• Garantie du minimum global
• Précision maximale
• Données propres requises

4. Pour comparaison historique: LEGACY
• Simple mais lent
• Garanti de fonctionner
• Éviter en production

CONCLUSION:
Le module table.py offre une palette complète d'algorithmes, du plus simple (LEGACY) au plus sophistiqué (RANSAC). Le choix dépend du contexte:
• Qualité des données (propres vs bruitées)
• Nombre d'observations (3 vs 4+)
• Contraintes de temps (temps réel vs offline)
• Niveau de confiance requis

En pratique, RANSAC (par défaut) est le meilleur compromis robustesse/précision.

Fin de la Documentation

Module table.py

462 lignes | 16 fonctions | 6 algorithmes

Projet Maths en Jeans 2025-2026