PROJET D'EXAMEN · MULTI-LANGAGES

Projet 1
Station de mesures physiques

Système de monitoring de capteurs environnementaux

Domaine :	Physique · Capteurs · IoT
Composantes :	C (DLL) · Python (module + tests) · Flask (API + dashboard) · C# (GUI)
Cours :	5e TTR Informatique · 6e TTR Informatique
Format :	Projet individuel
Édition :	19 mai 2026

1 Contexte et objectifs

Tu construis un système de monitoring qui collecte des mesures de capteurs physiques (température, pression atmosphérique, humidité), les stocke, détecte les anomalies, et présente les résultats sur un dashboard temps réel. Cas d'usage typiques : station météo amateur, serre intelligente, laboratoire scolaire, monitoring d'un local technique.

Objectifs pédagogiques

- Compiler une bibliothèque C en DLL et l'appeler depuis Python via `ctypes`.
- Écrire un module Python testable, avec validation et persistance des données.
- Concevoir une API REST Flask et un dashboard Chart.js qui la consomme.
- Développer une interface C# Windows Forms qui communique avec l'API.
- Comparer trois approches de calcul (Python pur, ctypes scalaire, ctypes batch).

2 Cahier des charges fonctionnel

L'application doit permettre à l'utilisateur de :

- Saisir une nouvelle mesure (type, valeur, unité, horodatage automatique).
- Lister les mesures avec filtre par type et par plage de dates.
- Calculer et afficher les statistiques (moyenne, écart-type, min, max) par type.
- Détecter automatiquement les valeurs aberrantes (hors plages physiques).
- Convertir une mesure entre différentes unités (°C ↔ °F, hPa ↔ mmHg).
- Visualiser l'évolution temporelle sur un graphique Chart.js.
- Exporter les données filtrées au format CSV.

> **Plages physiques de validation —** temperature: [-50 °C ; 100 °C] · pressure: [800 ; 1100] hPa · humidity: [0 ; 100] %

3 Module C — `calculs.c` → `calculs.dll`

Les fonctions ci-dessous sont à exporter dans la DLL. Chaque fonction doit être présente en version **scalaire** (pour l'application normale) ; les fonctions marquées *batch* servent au benchmark de la Partie 5.

Signature	Rôle
`float to_fahrenheit(float celsius);`	Conversion °C → °F
`float to_celsius(float fahrenheit);`	Conversion °F → °C
`float hpa_to_mmhg(float hpa);`	Conversion hPa → mmHg
`int is_anomaly(float value, float min, float max);`	1 si hors plage, 0 sinon
`float normalize_value(float v, float min, float max);`	Normalise dans [0, 1]
`float mean(float* values, int n);`	Moyenne arithmétique (batch)
`float std_deviation(float* values, int n);`	Écart-type (batch)
`void to_fahrenheit_batch(float* in, float* out, int n);`	Conversion batch
`void is_anomaly_batch(float* v, float min, float max, int* out, int n);`	Détection batch

Compilation Windows : `gcc -shared -o calculs.dll calculs.c -lm`

4 Module Python — `module.py`

Le module Python est le seul pont entre la DLL et le reste de l'application. Il contient les wrappers `ctypes` et la logique métier (validation, persistance, statistiques).

- `to_fahrenheit(c: float) -> float — wrapper ctypes`
- `is_anomaly(value: float, sensor_type: str) -> bool — détection métier`
- `validate_measure(value: float, sensor_type: str) -> bool`
- `save_measure(data: dict) -> dict — ajoute id + timestamp`
- `load_all() -> list[dict]`
- `load_by_type(sensor_type: str) -> list[dict]`
- `get_stats(sensor_type: str) -> dict — {mean, std, min, max, count}`
- `detect_anomalies(sensor_type: str) -> list[dict]`

5 API Flask — `app.py`

Toutes les réponses respectent le format `{"status": "ok|error", "data": ...}`. Statuts HTTP : 200 (lecture), 201 (création), 400 (validation), 404 (non trouvé), 500 (erreur serveur).

Méthode	Chemin	Description
GET	`/api/measures`	Liste paginée (filtrage par ?type=&from=&to=)
GET	`/api/measures/{id}`	Détail d'une mesure
POST	`/api/measures`	Ajoute une mesure (validation serveur)
GET	`/api/measures/stats`	Statistiques par type
GET	`/api/measures/anomalies`	Liste des mesures anormales
DELETE	`/api/measures/{id}`	Supprime une mesure

6 Dashboard web — `/dashboard`

Dashboard lecture seule, rafraîchissement automatique toutes les 30 secondes via fetch JS. Trois graphiques Chart.js obligatoires :

1. Évolution temporelle (line chart) : une courbe par type, axe X = temps.
2. Distribution (bar chart) : nombre de mesures par type sur la période.
3. Anomalies (KPI + doughnut) : compteur + camembert des anomalies par capteur.

7 Interface C# — Windows Forms

Application Windows Forms qui communique avec l'API uniquement via `HttpClient`. Aucun calcul ne doit être fait côté C#.

- MainForm — liste filtrable (DataGridView avec tri/recherche).
- AddMeasureForm — formulaire d'ajout (validation locale + POST API).
- StatsForm — tableau de statistiques par type, bouton « Recalculer ».
- ConvertForm — popup de conversion d'unités (offline, via module Python).

8 Format des données

Variante 5e — Persistance JSON

```
{
  "id": 42,
  "type": "temperature",
  "value": 22.5,
  "unit": "°C",
  "timestamp": "2026-05-19T10:30:00",
  "is_anomaly": false
}
```

Variante 6e — Persistance SQL (MySQL/SQLite)

```
CREATE TABLE measures (
  id         INTEGER PRIMARY KEY AUTOINCREMENT,
  type       TEXT NOT NULL CHECK (type IN ('temperature','pressure','humidity')),
  value      REAL NOT NULL,
  unit       TEXT NOT NULL,
  timestamp  TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
  is_anomaly INTEGER DEFAULT 0
);
CREATE INDEX idx_measures_type      ON measures(type);
CREATE INDEX idx_measures_timestamp ON measures(timestamp);
```

9 Plan de tests Pytest

Minimum 10 tests Pytest dans `tests_module.py`. Tous doivent retourner **PASSED**.

1. `test_to_fahrenheit_zero_celsius — 0 °C doit donner 32 °F`
2. `test_to_fahrenheit_100 — 100 °C doit donner 212 °F`
3. `test_to_celsius_inverse — round-trip C → F → C doit redonner la valeur`
4. `test_is_anomaly_in_range — valeur valide retourne False`
5. `test_is_anomaly_below_min — valeur < min retourne True`
6. `test_is_anomaly_above_max — valeur > max retourne True`
7. `test_validate_measure_temperature_valid`
8. `test_validate_measure_extreme_outside`
9. `test_save_and_load_roundtrip — POST puis GET retrouve la mesure`
10. `test_get_stats_average — moyenne sur valeurs connues`

10 Critères de validation et remise

Barème (sur 24)

Composante	Points
Module C — fonctions scalaires + batch, compile sans warning	4
Module Python — wrapper ctypes + logique métier	4
Tests Pytest — 10 tests PASSED	3
API Flask — tous les endpoints fonctionnels	4
Dashboard web — 3 graphiques opérationnels	3
GUI C# — communication API et écrans demandés	4
Benchmark — 3 scénarios + tableau + explication orale	2
Total	**/24**

Structure du dossier de remise

```
projet-1-station_mesures/
├── calculs.c
├── calculs.dll
├── module.py
├── tests_module.py
├── app.py
├── generator.py
├── benchmark.py
├── templates/
│   └── dashboard.html
├── data/                        (5e — JSON)   ou   db.sqlite   (6e — SQL)
└── GUI/
    └── Projet1.sln
```

Démonstration orale (5 minutes)

1. Lance ton API Flask et ton dashboard.
2. Lance le générateur de données et observe le dashboard se remplir.
3. Ouvre la GUI C# et démontre 2 cas d'usage.
4. Présente le résultat du benchmark et explique pourquoi le scénario batch est plus rapide.