IINTS-AF SDK Technical Reference Manual

IINTS-AF SDK Technical Reference Manual Version 0.1.17 | Python SDK
PRE-CLINICAL USE ONLY - NOT FOR PATIENT CARE
This SDK is intended for research, simulation, and algorithm validation. It has NOT received FDA clearance or CE marking for clinical use.

Table of Contents

---
1. Executive Summary
The IINTS-AF SDK (Intelligent Insulin Titration System for Artificial Pancreas) is a **safety-first simulation and validation platform** for insulin dosing algorithms targeting closed-loop insulin delivery research.
Key Capabilities

Plug-and-play algorithm architecture - Implement one method, get full simulation **9-layer Independent Safety Supervisor** - Deterministic override guarantees **Realistic device models** - CGM sensor and insulin pump error simulation **Commercial pump emulators** - Medtronic 780G, Omnipod 5, Tandem Control-IQ **Clinical metrics** - TIR, GMI, CV, LBGI, HBGI per ATTD/ADA guidelines **Complete audit trail** - JSONL + CSV + JSON summary with integrity hashing **PDF clinical reports** - Visual reports with glucose traces and safety summaries **Benchmark mode** - Head-to-head algorithm comparison **Real-world data import** - Dexcom, Libre, Nightscout, AIDE/PEDAP, AZT1D, HUPA-UCM **Optional AI predictor** - Proactive glucose forecasting **Reproducible runs** - Seeded randomness and signable manifests

Intended Use

This SDK is intended for: - **Pre-clinical algorithm validation** - **Academic research** - **Educational purposes** - **Regulatory submission preparation**

NOT intended for: - Direct patient care - Clinical decision-making - Medical device deployment without regulatory review

---

2. Getting Started

2.1 Installation Guide

Option 1: Install from PyPI (Recommended)

```bash # Create project folder mkdir my-aps-research && cd my-aps-research

# Create virtual environment (recommended) python3 -m venv .venv source .venv/bin/activate  # macOS/Linux # .venv\Scripts\activate   # Windows

# Install SDK pip install iints-sdk-python35

# Verify installation iints --help ```

#### Option 2: Development Install

```bash git clone https://github.com/python35/IINTS-SDK.git cd IINTS-SDK pip install -e ".[dev]" ```

#### Option 3: With Research Extras (AI Predictor)

```bash pip install iints-sdk-python35[research] ```

### 2.2 Your First Simulation (60 seconds)

#### Using CLI (Quickstart)

```bash # Create quickstart project iints quickstart --project-name iints_quickstart cd iints_quickstart

# Run clinic-safe preset iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py ```

#### Using Python API
```python from iints import run_simulation from iints.core.algorithms.pid_controller import PIDController
# Run 12-hour simulation results = run_simulation(     algorithm=PIDController(),     duration_minutes=720,     seed=42 )
print(f"Results saved to: {results['results_csv']}") print(f"Report generated: {results['clinical_report']}") ```
**What this does:** - Creates a virtual patient with default parameters - Runs PID controller algorithm for 12 hours - Generates results CSV, clinical report PDF, and audit trail - Compares against baseline algorithms automatically
### 2.3 Understanding the Output Files
After running a simulation, you'll find these files in the `results/` folder:
| File | Description | |------|-------------| | `results.csv` | Every simulation step with glucose, insulin, IOB, COB, safety events | | `clinical_report.pdf` | Visual report with charts and clinical metrics | | `config.json` | Exact configuration used (for reproducibility) | | `run_metadata.json` | Run ID, seed, platform info, timestamps | | `run_manifest.json` | SHA-256 hashes of all files (integrity verification) | | `audit/audit_trail.csv` | Detailed per-step audit trail | | `audit/safety_summary.json` | Safety interventions summary | | `baseline/pid_results.csv` | PID controller baseline comparison | | `baseline/standard_pump.csv` | Standard pump baseline comparison |
**Example: Loading Results in Python**
```python import pandas as pd
# Load main results df = pd.read_csv("results/your_run/results.csv") print(df.head())
# Load safety summary import json with open("results/your_run/audit/safety_summary.json") as f: safety = json.load(f) print(f"Total interventions: {safety['intervention_count']}") ```
### 2.4 Creating Your First Custom Algorithm
#### Step 1: Generate Template
```bash iints new-algo --name MyAlgorithm --output-dir algorithms/ ```
#### Step 2: Implement Logic
```python # algorithms/my_algorithm.py from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput
class MyAlgorithm(InsulinAlgorithm):     def get_algorithm_metadata(self):         return {             "name": "My Custom Algorithm",             "version": "0.1.0",             "description": "My first insulin dosing algorithm"         }         def predict_insulin(self, data: AlgorithmInput) -> dict:         # Simple example: basal rate + correction         basal = 0.9  # U/hr         correction = 0.0                  # Add correction if glucose is high         if data.current_glucose > 180:             correction = (data.current_glucose - 180) / 50  # ISF of 50                  return {             "basal_insulin": basal,             "bolus_insulin": correction,             "reason": f"Basal {basal}U + correction {correction}U for glucose {data.current_glucose}mg/dL"         } ```
#### Step 3: Test Your Algorithm
```bash # Run with your custom algorithm iints run --algo algorithms/my_algorithm.py --duration 1440 ```
```

2.5 Adding Stress Tests
Add realistic challenges to test algorithm robustness:
```python from iints.core.simulator import Simulator, StressEvent
sim = Simulator(algorithm=MyAlgorithm(), patient_config="default")
# Add meal at 8:00 AM (480 minutes) sim.add_stress_event(StressEvent(     start_time=480,     event_type="meal",     value=60  # 60g carbs ))
# Add exercise at 6:00 PM (1080 minutes) sim.add_stress_event(StressEvent(     start_time=1080,     event_type="exercise",     value=30  # 30 minutes moderate exercise ))
# Add sensor noise sim.add_stress_event(StressEvent(     start_time=300,     event_type="sensor_noise",     value=15.0  # 15 mg/dL standard deviation ))
results = sim.run_batch(duration_minutes=1440) ```
**Available Stress Events:** - `meal`: Carbohydrate intake (value = grams) - `exercise`: Physical activity (value = minutes) - `sensor_noise`: CGM noise (value = std deviation) - `sensor_dropout`: CGM signal loss (value = probability) - `pump_failure`: Insulin delivery failure (value = probability)
- `hormonal_change`: Dawn phenomenon simulation
### 2.6 Benchmarking Against Baselines
Compare your algorithm against standard approaches:
```bash # CLI benchmark iints benchmark \   --algo-to-benchmark algorithms/my_algorithm.py \   --patient-configs-dir src/iints/data/virtual_patients \   --scenarios-dir scenarios \   --output-dir results/benchmark ```
```python # Python API benchmark from iints.analysis.benchmark import run_benchmark
results = run_benchmark(     algorithm_path="algorithms/my_algorithm.py",     patient_configs=["default", "adolescent", "insulin_resistant"],     scenarios=["baseline", "meal_challenge", "exercise_stress"],     duration_minutes=1440 ) ```
### 2.7 Using Preset Scenarios
Clinic-safe scenarios for reproducible testing:
```bash # List available presets iints presets list
# Run a specific preset iints presets run --name hypo_prone_night --algo algorithms/my_algorithm.py
```
Available Presets: - `baseline_t1d`: Standard Type 1 diabetes profile - `hypo_prone_night`: Nighttime hypoglycemia risk - `hyper_challenge`: Post-meal hyperglycemia - `pizza_paradox`: Delayed carb absorption - `midnight_crash`: Nocturnal hypoglycemia - `exercise_stress`: Physical activity impact
 - `sensor_noise`: CGM accuracy challenges
2.8 Importing Real CGM Data
```

Test against real patient data:

```bash # From CSV file iints import-data --input-csv my_cgm_export.csv \   --data-format dexcom \
 --output-dir results/imported ```

```python # Python API from iints.data.import_cgm_csv
result = import_cgm_csv(     "my_cgm_export.csv",     data_format="dexcom",  # or "libre", "generic"
     scenario_name="Patient A - Week 1" )
# Use in simulation sim = Simulator(     algorithm=MyAlgorithm(),     scenario=result.scenario ) ```

**Supported Formats:** - Dexcom CSV export - Libre CSV export - Generic CSV (auto-detects columns) - Nightscout JSON - Dataset registry packs (AIDE, PEDAP, AZT1D, HUPA-UCM)

### 2.9 Generating Custom Reports

```python from iints.analysis.reporting import ClinicalReportGenerator
generator = ClinicalReportGenerator() generator.generate_pdf(     results_df,  # Your simulation results DataFrame     safety_report,  # Safety summary dict     "results/custom_report.pdf",     title="My Algorithm Performance Report",     include_trend_analysis=True,     highlight_safety_events=True
 ) ```

**Report Contents:** - Glucose trace chart with target range - Insulin delivery chart - Clinical metrics table (TIR, GMI, CV, LBGI, HBGI) - Safety interventions summary - Top intervention reasons - Configuration overview

### 2.10 What Next?

**Recommended Next Steps:**

1.  Complete the Getting Started guide 2.  Run benchmark comparisons 3.  Test with stress scenarios 4.  Import real CGM data 5.  Explore AI predictor integration 6.  Generate clinical reports 7.  Review safety architecture 8.  Customize patient profiles 9.  Package algorithm for distribution 10.  Share results with community

**Need Help?** - Check Troubleshooting section (9.0) - Review FAQ (17.0) - Join community discussions - Submit issues on GitHub

---

## 3. Architecture Overview

### 3.1 System Components

```
IINTS-AF SDK Architecture
Algorithm Layer    Safety Layer      Output                                    Layer

- Custom Algorithm - Input Validator - Results - PID Controller - Safety Supervisor - Reports - ML Predictor - Safety Config - Audit - Pump Emulators - Metrics
Patient Simulation - Virtual Patient Model - CGM Sensor Model (noise, lag, dropout) - Insulin Pump Model (limits, quantization) - Pharmacokinetics (insulin absorption) - Pharmacodynamics (glucose response)
```

3.2 Data Flow

``` 1. Algorithm requests insulin dose 2. Input Validator checks glucose values 3. Safety Supervisor applies 9 safety checks 4. Approved dose sent to pump model 5. Pump model simulates delivery (with possible errors) 6. Patient model calculates glucose impact 7. CGM sensor model adds noise/lag 8. New glucose reading returned to algorithm 9. Audit trail logs all decisions 10. Repeat every time step (default: 5 minutes) ```

3.3 Safety Layer Integration

The Independent Safety Supervisor runs **deterministically** and can: - Override dangerous algorithm requests - Log all interventions with reasons - Enforce hard limits (hypoglycemia protection) - Apply dynamic limits (IOB clamping) - Validate all inputs/outputs

Key Principle: Safety layer is **always active** and cannot be disabled.

---

4. Safety Architecture (Critical Section)

4.1 Design Philosophy

Safety-First Principles:

1. **Deterministic Overrides**: Same input same safety decision 2. **Fail-Safe Defaults**: When in doubt, reduce insulin 3. **Audit Everything**: Every decision logged for accountability 4. **Transparent Logic**: Clear reasons for all interventions 5. **Configurable Thresholds**: Adapt to different patient profiles

Safety Guarantees: - No algorithm can deliver unsafe doses - Hypoglycemia protection is absolute (< 40 mg/dL emergency stop) - All interventions are logged and explainable - Configuration is validated before simulation starts

4.2 SafetyConfig Configuration

```python from iints.core.safety import SafetyConfig
# Default configuration (clinic-safe) config = SafetyConfig( # Hypoglycemia protection hypo_cutoff=70.0, # mg/dL - start reducing insulin severe_hypo_cutoff=54.0, # mg/dL - emergency stop critical_hypo_cutoff=40.0, # mg/dL - immediate termination # Hyperglycemia limits hyper_cutoff=300.0, # mg/dL - maximum allowed # Insulin limits max_basal_rate=2.0, # U/hr max_bolus=5.0, # U per bolus max_iob=10.0, # U total active insulin # Rate limits max_insulin_per_hour=15.0, # U/hr rolling window max_insulin_per_day=80.0, # U/day absolute limit # Trend protection contract_enabled=True, contract_glucose_threshold=90.0, # mg/dL contract_trend_threshold=-1.0, # mg/dL per 5 minutes ) ```

### 4.3 9 Safety Checks Explained

The IndependentSupervisor applies these checks **in order**:

``` 1. Predictive Hypo Guard [EMERGENCY] - If glucose < 70 AND falling fast (-3+ mg/dL per 5min) - Action: Suspend insulin for 30 minutes - Rationale: Prevent imminent severe hypo

2. Basal Rate Limit [WARNING]    - If basal > max_basal_rate    - Action: Cap at max_basal_rate    - Rationale: Prevent basal overdose
3. Hard Hypo Cutoff [EMERGENCY]    - If glucose < 54 mg/dL    - Action: Suspend all insulin    - Rationale: Severe hypoglycemia protection
4. Severe Hypo Emergency Stop [EMERGENCY]    - If glucose < 40 mg/dL    - Action: Terminate simulation    - Rationale: Critical hypoglycemia - stop everything
5. Glucose Level Clamp [CRITICAL/WARNING]    - If glucose > 300 mg/dL    - Action: Reduce insulin by 50%    - Rationale: Prevent over-correction
6. Rate-of-Change Trend Stop [CRITICAL]    - If glucose falling > 5 mg/dL per 5min    - Action: Suspend insulin    - Rationale: Rapid drop protection
7. Dynamic IOB Clamp [WARNING]    - If IOB > max_iob    - Action: Reduce dose to stay under max_iob    - Rationale: Prevent insulin stacking
8. Bolus Stacking Check [WARNING]    - If recent boluses > safety limit    - Action: Delay or reduce bolus    - Rationale: Prevent bolus overlap
9. 60-Minute Rolling Cap [WARNING]    - If insulin in last 60min > max_insulin_per_hour    - Action: Reduce dose to stay under limit    - Rationale: Hourly limit enforcement ```

### 4.4 Safety Levels

| Level | Severity | Action | |-------|----------|--------| | INFO | Informational | Log only, no intervention | | WARNING | Potential issue | Adjust dose within safe limits | | CRITICAL | Serious risk | Significant dose reduction | | EMERGENCY | Immediate danger | Suspend insulin completely |

### 4.5 Input Validation

The InputValidator checks:

```python # Glucose range validation if glucose < 20 or glucose > 600:     raise InvalidGlucoseError(f"Glucose {glucose} outside valid range")
# Insulin request validation if insulin < 0 or insulin > config.max_bolus:     raise InvalidInsulinError(f"Insulin {insulin} invalid")
# Timestep validation if timestep < 1 or timestep > 15:     raise InvalidTimestepError(f"Timestep {timestep} invalid") ```

### 4.6 Simulation Termination

Automatic termination occurs when:

1. **Critical hypoglycemia**: Glucose < 40 mg/dL for 30+ minutes 2. **Configuration error**: Invalid safety configuration 3. **Algorithm error**: Unhandled exception in algorithm 4. **Manual stop**: User interrupts simulation

**Termination Output:** - `SimulationLimitError` exception raised - Safety report marks `terminated_early: true` - Final glucose and intervention reason logged - Partial results still saved

---
```

5. Tutorials and Cookbook

5.1 24-Hour Simulation Walkthrough

Complete example from setup to analysis:

```python import iints import pandas as pd import matplotlib.pyplot as plt from iints.core.algorithm
s.pid_controller import PIDController from iints.core.simulator import Simulator, StressEvent
# 1. Setup simulation sim = Simulator(     algorithm=PIDController(),     patient_config="default",
    time_step=5,  # 5-minute steps     enable_profiling=True )
# 2. Add realistic stress events sim.add_stress_event(StressEvent(start_time=480, event_type="meal",
 value=60))  # 8:00 AM breakfast sim.add_stress_event(StressEvent(start_time=720, event_type="meal",
 value=45))  # 12:00 PM lunch sim.add_stress_event(StressEvent(start_time=1080, event_type="exercise
", value=45))  # 6:00 PM workout sim.add_stress_event(StressEvent(start_time=1320, event_type="meal"
, value=75))  # 8:00 PM dinner
# 3. Run 24-hour simulation results_df, safety_report = sim.run_batch(duration_minutes=1440)
# 4. Analyze results print(f"Time in Range (70-180 mg/dL): {iints.metrics.calculate_tir(results_df):
.1f}%") print(f"Glucose Management Indicator: {iints.metrics.calculate_gmi(results_df):.1f}") print(
f"Safety interventions: {safety_report['intervention_count']}")
# 5. Visualize plt.figure(figsize=(12, 6)) plt.plot(results_df['timestamp'], results_df['glucose_act
ual_mgdl']) plt.axhline(180, color='red', linestyle='--', label='Hyperglycemia') plt.axhline(70, col
or='green', linestyle='--', label='Target') plt.axhline(54, color='orange', linestyle='--', label='H
ypoglycemia') plt.title('24-Hour Glucose Profile') plt.xlabel('Time') plt.ylabel('Glucose (mg/dL)')
plt.legend() plt.grid(True) plt.show()
# 6. Generate report iints.generate_clinical_report(     results_df,     safety_report,     "results
/24hour_report.pdf" ) ```
```