IINTS-AF SDK

Intelligent Insulin Titration System

for Artificial Pancreas Research

Technical Reference Manual

Version 0.1.17 | Python SDK

Author: Rune Bobbaers

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.

Audiences: Clinical Researchers | Regulatory Reviewers | Open-Source Contributors

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. It provides a complete pipeline from algorithm development through simulation, safety supervision, clinical metrics analysis, and audit-ready reporting.

The SDK enables researchers to test AI or classical insulin controllers on virtual patients, enforce deterministic safety constraints via an independent supervisor layer, generate audit-ready clinical PDF reports, import and process real-world CGM data, benchmark custom algorithms against PID and standard pump baselines, and stress-test sensor noise, pump limits, and human-in-the-loop interventions.

Key Capabilities

- Plug-and-play algorithm architecture: implement one method, get full simulation
- 9-layer Independent Safety Supervisor with deterministic override guarantees
- Realistic CGM sensor and insulin pump error models (noise, bias, lag, dropout)
- Commercial pump emulators: Medtronic 780G, Omnipod 5, Tandem Control-IQ
- Clinical metrics: TIR, GMI, CV, LBGI, HBGI per ATTD/ADA guidelines
- Audit trail: JSONL + CSV + JSON summary with SHA-256 integrity hashing
- PDF clinical reports with glucose traces, insulin plots, and safety summaries
- Benchmark mode: head-to-head algorithm comparison (CLI: iints benchmark)
- Real-world CGM data import (Dexcom, Libre, Nightscout, AIDE/PEDAP, AZT1D, HUPA-UCM)
- Optional AI predictor for proactive glucose forecasting (forecast signal only)
- Reproducible runs with seeded randomness and GPG-signable manifests
- CLI (iints command) and Python API for flexible integration

Intended Use

This SDK is intended for pre-clinical algorithm validation, academic research, and educational purposes. It is NOT intended for direct patient care, clinical decision-making, or deployment in medical devices without independent regulatory review and clearance.

2. Getting Started

This chapter walks you through everything you need to go from zero to running your first simulation in under 10 minutes. Follow the steps in order.

2.1 Step 1: Install the SDK

Open a terminal and run:

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

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

# Install the SDK from PyPI
pip install iints-sdk-python35
```

Verify the installation by running the CLI:

```
iints --help
```

You should see a list of available commands. If you see a 'command not found' error, make sure your virtual environment is activated.

2.2 Step 2: Run Your First Simulation (60 seconds)

The fastest way to see the SDK in action is to use the Quickstart project and run a clinic-safe preset. This creates an example algorithm file and produces a full report:

```
# CLI: quickstart project + preset run
iints quickstart --project-name iints_quickstart
cd iints_quickstart
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py

# Or from Python:
from iints import run_simulation
from iints.core.algorithms.pid_controller import PIDController

results = run_simulation(algorithm=PIDController(), duration_minutes=720, seed=42)
print(f"Results CSV: {results['results_csv']}")
```

This runs a 12-hour (720-minute) simulation with a safe baseline preset. The preset run produces a PDF report and audit trail inside results/presets/.

2.3 Step 3: Understand the Output

After the run completes, you will find a timestamped folder in results/ containing everything the SDK produced. Here is what each file means:

- results.csv -- Every simulation step: glucose, insulin, IOB, COB, safety events. Open in Excel, pandas, or any CSV viewer.
- clinical_report.pdf -- Visual report with glucose trace chart, insulin delivery chart, clinical metrics table (TIR, GMI, CV), and safety summary.
- config.json -- Exact configuration used for this run. Useful for reproducing results.
- run_metadata.json -- Run ID, random seed, platform info. Proves when and where the run happened.
- run_manifest.json -- SHA-256 hashes of every file. Proves nothing was tampered with.
- audit/ folder -- Detailed per-step audit trail (JSONL + CSV + summary). Every decision is logged.
- baseline/ folder -- How PID and Standard Pump performed on the same scenario for comparison.

2.4 Step 4: Create Your First Custom Algorithm

The whole point of the SDK is testing YOUR algorithm. Here is the minimal template to get started. Create a file called my_algorithm.py:

```
from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput

class MyFirstAlgorithm(InsulinAlgorithm):
    """My first insulin algorithm."""

    def predict_insulin(self, data: AlgorithmInput):
        self.why_log = []   # Clear reasoning log each step
        dose = 0.0

        # Simple rule: correct if glucose > 130 mg/dL
        if data.current_glucose > 130:
            dose = (data.current_glucose - 110) / self.isf
            self._log_reason(
                f"Correcting: glucose {data.current_glucose:.0f} > 130",
                "glucose_level", dose
            )

        # Cover meals
        if data.carb_intake > 0:
            meal_dose = data.carb_intake / self.icr
            dose += meal_dose
            self._log_reason(
                f"Meal cover: {data.carb_intake:.0f}g -> {meal_dose:.2f}U",
                "calculation", meal_dose
            )

        # Subtract IOB to prevent stacking
        dose = max(0.0, dose - data.insulin_on_board)

        # Suspend if glucose low or dropping fast
        if data.current_glucose < 80 or (
            data.glucose_trend_mgdl_min is not None
            and data.glucose_trend_mgdl_min < -2.0
        ):
            dose = 0.0
            self._log_reason("SUSPENDED: low/dropping glucose", "safety")

        return {"total_insulin_delivered": dose}
```

Now run it:

```
from iints import run_simulation
from my_algorithm import MyFirstAlgorithm

results = run_simulation(
    algorithm=MyFirstAlgorithm,
    duration_minutes=720,
    seed=42,
)
print(f"TIR: check {results['report_pdf']}")
```

2.5 Step 5: Add Stress Tests (Meals, Exercise, Sensor Faults)

Real-world scenarios are messy. Test how your algorithm handles them:

```
from iints import Simulator, PatientModel, StressEvent
from my_algorithm import MyFirstAlgorithm

sim = Simulator(
    patient_model=PatientModel(initial_glucose=120),
    algorithm=MyFirstAlgorithm(),
    time_step=5, seed=42,
)

# Breakfast at t=60 min (algo told 50g, patient eats 65g)
sim.add_stress_event(StressEvent(
    start_time=60, event_type="meal",
    value=65, reported_value=50,
))

# Unannounced snack at t=180 (algo has NO idea)
sim.add_stress_event(StressEvent(
    start_time=180, event_type="missed_meal", value=25
))

# Afternoon jog at t=300
sim.add_stress_event(StressEvent(
    start_time=300, event_type="exercise",
    value=0.5, duration=30,
))

df, safety = sim.run_batch(480)
print(f"Safety violations: {safety['total_violations']}")
```

2.6 Step 6: Benchmark Against Baselines

Once your algorithm works, you want to know: is it better than PID? Use the CLI benchmark mode for most workflows, or the BattleRunner API when you need programmatic access.

```
# CLI benchmark (recommended)
iints benchmark --algo-to-benchmark algorithms/my_algorithm.py \
  --scenarios-dir scenarios --output-dir results/benchmarks
```

Python API (BattleRunner) for in-notebook ranking:

```
from iints.core.algorithms.battle_runner import BattleRunner
from iints.core.algorithms.pid_controller import PIDController
from my_algorithm import MyFirstAlgorithm
import pandas as pd

# Create minimal patient data
patient_df = pd.DataFrame({
    "time": range(0, 720, 5),
    "glucose": [120] * 144,
    "carbs": [0] * 144,
})
# Add a 60g meal at t=60
patient_df.loc[patient_df["time"] == 60, "carbs"] = 60

runner = BattleRunner(
    algorithms={
        "PID": PIDController(),
        "MyAlgo": MyFirstAlgorithm(),
    },
    patient_data=patient_df,
)
report, sim_data = runner.run_battle()
print(f"Winner: {report['winner']}")
for r in report["rankings"]:
    print(f"  {r['participant']}: TIR={r['tir']:.1f}%  CV={r['cv']:.1f}%")
```

2.7 Step 7: Use the Preset Scenarios

The SDK ships with 6 clinic-safe preset scenarios that cover common real-world situations. These are ready to use out of the box:

```
# List available presets
iints presets list

# Run a preset from the CLI
iints presets run baseline_t1d
iints presets run hypo_prone_night
iints presets run pizza_paradox
```

Preset Name	Scenario	What It Tests
baseline_t1d	24h, 3 meals + exercise	Standard day with realistic schedule
stress_test_meal	Large meal challenge	Tests post-meal glucose recovery
hypo_prone_night	Overnight hypo scenario	Tests low-glucose prevention
hyper_challenge	110g carb meal	Extreme carb load stress test
pizza_paradox	180-min delayed absorption	Fat+protein delayed glucose spike
midnight_crash	Evening exercise + overnight	Exercise-induced delayed hypo

2.8 Step 8: Import Your Own CGM Data

Test your algorithm against real patient data from your Dexcom, Libre, or Nightscout setup:

```
# From a CSV file (Dexcom, Libre, or generic)
iints import-data --input-csv my_cgm_export.csv --data-format dexcom \
  --output-dir results/imported

# From Python
from iints import import_cgm_csv
result = import_cgm_csv("my_cgm_export.csv", data_format="dexcom")
df = result.dataframe   # Standardized: timestamp, glucose, carbs, insulin

# Use bundled demo data (no external files needed)
from iints import load_demo_dataframe
demo_df = load_demo_dataframe()
```

2.9 Step 9: Generate and Customize Reports

The SDK generates PDF clinical reports automatically during run_simulation(). You can also generate reports from existing CSV data:

```
# Generate report from an existing results CSV
iints report --results-csv results/results.csv --output-path my_report.pdf

# From Python, with a custom title
from iints import generate_report
import pandas as pd

df = pd.read_csv("results/results.csv")
generate_report(df, output_path="my_report.pdf")
```

2.10 Step 10: What Next?

Now that you have the basics, here is your roadmap for going deeper:

- Chapter 4 (Safety Architecture) -- Understand the 9 safety checks protecting your algorithm
- Chapter 5 (Tutorials & Cookbook) -- Step-by-step walkthroughs of advanced scenarios
- Chapter 7 (API Reference) -- Complete reference for every class and method
- Chapter 9 (Pump Emulators) -- Compare your algorithm against Medtronic 780G, Omnipod 5, Control-IQ
- Chapter 12 (Limitations) -- Critical reading before publishing any results
- Chapter 15 (Troubleshooting) -- Solutions for common issues

3. Architecture Overview

The SDK follows a layered architecture where each layer has a single responsibility and communicates through well-defined interfaces. The layers are designed so that the safety layer operates independently from the algorithm layer, providing a defense-in-depth model analogous to avionics safety architectures.

3.1 Architectural Layers

Layer 1: Data Ingestion (iints.data)

Universal schema bridge for CGM data from CSV, Nightscout, Tidepool, and bundled datasets. Includes column auto-detection, format normalization, and data quality checks.

Layer 2: Patient Model (iints.core.patient)

Simplified biophysical glucose dynamics model with configurable ISF, ICR, DIA, basal rate, dawn phenomenon, exercise effects, and meal mismatch simulation.

Layer 3: Algorithm Contract (iints.api)

Abstract base class (InsulinAlgorithm) defining the plug-and-play interface. Algorithms implement predict_insulin() and receive structured AlgorithmInput.

Layer 4: Safety Layer (iints.core.safety + iints.core.supervisor)

Independent deterministic supervisor with 9 sequential safety checks, input validation, and centralized SafetyConfig. Operates independently of the algorithm.

Layer 5: Simulation Engine (iints.core.simulator)

5-minute step orchestrator tying patient, algorithm, supervisor, sensor model, pump model, stress events, audit logging, and human-in-the-loop callbacks.

Layer 6: Device Models (iints.core.devices)

CGM SensorModel (noise, bias, lag, dropout) and PumpModel (quantization, max delivery, occlusion, delivery noise) for realistic hardware simulation.

Layer 7: Emulation (iints.emulation)

Behavioral emulators for Medtronic 780G, Omnipod 5, and Tandem Control-IQ based on published FDA 510(k) documents and clinical studies.

Layer 8: Analysis (iints.analysis)

Clinical metrics (TIR, GMI, CV, LBGI, HBGI), PDF reporting, baseline comparison, explainability analysis, and hardware benchmarking.

Layer 9: High-Level API (iints.highlevel)

One-line run_simulation() and run_full() functions that orchestrate the entire pipeline with reproducible run bundles and SHA-256 manifest hashing.

Layer 10: CLI (iints.cli)

Typer-based command-line interface exposing the full workflow: init, run, import, validate, benchmark, report, algorithm discovery, and more.

4. Data Flow

4.1 Simulation Loop (Per Time Step)

The core simulation operates on a fixed time step (default 5 minutes). Each step executes the following pipeline:

- 1. Patient model provides true glucose value
- 2. InputValidator checks glucose for physiological plausibility
- 3. SensorModel applies noise/bias/lag/dropout to produce CGM reading
- 4. Stress events are processed (meals, missed meals, sensor errors, exercise, ratio changes)
- 5. Delayed meal absorption queue is updated
- 6. Glucose trend (mg/dL/min) is calculated from sequential sensor readings
- 7. Heuristic glucose prediction (30-min horizon) using trend + IOB + COB
- 8. Optional AI predictor (LSTM) refines the 30-min glucose forecast
- 9. AlgorithmInput dataclass is assembled with all context
- 10. Algorithm.predict_insulin() is called to get proposed dose
- 11. InputValidator checks proposed insulin for negative values
- 12. IndependentSupervisor.evaluate_safety() applies 9 sequential safety checks
- 13. PumpModel applies delivery constraints (quantization, max dose, occlusion)
- 14. Optional on_step() human-in-the-loop callback for manual override
- 15. PatientModel.update() applies delivered insulin + actual carbs
- 16. Full record is logged with all intermediate values
- 17. Critical failure check: if glucose < 40 mg/dL for 30+ minutes, simulation stops

4.2 Output Artifacts

Each simulation run produces a structured output directory containing:

- results.csv - Complete simulation DataFrame with all columns
- config.json - Full run configuration (algorithm, patient, scenario, safety)
- run_metadata.json - Run ID, seed, timestamp, platform info, SDK version
- run_manifest.json - SHA-256 hashes of all output files for integrity verification
- clinical_report.pdf - PDF with glucose trace, insulin plot, metrics, safety summary
- audit/audit_trail.jsonl - Per-step audit log in JSON Lines format
- audit/audit_trail.csv - Same audit data as CSV
- audit/audit_summary.json - Override counts and top intervention reasons
- baseline/ - PID and Standard Pump comparison results (JSON + CSV)
- profiling.json - Algorithm and supervisor latency statistics (if profiling enabled)

4.3 Data Import Flow

CGM data can be imported from multiple sources through a unified pipeline: CSV files (generic, Dexcom, Libre formats), Nightscout REST API, official dataset registry packs (AIDE, PEDAP, AZT1D, HUPA-UCM), and bundled demo data. The UniversalParser auto-detects column names with fuzzy matching, normalizes timestamps, validates glucose ranges, and outputs a standardized DataFrame with columns: timestamp, glucose, carbs, insulin, source.

5. Safety Architecture

SAFETY-CRITICAL DESIGN

The safety layer is the most critical component of this SDK. It operates independently from any algorithm and applies deterministic, hard-coded safety checks that cannot be bypassed by algorithm output. The supervisor ALWAYS has final authority over insulin delivery.

5.1 Design Philosophy

The IINTS-AF safety architecture follows the principle of defense-in-depth with separation of concerns. The algorithm proposes a dose; the supervisor decides whether to allow it. This mirrors the independent safety monitor pattern used in avionics (DO-178C) and medical device software (IEC 62304). The supervisor is stateless with respect to algorithm internals - it only observes glucose, IOB, trends, and predictions.

5.2 SafetyConfig (Central Configuration)

All safety thresholds are centralized in a single SafetyConfig dataclass:

Parameter	Default	Unit	Description
min_glucose	20.0	mg/dL	Minimum valid glucose (input validation)
max_glucose	600.0	mg/dL	Maximum valid glucose (input validation)
max_glucose_delta_per_5_min	35.0	mg/dL	Max plausible rate change
hypoglycemia_threshold	70.0	mg/dL	Hypo alert threshold
severe_hypoglycemia_threshold	54.0	mg/dL	Severe hypo (EMERGENCY)
hyperglycemia_threshold	250.0	mg/dL	Hyper alert threshold
max_insulin_per_bolus	5.0	Units	Hard bolus cap
max_insulin_per_hour	3.0	Units	60-minute rolling window cap
max_iob	4.0	Units	Maximum insulin on board
trend_stop	-2.0	mg/dL/min	Fast-drop insulin suspension
hypo_cutoff	70.0	mg/dL	Hard hypoglycemia insulin stop
max_basal_multiplier	3.0	x	Max basal relative to patient rate
predicted_hypo_threshold	60.0	mg/dL	Predictive hypo threshold
predicted_hypo_horizon	30	min	Prediction lookahead
critical_glucose_threshold	40.0	mg/dL	Simulation termination level
critical_glucose_duration	30	min	Duration before sim terminates

5.3 IndependentSupervisor: 9 Safety Checks

The supervisor evaluates proposed insulin in strict sequential order. Each check can reduce or zero the proposed

dose. Checks are cumulative - a dose reduced by check #3 may be further reduced by check #5.

1. Predictive Hypo Guard [EMERGENCY]

If predicted glucose in 30 min <= 60 mg/dL, insulin is zeroed immediately. Uses heuristic or AI prediction.

2. Basal Rate Limit [WARNING]

If proposed basal exceeds patient's basal rate x max_basal_multiplier (3x), the excess is subtracted from the total dose.

3. Hard Hypo Cutoff [EMERGENCY]

If current glucose <= 70 mg/dL, all insulin delivery is stopped.

4. Severe Hypo Emergency Stop [EMERGENCY]

If current glucose <= 54 mg/dL, insulin is zeroed and emergency mode activates.

5. Glucose Level Clamp [CRITICAL/WARNING]

Glucose 54-70: insulin capped at 0.1 U. Glucose >= 250: warning logged.

6. Rate-of-Change Trend Stop [CRITICAL]

If glucose dropping >= 2 mg/dL/min, all insulin is suspended.

7. Dynamic IOB Clamp [WARNING]

If insulin on board >= max_iob (4.0 U), no additional insulin is allowed.

8. Bolus Stacking Check [WARNING]

When IOB exceeds max bolus threshold and glucose < 150 mg/dL, dose is proportionally reduced to prevent stacking.

9. 60-Minute Rolling Cap [WARNING]

Total insulin in the last 60 minutes is tracked. If adding proposed dose would exceed max_insulin_per_hour (3.0 U), it is capped to the remaining allowance.

5.4 Safety Levels

Each safety evaluation returns one of four levels: SAFE (no intervention), WARNING (dose modified, non-critical), CRITICAL (significant dose reduction), EMERGENCY (all insulin stopped, emergency mode activated). Emergency mode persists until glucose recovers above hypoglycemia_threshold + 20 mg/dL.

5.5 Input Validation

Before reaching the algorithm, all glucose values pass through InputValidator which rejects physiologically impossible readings (< 20 or > 600 mg/dL) and validates rate of change (max 35 mg/dL per 5 minutes). Negative insulin proposals from algorithms are clamped to zero.

5.6 Simulation Termination

If true glucose remains below critical_glucose_threshold (40 mg/dL) for critical_glucose_duration_minutes (30 min), the simulation raises SimulationLimitError. This prevents algorithms from silently producing dangerous outcomes in long-running simulations.

6. Tutorials and Cookbook

This chapter provides step-by-step tutorials for common workflows. Each recipe is self-contained and can be copied directly into your project.

6.1 Tutorial: Your First 24-Hour Simulation

Goal: Run a full-day simulation with breakfast, lunch, dinner, and a snack, then analyze the clinical metrics to see how well the algorithm performed.

```
from iints import (
    Simulator, PatientModel, StressEvent,
    PatientProfile, SafetyConfig, SensorModel
)
from iints.core.algorithms.pid_controller import PIDController
from iints.analysis.clinical_metrics import ClinicalMetricsCalculator

# 1. Configure the virtual patient
patient = PatientModel(
    initial_glucose=110,
    basal_insulin_rate=0.8,    # U/hr
    insulin_sensitivity=50,     # mg/dL per U
    carb_factor=10,            # grams per U
    dawn_phenomenon_strength=8, # mg/dL per hour
    dawn_start_hour=4.0,
    dawn_end_hour=7.0,
)

# 2. Set up the simulation with a realistic sensor
sim = Simulator(
    patient_model=patient,
    algorithm=PIDController(),
    time_step=5,
    seed=42,
    sensor_model=SensorModel(noise_std=3.0, lag_minutes=5),
)

# 3. Add realistic meal schedule
sim.add_stress_event(StressEvent(start_time=60,  event_type="meal", value=45))
sim.add_stress_event(StressEvent(start_time=300, event_type="meal", value=60))
sim.add_stress_event(StressEvent(start_time=480, event_type="meal", value=30))
sim.add_stress_event(StressEvent(start_time=720, event_type="meal", value=70))

# 4. Run
df, safety = sim.run_batch(1440)  # 24 hours

# 5. Analyze with clinical metrics
calc = ClinicalMetricsCalculator()
metrics = calc.calculate(glucose=df["glucose_actual_mgdl"], duration_hours=24)
print(metrics.get_summary())
print(f"Rating: {metrics.get_rating()}")
```

6.2 Tutorial: Building an ML-Hybrid Algorithm

Goal: Wrap an external ML model (scikit-learn, PyTorch, etc.) into the IINTS algorithm interface with proper fallback to rules when the model's confidence is low.

```
from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput

class MLHybridAlgorithm(InsulinAlgorithm):
    def __init__(self, model, confidence_threshold=0.7):
        super().__init__()
        self.model = model        # Your trained model
        self.threshold = confidence_threshold
        self.fallback_count = 0

    def predict_insulin(self, data: AlgorithmInput):
        self.why_log = []

        # Prepare features for your model
        features = [
            data.current_glucose,
            data.insulin_on_board,
            data.carb_intake,
            data.glucose_trend_mgdl_min or 0.0,
            data.predicted_glucose_30min or data.current_glucose,
        ]

        # Get prediction + confidence from your model
        prediction = self.model.predict([features])[0]
        confidence = self.model.predict_confidence([features])[0]

        if confidence >= self.threshold:
            dose = max(0.0, prediction)
            self._log_reason(
                f"ML prediction: {dose:.2f}U (conf: {confidence:.0%})",
                "calculation", dose
            )
        else:
            # Fallback to simple correction logic
            dose = 0.0
            if data.current_glucose > 130:
                dose = (data.current_glucose - 110) / self.isf
            self.fallback_count += 1
            self._log_reason(
                f"Fallback to rules (conf: {confidence:.0%})",
                "safety", dose,
                clinical_impact="Low model confidence"
            )

        return {
            "total_insulin_delivered": dose,
            "uncertainty": 1.0 - confidence,
            "fallback_triggered": confidence < self.threshold,
        }
```

6.3 Tutorial: Running Batch Experiments

Goal: Run your algorithm against multiple patients and multiple scenarios systematically, collecting results into a single comparison table.

```
from iints import run_simulation, PatientProfile, SafetyConfig
from my_algorithm import MyAlgorithm
import pandas as pd

# Define patient variants
patients = {
    "insulin_sensitive": PatientProfile(isf=80, icr=15, basal_rate=0.5),
    "average":           PatientProfile(isf=50, icr=10, basal_rate=0.8),
    "insulin_resistant":  PatientProfile(isf=25, icr=6,  basal_rate=1.5),
}

# Define scenario files
scenarios = [
    "scenarios/baseline.json",
    "scenarios/stress_test_meal.json",
    "scenarios/hypo_prone.json",
]

# Run all combinations
all_results = []
for patient_name, profile in patients.items():
    for scenario in scenarios:
        res = run_simulation(
            algorithm=MyAlgorithm(),
            patient_config=profile,
            scenario=scenario,
            duration_minutes=1440,
            seed=42,
        )
        all_results.append({
            "patient": patient_name,
            "scenario": scenario,
            "tir": res["safety_report"].get("tir_70_180", 0),
            "run_id": res["run_id"],
        })

# Summarize
summary = pd.DataFrame(all_results)
print(summary.to_string(index=False))
```

6.4 Tutorial: Analyzing the Audit Trail

Goal: After a run, dig into the audit trail to understand exactly when and why the safety supervisor intervened.

```
import json
import pandas as pd

# Load the audit trail
audit_df = pd.read_csv("results/RUN_ID/audit/audit_trail.csv")

# Find all safety interventions
interventions = audit_df[audit_df["safety_triggered"] == True]
print(f"Total interventions: {len(interventions)}")
print(f"Intervention rate: {len(interventions)/len(audit_df)*100:.1f}%")

# Break down by reason
reasons = interventions["safety_reason"].value_counts()
print("\nIntervention reasons:")
print(reasons.to_string())

# Find the most dangerous moments
critical = audit_df[audit_df["glucose_actual_mgdl"] < 70]
print(f"\nHypoglycemia episodes: {len(critical)} steps")
print(f"Lowest glucose: {audit_df['glucose_actual_mgdl'].min():.0f} mg/dL")

# Load the audit summary
with open("results/RUN_ID/audit/audit_summary.json") as f:
    summary = json.load(f)
print(f"\nTop override reasons: {summary['top_reasons']}")
```

6.5 Tutorial: Custom Safety Thresholds

Goal: Adjust the safety supervisor for a specific patient population (e.g., more conservative for pediatric patients, or relaxed for research testing).

```
from iints import run_simulation, SafetyConfig
from my_algorithm import MyAlgorithm

# Conservative config for a pediatric scenario
pediatric_safety = SafetyConfig(
    hypoglycemia_threshold=75.0,     # Higher alert (default: 70)
    severe_hypoglycemia_threshold=60.0, # (default: 54)
    max_iob=2.5,                     # Lower IOB cap (default: 4.0)
    max_insulin_per_bolus=3.0,       # Lower bolus cap (default: 5.0)
    max_insulin_per_hour=2.0,        # (default: 3.0)
    trend_stop=-1.5,                 # More sensitive (default: -2.0)
    predicted_hypo_threshold=65.0,   # (default: 60.0)
)

results = run_simulation(
    algorithm=MyAlgorithm(),
    safety_config=pediatric_safety,
    duration_minutes=1440,
)
```

6.6 Tutorial: Using the Pump Emulators as Benchmarks

Goal: Compare your algorithm's performance against the behavior of commercial pump systems.

```
from iints.emulation.medtronic_780g import Medtronic780GEmulator
from iints.emulation.tandem_controliq import TandemControlIQEmulator
from iints.emulation.omnipod_5 import Omnipod5Emulator

# Create emulator instances
emulators = {
    "Medtronic 780G": Medtronic780GEmulator(),
    "Tandem CIQ":     TandemControlIQEmulator(),
    "Omnipod 5":      Omnipod5Emulator(),
}

# Query each emulator at a given glucose state
for name, emulator in emulators.items():
    decision = emulator.make_decision(
        current_glucose=185.0,
        glucose_trend=-0.5,    # slowly dropping
        iob=1.2,
        cob=20.0,
    )
    print(f"{name}: {decision.action} "
          f"insulin={decision.insulin_dose:.2f}U "
          f"safety={decision.safety_level}")
```

6.7 Tutorial: Live Streaming Simulation

Goal: Process simulation results step-by-step as they happen, for real-time dashboards or streaming pipelines.

```
from iints import Simulator, PatientModel, StressEvent
from iints.core.algorithms.pid_controller import PIDController

sim = Simulator(
    patient_model=PatientModel(initial_glucose=120),
    algorithm=PIDController(),
    time_step=5,
)

sim.add_stress_event(StressEvent(start_time=30, event_type="meal", value=50))

# run_live() is a generator - yields one record per step
for step in sim.run_live(duration_minutes=240):
    glucose = step["glucose_actual_mgdl"]
    insulin = step["delivered_insulin"]
    time_min = step["time_minutes"]

    # Do anything: update a plot, send to websocket, log to DB
    if glucose < 70:
        print(f"  !! HYPO at t={time_min}: {glucose:.0f} mg/dL")
    elif glucose > 200:
        print(f"  !! HIGH at t={time_min}: {glucose:.0f} mg/dL")
```

6.8 Recipe: Reproducible Runs for Publications

When publishing results, you need to prove reproducibility. Use run_full() with a fixed seed and the SDK handles the rest:

```
from iints import run_full

# run_full() enables ALL outputs: audit, baseline, report, manifest
results = run_full(
    algorithm=MyAlgorithm(),
    duration_minutes=1440,
    seed=12345,           # Fixed seed = deterministic
)

# The output directory now contains:
# - run_manifest.json  (SHA-256 hashes of every file)
# - run_metadata.json  (seed, SDK version, platform)
# - config.json        (exact parameters used)
# - If GPG key available: run_manifest.json.sig

# To verify integrity later:
import json, hashlib
with open("results/RUN_ID/run_manifest.json") as f:
    manifest = json.load(f)
for filename, expected_hash in manifest["files"].items():
    with open(f"results/RUN_ID/{filename}", "rb") as fh:
        actual = hashlib.sha256(fh.read()).hexdigest()
    assert actual == expected_hash, f"TAMPERED: {filename}"
print("All files verified.")
```

7. Presets and Scenario Design

Scenarios define the stress events and conditions for a simulation run. The SDK supports hand-crafted JSON scenarios, built-in clinic-safe presets, random scenario generation, and patient YAML profiles.

7.1 Scenario JSON Format

A scenario file is a JSON object with the following structure:

```
{
  "schema_version": 2,
  "name": "Breakfast Stress Test",
  "description": "Tests post-breakfast glucose control",
  "duration_minutes": 480,
  "stress_events": [
    {
      "start_time": 30,
      "event_type": "meal",
      "value": 60,
      "reported_value": 50,
      "absorption_delay_minutes": 10
    },
    {
      "start_time": 240,
      "event_type": "exercise",
      "value": 0.6,
      "duration": 30
    }
  ],
  "patient_config": {
    "initial_glucose": 110,
    "basal_insulin_rate": 0.8,
    "insulin_sensitivity": 50,
    "carb_factor": 10
  }
}
```

7.2 Patient YAML Profiles

Patient configurations can be stored as YAML files in data/virtual_patients/. These define the physiological parameters of a virtual patient:

```
# data/virtual_patients/my_patient.yaml
name: "Hypo-Prone Adult"
initial_glucose: 95
basal_insulin_rate: 1.0
insulin_sensitivity: 65    # Very sensitive
carb_factor: 12
insulin_action_duration: 330
dawn_phenomenon_strength: 5
dawn_start_hour: 3.5
dawn_end_hour: 7.0
```

Use it from the CLI:

```
iints run-full --algo algorithms/my_algorithm.py \
  --patient-config-name my_patient --duration 1440
```

7.3 Random Scenario Generation

Generate random stress-test scenarios for robustness testing:

```
from iints.scenarios import ScenarioGeneratorConfig, generate_random_scenario

config = ScenarioGeneratorConfig(
    duration_minutes=1440,
    min_meals=2,
    max_meals=5,
    min_carbs=20,
    max_carbs=80,
    exercise_probability=0.3,
)

# Generate 10 random scenarios
for i in range(10):
    scenario = generate_random_scenario(config, seed=i)
    # Save or use directly
    with open(f"scenarios/random_{i}.json", "w") as f:
        import json
        json.dump(scenario, f, indent=2)
```

Or from the CLI:

```
iints scenarios generate --count 10 --output-dir scenarios/
```

7.4 Validation

Always validate your scenario and patient files before running simulations. The SDK uses Pydantic v2 for strict schema validation:

```
# CLI validation
iints validate --scenario my_scenario.json
iints validate --patient my_patient.yaml

# Python validation
from iints.validation import load_scenario, load_patient_config

scenario = load_scenario("my_scenario.json")  # Raises on invalid
patient = load_patient_config("my_patient.yaml")  # Raises on invalid
```

8. API Reference

8.1 InsulinAlgorithm (Abstract Base Class)

Location: iints.api.base_algorithm.InsulinAlgorithm

The central interface for all insulin algorithms. Subclass this to create a custom algorithm. Only predict_insulin() must be implemented.

Abstract Method: predict_insulin(data: AlgorithmInput) -> Dict[str, Any]

Called at each simulation time step. Receives an AlgorithmInput dataclass and must return a dictionary containing at minimum 'total_insulin_delivered' (float). Optional keys: 'basal_insulin', 'bolus_insulin', 'correction_bolus', 'meal_bolus', 'uncertainty', 'fallback_triggered'.

Provided Methods

- reset(): Resets internal state and why_log for a new simulation run.
- get_state() -> Dict: Returns internal algorithm state for serialization.
- set_state(state: Dict): Restores internal state from a previous save.
- set_isf(isf: float): Sets Insulin Sensitivity Factor. Must be positive.
- set_icr(icr: float): Sets Insulin-to-Carb Ratio. Must be positive.
- get_algorithm_metadata(): Returns AlgorithmMetadata. Override for custom info.
- calculate_uncertainty(data): Returns uncertainty score 0.0-1.0. Override for custom.
- calculate_confidence_interval(): Returns (lower, upper) CI. Override for custom.
- explain_prediction(data, prediction): Returns human-readable explanation string.
- _log_reason(reason, category, ...): Adds a WhyLogEntry to the decision log.
- get_why_log() -> List[WhyLogEntry]: Returns the reasoning log for the last step.
- get_why_log_text() -> str: Returns formatted text version of the why log.

8.2 AlgorithmInput (Dataclass)

Location: iints.api.base_algorithm.AlgorithmInput

All inputs available to an algorithm at each time step:

Field	Type	Description
current_glucose	float	Current CGM reading (mg/dL), after sensor model
time_step	float	Simulation step duration in minutes
insulin_on_board	float	Active insulin remaining (Units). Default: 0.0
carb_intake	float	Announced carbs this step (grams). Default: 0.0
carbs_on_board	float	Carbs still being absorbed (grams). Default: 0.0

isf	Optional[float]	Insulin Sensitivity Factor (mg/dL per Unit)
icr	Optional[float]	Insulin-to-Carb Ratio (grams per Unit)
dia_minutes	Optional[float]	Duration of Insulin Action (minutes)
basal_rate_u_per_hr	Optional[float]	Current basal rate (Units/hour)
glucose_trend_mgdl_min	Optional[float]	Glucose rate of change (mg/dL/min)
predicted_glucose_30min	Optional[float]	30-minute glucose prediction
patient_state	Dict	Patient model state dictionary
current_time	float	Current simulation time (minutes). Default: 0.0

8.3 AlgorithmResult (Dataclass)

Location: iints.api.base_algorithm.AlgorithmResult

Structured result with uncertainty quantification:

Field	Type	Description
total_insulin_delivered	float	Total insulin dose (required)
bolus_insulin	float	Bolus component. Default: 0.0
basal_insulin	float	Basal component. Default: 0.0
correction_bolus	float	Correction bolus component. Default: 0.0
meal_bolus	float	Meal bolus component. Default: 0.0
uncertainty	float	0.0 = certain, 1.0 = very uncertain
confidence_interval	tuple	(lower_bound, upper_bound)
primary_reason	str	Main clinical reasoning
secondary_reasons	List[str]	Additional reasoning factors
safety_constraints	List[str]	Safety constraints applied
algorithm_name	str	Name of the algorithm
timestamp	datetime	Prediction timestamp

8.4 WhyLogEntry (Dataclass)

Location: iints.api.base_algorithm.WhyLogEntry

A single entry in the decision reasoning log. Categories: 'glucose_level', 'velocity', 'insulin_on_board', 'safety', 'context', 'calculation', 'control_parameter'.

Field	Type	Description
reason	str	Human-readable explanation of the decision factor
category	str	Classification of the reasoning type
value	Any	Associated numeric or contextual value. Default: None

clinical_impact	str	Expected clinical impact. Default: empty

8.5 Simulator

Location: iints.core.simulator.Simulator

The main simulation engine.

Constructor Parameters

Parameter	Type	Description
patient_model	PatientModel	The virtual patient instance
algorithm	InsulinAlgorithm	The algorithm under test
time_step	int	Step duration in minutes (default: 5)
seed	Optional[int]	Random seed for reproducibility
enable_profiling	bool	Enable latency profiling (default: False)
sensor_model	Optional[SensorModel]	Custom CGM sensor model
pump_model	Optional[PumpModel]	Custom pump model
on_step	Optional[Callable]	Human-in-the-loop callback
safety_config	Optional[SafetyConfig]	Custom safety thresholds
predictor	Optional[object]	AI glucose predictor (LSTM)

Key Methods

- run_batch(duration_minutes): Runs full simulation, returns (DataFrame, safety_report)
- run_live(duration_minutes): Generator yielding per-step records for streaming
- add_stress_event(event): Adds a StressEvent to the simulation timeline
- export_audit_trail(df, output_dir): Exports JSONL + CSV + JSON audit files
- save_state() -> Dict: Serializes full simulator state for time-travel debugging
- load_state(state: Dict): Restores simulator from a previous save_state()

8.6 PatientProfile (Dataclass)

Location: iints.core.patient.profile.PatientProfile

User-friendly patient configuration:

Field	Default	Description
isf	50.0	Insulin Sensitivity Factor (mg/dL per Unit)
icr	10.0	Insulin-to-Carb Ratio (grams per Unit)
basal_rate	0.8	Basal insulin rate (Units/hour)
initial_glucose	120.0	Starting glucose (mg/dL)
dawn_phenomenon_strength	0.0	Dawn rise (mg/dL per hour)

dawn_start_hour	4.0	Dawn phenomenon start (24h)
dawn_end_hour	8.0	Dawn phenomenon end (24h)
glucose_decay_rate	0.05	Metabolic glucose decay rate
glucose_absorption_rate	0.03	Carb absorption rate
insulin_action_duration	300.0	DIA in minutes (5 hours)
insulin_peak_time	75.0	Time to peak insulin action (min)
meal_mismatch_epsilon	1.0	Carb estimation error multiplier

8.7 SensorModel

Location: iints.core.devices.models.SensorModel

CGM error simulation:

Parameter	Default	Description
noise_std	0.0	Gaussian noise standard deviation (mg/dL)
bias	0.0	Constant sensor bias (mg/dL)
lag_minutes	0	Sensor reading delay (minutes)
dropout_prob	0.0	Probability of signal dropout per reading
seed	None	Random seed for reproducibility

Returns SensorReading(value: float, status: str) where status is 'ok' or 'dropout_hold'.

8.8 PumpModel

Location: iints.core.devices.models.PumpModel

Insulin pump error simulation:

Parameter	Default	Description
max_units_per_step	None	Maximum insulin per time step
quantization_units	None	Delivery quantization (e.g., 0.05 U)
dropout_prob	0.0	Probability of occlusion/dropout
delivery_noise_std	0.0	Gaussian noise on delivered amount
seed	None	Random seed for reproducibility

Returns PumpDelivery(delivered_units: float, status: str, reason: str).

8.9 High-Level API

run_simulation()

Location: iints.highlevel.run_simulation

One-line simulation runner:

```
from iints import run_simulation
from my_algorithm import MyAlgorithm

results = run_simulation(
    algorithm=MyAlgorithm(),          # instance
    scenario="stress_test.json",    # path, dict, or None
    patient_config="default_patient", # name, path, dict, or PatientProfile
    duration_minutes=720,
    time_step=5,
    seed=42,
    output_dir="./results",
    compare_baselines=True,
    export_audit=True,
    generate_report=True,
    safety_config=SafetyConfig(),
    predictor=None,
)

df = results["results"]              # pd.DataFrame
report = results["safety_report"]    # Dict
pdf_path = results["report_pdf"]     # str
```

run_full()

Location: iints.highlevel.run_full

Same as run_simulation but always enables all outputs (audit, baseline, report) and supports enable_profiling for latency measurement.

8.10 StressEvent

Location: iints.core.simulator.StressEvent

Discrete events for stress testing:

Type	Description	Example
meal	Carbohydrate intake (grams)	value=60, reported_value=50
missed_meal	Unannounced carbs (algorithm blind)	value=40
sensor_error	Inject false glucose reading	value=200.0
exercise	Start exercise session	value=0.7 (intensity), duration=45
exercise_end	End exercise (auto-generated)	n/a
ratio_change	Dynamic ISF/ICR/basal change	isf=60, icr=12, duration=120

8.11 BattleRunner

Location: iints.core.algorithms.battle_runner.BattleRunner

Runs multiple algorithms on the same patient data and ranks them by TIR. Returns a battle report dict with 'winner', 'rankings' (sorted by TIR), and detailed simulation DataFrames for each algorithm. For CLI usage, see

`iints benchmark`.

```
from iints.core.algorithms.battle_runner import BattleRunner
from iints.core.algorithms.pid_controller import PIDController
from my_algo import MyAlgo

runner = BattleRunner(
    algorithms={"PID": PIDController(), "Mine": MyAlgo()},
    patient_data=patient_df,       # DataFrame with time, glucose, carbs
    stress_events=[...],
)
report, sim_data = runner.run_battle()
print(f"Winner: {report['winner']}")
```

8.12 ClinicalMetricsCalculator

Location: iints.analysis.clinical_metrics.ClinicalMetricsCalculator

Calculates standard clinical metrics per ATTD/ADA guidelines.

Metric	Description
TIR 70-180	Time in Range (consensus target: >= 70%)
TIR 70-140	Tight Time in Range
TIR 70-110	Very Tight Time in Range
Time < 70	Hypoglycemia (target: < 4%)
Time < 54	Severe Hypoglycemia (target: < 1%)
Time > 180	Hyperglycemia (target: < 25%)
Time > 250	Severe Hyperglycemia (target: < 5%)
GMI	Glucose Management Indicator (estimated HbA1c)
CV	Coefficient of Variation (target: <= 36%)
LBGI	Low Blood Glucose Index (Kovatchev et al.)
HBGI	High Blood Glucose Index (Kovatchev et al.)

9. Example Usage

9.1 Creating a Custom Algorithm

```
from iints.api.base_algorithm import (
    InsulinAlgorithm, AlgorithmInput, AlgorithmMetadata
)

class MySmartAlgorithm(InsulinAlgorithm):
    def get_algorithm_metadata(self):
        return AlgorithmMetadata(
            name="My Smart Algorithm",
            version="1.0.0",
            author="Dr. Research",
            algorithm_type="hybrid"
        )

    def predict_insulin(self, data: AlgorithmInput):
        self.why_log = []
        target = 110
        correction = 0.0
        meal = 0.0

        # Correction bolus
        if data.current_glucose > target:
            correction = (data.current_glucose - target) / self.isf
            self._log_reason(
                f"Correction: {correction:.2f}U",
                "calculation", correction
            )

        # Meal bolus
        if data.carb_intake > 0:
            meal = data.carb_intake / self.icr
            self._log_reason(
                f"Meal bolus: {meal:.2f}U for {data.carb_intake}g",
                "calculation", meal
            )

        # IOB safety guard
        total = max(0, correction + meal - data.insulin_on_board)
        self._log_reason(
            f"After IOB deduction: {total:.2f}U",
            "safety", total
        )

        # Hypo guard
        if data.current_glucose < 80:
            total = 0.0
            self._log_reason("SUSPENDED: low glucose", "safety")

        return {"total_insulin_delivered": total}
```

9.2 Running a Simulation (Python)

```
from iints import run_simulation, PatientProfile, SafetyConfig

results = run_simulation(
    algorithm=MySmartAlgorithm(),
    patient_config=PatientProfile(
        isf=45, icr=12, basal_rate=0.9,
        initial_glucose=130,
        dawn_phenomenon_strength=10
    ),
    scenario="scenarios/stress_test_meal.json",
    duration_minutes=1440,
    seed=42,
    safety_config=SafetyConfig(max_iob=5.0),
)

print(f"Run ID: {results['run_id']}")
print(f"Report: {results['report_pdf']}")
```

9.3 Running via CLI

```
# Quickstart + preset run (generates PDF + audit)
iints quickstart --project-name iints_quickstart
cd iints_quickstart
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py

# Full run with custom scenario (CSV + audit + PDF + baseline)
iints run-full --algo algorithms/example_algorithm.py \
    --scenario-path scenarios/clinic_safe_baseline.json \
    --output-dir results/run_full \
    --seed 42

# Import CGM data from CSV
iints import-data --input-csv data/my_cgm_export.csv --data-format dexcom --output-dir resul...

# Benchmark against standard pump
iints benchmark --algo-to-benchmark algorithms/example_algorithm.py --scenarios-dir scenario...

# Generate report from existing results
iints report --results-csv results/run_full/results.csv --output-path results/run_full/clini...
```

9.4 Stress Testing with Events

```
from iints import Simulator, PatientModel, StressEvent

sim = Simulator(
    patient_model=PatientModel(initial_glucose=120),
    algorithm=MySmartAlgorithm(),
    time_step=5,
    seed=42,
)

# Standard meal (algorithm knows about it)
sim.add_stress_event(StressEvent(
    start_time=60, event_type="meal",
    value=60, reported_value=50,       # 20% undercount
    absorption_delay_minutes=15,
))

# Unannounced snack (algorithm blind)
sim.add_stress_event(StressEvent(
    start_time=240, event_type="missed_meal",
    value=30
))

# Exercise session
sim.add_stress_event(StressEvent(
    start_time=360, event_type="exercise",
    value=0.6, duration=45
))

# Sensor malfunction
sim.add_stress_event(StressEvent(
    start_time=480, event_type="sensor_error",
    value=200.0     # false high reading
))

df, safety = sim.run_batch(720)
```

9.5 Using Device Models

```
from iints import Simulator, PatientModel, SensorModel, PumpModel

# Realistic CGM with 10-min lag and 5% dropout
sensor = SensorModel(
    noise_std=5.0,    # 5 mg/dL Gaussian noise
    bias=2.0,         # +2 mg/dL constant bias
    lag_minutes=10,   # 10-minute interstitial delay
    dropout_prob=0.05,# 5% chance of signal dropout
    seed=42,
)

# Pump with quantization and occasional occlusion
pump = PumpModel(
    max_units_per_step=2.0,
    quantization_units=0.05,   # 0.05U increments
    dropout_prob=0.02,         # 2% occlusion chance
    delivery_noise_std=0.01,
    seed=42,
)

sim = Simulator(
    patient_model=PatientModel(),
    algorithm=MySmartAlgorithm(),
    sensor_model=sensor,
    pump_model=pump,
    time_step=5,
)
```

9.6 Human-in-the-Loop Callback

```
def nurse_override(context):
    """Simulates a nurse intervening during hypoglycemia."""
    if context["glucose_actual_mgdl"] < 60:
        return {
            "additional_carbs": 15,      # 15g fast-acting carbs
            "override_delivered_insulin": 0.0,
            "note": "Nurse administered 15g glucose tabs"
        }
    return None

sim = Simulator(
    patient_model=PatientModel(),
    algorithm=MySmartAlgorithm(),
    on_step=nurse_override,
    time_step=5,
)
```

10. Commercial Pump Emulators

The SDK includes behavioral emulators for three major commercial insulin pump systems. These are NOT clinical replicas - they approximate documented behavior from FDA filings and published clinical studies for benchmarking purposes only.

EMULATOR DISCLAIMER

These emulators are approximations based on public documentation. They do NOT represent the actual proprietary algorithms used by these devices. Do not use emulator output for clinical decision-making.

10.1 Medtronic 780G (SmartGuard)

Module: iints.emulation.medtronic_780g

PID auto-correction micro-boluses every 5 min, predictive low glucose suspend (PLGS), configurable target 100-120 mg/dL, max auto-basal 6.0 U/hr. References: Bergenstal et al., FDA 510(k) K193510.

10.2 Omnipod 5 (Horizon)

Module: iints.emulation.omnipod_5

Adaptive learning window (672 hours), 67 mg/dL low suspend threshold, aggressive basal adjustment, tubeless pod delivery model.

10.3 Tandem Control-IQ

Module: iints.emulation.tandem_controliq

Predictive suspend at 70 mg/dL, correction bolus when >180 and rising, sleep mode and exercise mode support, TypeZero algorithm basis.

11. Data Import and Datasets

11.1 Supported Formats

- Generic CSV (auto-detected columns)
- Dexcom Clarity export
- FreeStyle Libre export
- Nightscout REST API (requires py-nightscout)
- Official dataset registry (AIDE, PEDAP, AZT1D, HUPA-UCM)
- Bundled demo CGM data (offline, no download required)

11.2 Dataset Registry

The SDK maintains a dataset registry (data/datasets.json) with SHA-256 integrity checks. Available datasets:

Name	Description	Access	License
sample	Bundled demo CSV	Included	MIT
aide_t1d	Jaeb Center T1D data	Download	Public
pedap	Pediatric APS trial	Download	Public
azt1d	Mendeley AZ T1D	Manual	CC-BY-4.0
hupa_ucm	HUPA-UCM Diabetes Dataset	Manual	CC-BY-4.0

11.3 Import Example

```
from iints import import_cgm_csv, load_demo_dataframe

# Import from a Dexcom export
result = import_cgm_csv("clarity_export.csv", data_format="dexcom")
df = result.dataframe    # Standardized DataFrame
scenario = result.scenario   # Auto-generated scenario dict

# Load bundled demo data (no file needed)
demo_df = load_demo_dataframe()
```

12. Built-in Algorithms

PIDController [rule_based]

Module: iints.core.algorithms.pid_controller

Industry-standard PID controller. Kp=0.1, Ki=0.01, Kd=0.05, target=120 mg/dL. Logs WhyLog entries for each P, I, D component.

FixedBasalBolus [rule_based]

Module: iints.core.algorithms.fixed_basal_bolus

Fixed basal rate with correction and meal bolus. Standard clinical logic.

CorrectionBolusAlgorithm [rule_based]

Module: iints.core.algorithms.correction_bolus

Correction-only strategy. No basal modulation.

StandardPumpAlgorithm [rule_based]

Module: iints.core.algorithms.standard_pump_algo

Standard insulin pump baseline. Used as a baseline comparator.

LSTMAlgorithm [ml]

Module: iints.core.algorithms.lstm_algorithm

PyTorch LSTM wrapper. Requires [torch] extra. Loads trained model checkpoint.

HybridAlgorithm [hybrid]

Module: iints.core.algorithms.hybrid_algorithm

Combines rule-based and ML predictions with fallback mechanism.

ConstantDoseAlgorithm [mock]

Module: iints.core.algorithms.mock_algorithms

CI-safe constant dose mock (no torch dependency). Default: 0.5U.

RandomDoseAlgorithm [mock]

Module: iints.core.algorithms.mock_algorithms

CI-safe random dose mock (no torch dependency). Range: 0-1U.

13. Limitations and Known Constraints

IMPORTANT

Understanding these limitations is critical for proper interpretation of simulation results. Failure to account for these constraints may lead to overconfidence in algorithm performance.

13.1 Patient Model Limitations

- The patient model uses a simplified bilinear insulin action curve, not an FDA-approved UVA/Padova model.
- Glucose dynamics are approximated - real physiology involves hormonal feedback, hepatic glucose production, and non-linear absorption kinetics not captured here.
- Exercise effects use a linear glucose consumption model that does not capture the complex delayed glycogen depletion effects.
- Dawn phenomenon is modeled as a linear glucose rise during a fixed time window, not the complex hormonal interplay of cortisol and growth hormone.
- Inter-patient variability is limited to configurable parameters. Intra-day variability (e.g., circadian ISF changes) requires manual ratio_change events.
- The model does not simulate illness, stress hormones, alcohol, or medication interactions.

13.2 Safety Supervisor Limitations

- The supervisor is deterministic and rule-based. It does not learn or adapt over time.
- The 30-minute glucose prediction uses a heuristic (trend + IOB + COB extrapolation). This is less accurate than physiological models, especially during rapid changes.
- Safety thresholds are fixed for the duration of a run. Real clinical scenarios may require dynamic threshold adjustment.
- The supervisor cannot detect all possible failure modes (e.g., systematic sensor drift over days).

13.3 Simulation Engine Limitations

- Fixed 5-minute time step. Real CGM devices report at variable intervals, and some events occur on faster timescales.
- Sensor and pump models are statistical approximations. Actual device failure modes (e.g., compression lows, acetaminophen interference) are not modeled.
- The simulation is single-threaded. Large batch comparisons may be slow.
- AI predictor integration requires PyTorch, which is not included by default.

13.4 Regulatory and Clinical Limitations

- This SDK has NOT received FDA 510(k) clearance, CE marking, or any regulatory approval.
- Simulation results do NOT constitute clinical evidence and cannot replace clinical trials.

- The commercial pump emulators are behavioral approximations, not validated replicas.

- The SDK is in Alpha status (v0.1.x). API stability is guaranteed only for public API as defined in API_STABILITY.md.

- No formal verification or validation (V&V) has been performed according to IEC 62304 or FDA guidance.

13.5 Data Limitations

- Imported CGM data undergoes column auto-detection with fuzzy matching, which may misidentify columns in unusual formats.

- The bundled demo dataset is synthetic and does not represent real patient data distributions.

- AIDE, PEDAP, AZT1D, HUPA-UCM, and other public datasets may have their own collection biases and limitations.

14. Installation and Configuration

14.1 Installation

```
# Standard installation
pip install iints-sdk-python35

# With PyTorch support (for LSTM algorithm)
pip install iints-sdk-python35[torch]

# With Nightscout support
pip install iints-sdk-python35[nightscout]

# Full research environment
pip install iints-sdk-python35[research]

# Development installation (from source)
git clone https://github.com/python35/IINTS-SDK.git
cd IINTS-SDK
pip install -e ".[dev]"
```

14.2 System Requirements

- Python >= 3.8
- Core dependencies: numpy, pandas, pydantic, scipy, matplotlib, fpdf2, rich, typer, PyYAML
- Optional: torch >= 1.9 (ML algorithms), py-nightscout (data import), pyarrow + h5py (research)
- Platforms: macOS, Linux, Windows (tested on Darwin, Ubuntu)

14.3 Project Structure

```
iints-sdk-python35/
  src/iints/
    __init__.py           # Public API surface
    highlevel.py          # run_simulation(), run_full()
    metrics.py            # Thin metrics facade
    api/
      base_algorithm.py   # InsulinAlgorithm, AlgorithmInput, etc.
      template_algorithm.py  # Algorithm template for new users
    core/
      simulator.py        # Simulation engine
      supervisor.py       # IndependentSupervisor
      safety/
        config.py         # SafetyConfig
        input_validator.py  # InputValidator
      patient/
        models.py         # PatientModel
        profile.py        # PatientProfile
      devices/
        models.py         # SensorModel, PumpModel
      algorithms/
        pid_controller.py, fixed_basal_bolus.py, ...
        battle_runner.py  # Algorithm comparison
    emulation/
      medtronic_780g.py, omnipod_5.py, tandem_controliq.py
    analysis/
      clinical_metrics.py  # ClinicalMetricsCalculator
      reporting.py         # PDF report generator
      baseline.py          # Baseline comparison
    data/
      importer.py          # CGM data import
      universal_parser.py  # Auto-detect column formats
      nightscout.py        # Nightscout API import
    cli/
      cli.py               # Typer CLI application
    validation/
      schemas.py           # Pydantic validation models
```

14.4 CLI Quick Reference

Command	Description
iints run	Run simulation with specified algorithm
iints run-full	Full run with all outputs enabled
iints run-parallel	Batch run across scenario directory
iints new-algo	Generate algorithm template from scaffold
iints presets list	List available clinic-safe presets
iints presets run <name>	Run a named preset scenario
iints import-data	Import CGM data from CSV
iints import-nightscout	Pull data from Nightscout API

iints validate	Validate scenario/patient config files
iints benchmark	Multi-patient benchmarking
iints report	Generate PDF from existing results CSV
iints algorithms list	List all registered algorithms
iints data list	List available datasets
iints data fetch <name>	Download a registered dataset
iints check-deps	Check optional dependency status

15. Reproducibility and Audit Trail

15.1 Reproducible Runs

Every simulation run is fully reproducible. The SDK captures: a random seed (auto-generated or user-specified), the complete run configuration (algorithm class, patient parameters, scenario, safety config), platform metadata (Python version, OS, SDK version), and timestamps. All of this is written to run_metadata.json.

15.2 SHA-256 Integrity Hashing

The run_manifest.json file contains SHA-256 hashes of all output artifacts (config, metadata, results CSV, report PDF, audit files, baseline files). This allows any reviewer to verify that output files have not been modified after generation.

15.3 GPG Manifest Signing

If a GPG key is available on the system, the SDK will automatically sign the run manifest file, producing a .sig detached signature. This provides cryptographic proof of authorship and integrity.

15.4 Audit Trail Format

The audit trail records every simulation step with: time, actual glucose, glucose sent to algorithm, algorithm recommendation, delivered insulin, safety reason and trigger status, supervisor latency, sensor/pump status, and human intervention notes. Available in JSONL (one JSON object per line) and CSV formats.

16. Contributing (Open-Source Guide)

16.1 Development Setup

```
git clone https://github.com/python35/IINTS-SDK.git
cd IINTS-SDK
python -m venv .venv
source .venv/bin/activate
pip install -e ".[dev]"
make test     # Run test suite
make lint     # Run flake8
make typecheck  # Run mypy
```

16.2 Adding a New Algorithm

1. Copy src/iints/api/template_algorithm.py to a new file.

2. Rename the class and implement predict_insulin().

3. Register in pyproject.toml under [project.entry-points."iints.algorithms"].

4. Add tests under tests/algorithm/.

5. The algorithm will be auto-discovered by the CLI and BattleRunner.

16.3 Code Quality Standards

- All code must pass flake8 linting and mypy type checking.
- New features require corresponding test coverage.
- Safety-critical code changes require review by a second contributor.
- Follow existing code patterns and naming conventions.
- WhyLog reasoning must be included in all algorithm implementations.

16.4 API Stability Policy

The SDK follows Semantic Versioning (SemVer). The public API surface is defined in API_STABILITY.md. Breaking changes to public API require a minor version bump (during 0.x development) or a major version bump (post-1.0). Deprecated features will emit warnings for at least one minor version before removal.

17. Troubleshooting and FAQ

17.1 Installation Issues

Q: 'iints: command not found' after pip install

A: Your virtual environment is likely not activated, or the iints script is not on your PATH. Fix: activate your venv (source .venv/bin/activate), or install with pip install --user iints-sdk-python35. On Windows, ensure Scripts/ is on PATH.

Q: Import error for torch / PyTorch not found

A: PyTorch is an optional dependency. Install it with: pip install iints-sdk-python35[torch]. The LSTMAlgorithm and research modules require this. All other features work without torch.

Q: 'No module named iints' when running my script

A: If you installed from source (git clone), make sure you ran pip install -e ".[dev]" from the IINTS-SDK directory. If using PyPI, ensure the same venv is active.

17.2 Simulation Issues

Q: Simulation raises SimulationLimitError

A: This means your algorithm drove glucose below 40 mg/dL for 30+ consecutive minutes. This is a safety termination, not a bug. Your algorithm is producing too much insulin. Reduce aggressiveness, check IOB logic, or increase the safety_config.critical_glucose_threshold if you want to study extreme cases.

Q: My algorithm output is always zero

A: Most likely the safety supervisor is overriding your algorithm. Check the audit trail (audit_summary.json) to see which safety check is triggering. Common causes: glucose is below hypo_cutoff (70 mg/dL), IOB exceeds max_iob (4.0 U), or glucose is dropping too fast (trend_stop at -2.0 mg/dL/min).

Q: Results differ between runs

A: Make sure you set a seed. Without a seed, the SDK generates a random one for sensor noise, pump dropout, etc. Pass seed=42 (or any integer) to get identical results every time. The seed is saved in run_metadata.json.

Q: Simulation is slow for long durations

A: A 24-hour simulation (288 steps) typically takes < 1 second. If it is slow: (1) check if you have torch imported but not using GPU -- it loads slowly; (2) large batch comparisons with BattleRunner scale linearly with algorithm count; (3) enable_profiling=True adds overhead -- only use when debugging latency.

17.3 Algorithm Development Issues

Q: How do I debug what the supervisor changed?

A: The results DataFrame contains both 'algorithm_recommendation' (what your algo proposed) and 'delivered_insulin' (what was actually delivered after supervisor). Compare these columns. Also check 'safety_triggered' and 'safety_reason' columns.

Q: My algorithm does not receive carb information

A: Carbs appear in AlgorithmInput.carb_intake only for 'meal' stress events. Events of type 'missed_meal' deliberately hide carbs from the algorithm (that is the point -- simulating unannounced meals). Check your scenario file.

Q: How do I access glucose history inside my algorithm?

A: The SDK calls predict_insulin() with only the CURRENT state. To track history, use your algorithm's self.state dictionary to store previous values. Example: self.state.setdefault('history', []).append(data.current_glucose). The state persists across all steps within a single simulation run.

17.4 Data Import Issues

Q: Column auto-detection picks the wrong columns

A: Use guess_column_mapping() to inspect what the SDK detects, then pass an explicit column_mapping dict to import_cgm_csv(). You can also use the import-wizard CLI command for interactive column selection.

Q: My CSV has timestamps in a non-standard format

A: Pre-process your CSV to convert timestamps to ISO 8601 format (YYYY-MM-DD HH:MM:SS) or Unix epoch. Pandas pd.to_datetime() handles most formats. The SDK expects a parseable timestamp column.

17.5 Reporting Issues

Q: PDF report is missing charts

A: Ensure matplotlib is installed (it is a core dependency). If running on a headless server, set the matplotlib backend: import matplotlib; matplotlib.use('Agg') before importing iints.

Q: Can I customize the report layout or add my logo?

A: The ClinicalReportGenerator class in iints.analysis.reporting accepts a logo_path parameter. For deeper customization, subclass ClinicalReportGenerator and override the rendering methods.

18. Quick Reference Card

Tear-out reference for the most commonly needed information. Keep this page bookmarked.

18.1 Python: Minimal Simulation (3 lines)

```
from iints import run_simulation
from iints.core.algorithms.pid_controller import PIDController
results = run_simulation(algorithm=PIDController(), duration_minutes=720, seed=42)
```

18.2 CLI: Essential Commands

Command	What It Does
iints run-full --algo X --seed N	Full run with audit + report
iints run-wizard	Interactive research run setup
iints presets list	Show available presets
iints presets run <name>	Run preset scenario
iints new-algo	Scaffold a new algorithm file
iints import-data --input-csv F	Import CGM CSV
iints benchmark --algo-to-benchmark A	Benchmark vs baselines
iints validate --scenario F	Validate scenario JSON
iints report --results-csv F	Generate PDF from CSV
iints algorithms list	List registered algorithms
iints check-deps	Check optional dependencies

18.3 Algorithm Contract: predict_insulin()

```
# INPUT: data (AlgorithmInput)
data.current_glucose      # float, mg/dL
data.insulin_on_board     # float, Units
data.carb_intake          # float, grams (0 if no meal)
data.glucose_trend_mgdl_min  # float or None
data.predicted_glucose_30min # float or None
data.current_time         # float, minutes from start

# OUTPUT: dict (must contain 'total_insulin_delivered')
return {"total_insulin_delivered": dose}

# OPTIONAL OUTPUT KEYS:
# "basal_insulin", "bolus_insulin", "correction_bolus",
# "meal_bolus", "uncertainty", "fallback_triggered"
```

18.4 Key Safety Thresholds (Defaults)

Threshold	Default	Action
Hypo cutoff	70 mg/dL	All insulin stopped
Severe hypo	54 mg/dL	EMERGENCY mode
Trend stop	-2.0 mg/dL/min	Insulin suspended
Max IOB	4.0 Units	No additional insulin
Max bolus	5.0 Units	Hard cap per dose
Max per hour	3.0 Units	60-min rolling cap
Predicted hypo	60 mg/dL in 30min	Preemptive stop
Critical (sim end)	40 mg/dL for 30 min	SimulationLimitError

18.5 Clinical Metrics Targets (ATTD/ADA Consensus)

Metric	Target	Significance
TIR 70-180	>= 70%	Primary outcome metric
Time < 70	< 4%	Hypoglycemia burden
Time < 54	< 1%	Severe hypoglycemia
Time > 180	< 25%	Hyperglycemia burden
Time > 250	< 5%	Severe hyperglycemia
CV	<= 36%	Glucose stability

18.6 Common Import Patterns

```
# One-line imports for the most common objects
from iints import (
    run_simulation, run_full,           # High-level runners
    Simulator, StressEvent,             # Low-level control
    PatientModel, PatientProfile,       # Virtual patients
    SafetyConfig,                       # Safety tuning
    SensorModel, PumpModel,             # Device simulation
    import_cgm_csv, load_demo_dataframe, # Data import
    generate_report,                    # Reporting
)
from iints.api.base_algorithm import (
    InsulinAlgorithm, AlgorithmInput,   # Algorithm contract
    AlgorithmMetadata, WhyLogEntry,
)
from iints.analysis.clinical_metrics import ClinicalMetricsCalculator
from iints.core.algorithms.pid_controller import PIDController
from iints.core.algorithms.battle_runner import BattleRunner
```

19. Glossary

Term	Definition
APS	Artificial Pancreas System - automated insulin delivery syst
CGM	Continuous Glucose Monitor - sensor measuring interstitial g
COB	Carbs on Board - active carbohydrates still being absorbed
CV	Coefficient of Variation - glucose variability metric (SD/Me
DIA	Duration of Insulin Action - how long insulin remains active
GMI	Glucose Management Indicator - estimated HbA1c from CGM data
HBGI	High Blood Glucose Index - risk metric for hyperglycemia
ICR	Insulin-to-Carb Ratio - grams of carbs per unit of insulin
IOB	Insulin on Board - active insulin remaining from previous do
ISF	Insulin Sensitivity Factor - mg/dL drop per unit of insulin
LBGI	Low Blood Glucose Index - risk metric for hypoglycemia
PID	Proportional-Integral-Derivative - classical control algorit
PLGS	Predictive Low Glucose Suspend - proactive insulin suspensio
T1D	Type 1 Diabetes
TIR	Time in Range - percentage of time glucose is 70-180 mg/dL
WhyLog	Decision reasoning log explaining each dosing decision