

HANAZONOシステム自動最適化プロジェクト - 統合ロードマップ

+

1. プロジェクト概要

1.1 システム基本構成

- 制御システム: Raspberry Pi Zero 2 W
- オペレーティングシステム: Linux (Raspbian)
- プログラミング言語: Python 3.11
- ソーラー蓄電システム: LVYUAN製
 - インバーター: SPI-10K-U (10kW)
 - バッテリー: FLCD16-10048 × 4台 (合計20.48kWh)
 - ソーラーパネル: 現在6枚稼働 (追加6枚は保管中、将来拡張予定)
- 通信モジュール: LSW-5A8153-RS485 WiFiモジュール (Modbus対応)
 - 通信仕様: ボーレート9600bps、データビット8bit、チェックビットNone、ストップビット1bit
- ネットワーク: 家庭内LAN、Tailscaleによるセキュアリモートアクセス (IPアドレス: 100.65.197.17)

1.2 電力プラン・料金体系

- 契約: 四国電力「季節別時間帯別電灯」
- 料金区分:
 - 夜間(23:00～翌7:00): 26.00円/kWh

- ・ 昼間その他季: 37.34円/kWh
- ・ 昼間夏季(7～9月): 42.76円/kWh

1.3 運用基本方針

- ・ **基本運用方式:** タイプB（省管理型・年3回設定）
 - ・ **季節区分:**
 - ・ 冬季(12-3月): 充電電流60A、充電時間60分、出力切替SOC 60%
 - ・ 春秋季(4-6月,10-11月): 充電電流50A、充電時間45分、出力切替SOC 45%
 - ・ 夏季(7-9月): 充電電流35A、充電時間30分、出力切替SOC 35%
- ・ **補助運用方式:** タイプA（変動型）
 - ・ 特殊気象条件時や特別な需要パターン時のみ一時的に手動切替
 - ・ 晴天/雨天が3日以上続く際に対応
- ・ **季節切替推奨時期:**
 - ・ 冬季設定への切替: 12月1日頃
 - ・ 春秋季設定への切替: 4月1日頃
 - ・ 夏季設定への切替: 7月1日頃
 - ・ 春秋季設定への切替: 10月1日頃

1.4 主要な家電・電力消費パターン

- ・ **エコキュート:** ダイキン EQ46NFV（深夜に自動運転）
- ・ **食洗機:** ミーレ G 7104 C SCi（深夜に使用）
- ・ **季節家電:** エアコン（夏季・冬季に使用頻度増加）

2. 現在の実装状況

2.1 コアモジュール

モジュール名	機能	実装状況	主な依存関係
lv yuan_collec tor.py	インバーターデ ータ収集	完了	pysolarman v5
email_notifie r.py	日次レポート送 信	部分完了(修 正中)	smtplib, matplotlib
settings_mana ger.py	設定管理	完了	json
logger_util.p y	ロギング機能	設計段階	logging
main.py	制御統合	完了	-

2.2 実装済みの機能詳細

2.2.1 データ収集システム (lv yuan_collector.py)

- **15分間隔**でインバーターからの各種パラメータ自動収集
- **取得項目:**
 - バッテリーSOC(%）、電圧(V)、電流(A)
 - PV出力電力(W)
 - グリッド・負荷電力(W)
 - 充放電状態
 - 運転パラメータ（充電電流、充電時間、出力SOC設定など）
- **データ保存:** data/lv yuan_data_YYYYMMDD.json

- **通信プロトコル:** Modbus TCP (PySolarmanV5ライブラリ使用)

- **IPアドレス変更自動検出:** ネットワークスキャン機能あり

2.2.2 メール通知機能 (`email_notifier.py`)

- **日次レポート送信:** 毎朝8時に前日データのサマリーを送信
- **レポート内容:**
 - バッテリーSOC推移グラフ
 - 電力生産/消費サマリー
 - 充放電パターン分析
 - システム状態サマリー
- **エラー通知:** 異常検出時の自動アラート
- **現在の課題:** 前日データがない場合のフォールバック処理を実装中

2.2.3 システム自動化

- **cron設定:**
 - 15分ごとのデータ収集
 - 毎朝8時の日次レポート送信
- **リモートアクセス:** Tailscaleによるセキュアアクセス

2.3 プロジェクトディレクトリ構造 (現行)

```
~/lv yuan_solar_control/  
├─ lv yuan_collector.py      # データ収集モジュール  
├─ email_notifier.py        # メール通知機能  
├─ settings_manager.py      # 設定管理クラス  
├─ main.py                  # メインエントリーポイント  
├─ settings.json            # システム設定ファイル  
└─ data/                    # 収集データ保存ディレクトリ
```

```
|   └─ lvyuan_data_YYYYMMDD.json
└─ logs/                                # ログファイル保存ディレクトリ
   └─ charts/                          # 生成されたグラフの保存ディレ
      クトリ
```

3. 開発フェーズと優先タスク

3.1 フェーズ1：基盤強化とモジュール化（1-2週間）

3.1.1 優先タスク

タスク	詳細	優先度	状態
メール送信 問題修正	前日データ不存在時のフォールバック実装	高	進行中
ロギング強化	logger_util.pyの実装とモジュールへの統合	高	未着手
ディレクトリ 構造整理	機能別モジュール分割とリファクタリング	中	未着手
Tailscaleリモート管理	接続監視と自動再接続機能	低	未着手

3.1.2 メール送信問題修正の詳細実装方針

1. フォールバック検索機能の追加

- ・ 特定日付のデータファイルが存在しない場合、利用可能な最新のデータファイルを検索
- ・ 複数の保存形式（JSON/CSV）に対応
- ・ 見つからない場合のエラーハンドリング強化

2. ロギング改善

- ・ ローテーションするログファイルの導入
- ・ 詳細なエラー情報とスタックトレースの記録

- ・ エラー通知の拡張（エラーレベルに応じた対応）

3. テストプラン

- ・ 日付指定での正常ケーステスト
- ・ 前日データなしケースのフォールバックテスト
- ・ データ完全なし時の適切なエラー処理確認

3.1.3 リファクタリングと構造整理

1. 新ディレクトリ構造案

~/lv yuan_solar_control/

```
|— modules/
|   |— collector.py          # データ収集、レジスタ読み取り
|   |— notifier.py          # 通知・レポート生成
|   |— weather.py           # 天気・気温情報処理（将来）
|   └ analyzer.py           # データ分析（将来）
|— utils/
|   |— logger.py            # ロギングユーティリティ
|   |— config.py            # 設定管理
|   └ helpers.py            # 汎用ヘルパー関数
|— data/                    # データ保存
|   └ db/                   # SQLiteデータベース（将来）
|— logs/                    # ログファイル
|— templates/               # レポートテンプレート
└ web/                      # Webダッシュボード（将来）
```

2. 設定管理の統一

- ・ settings.jsonの構造改善
- ・ 環境変数対応（本番/開発環境分離）
- ・ 秘密情報（SMTPパスワードなど）の安全な管理

3.2 フェーズ2：データ基盤とシステム監視（2-3週間）

3.2.1 SQLiteデータベースへの移行

タスク	詳細	優先度	状態
データベーススキーマ設計	テーブル構造と関連の定義	高	未着手
マイグレーションスクリプト	既存データのインポート	高	未着手
ORM層実装	データアクセスレイヤーの開発	中	未着手
データ圧縮戦略実装	詳細→日次→月次の自動集約	中	未着手

3.2.2 SQLiteデータベーススキーマ設計

Copy

-- 計測データテーブル（生データ、15分間隔）

```
CREATE TABLE measurements (  
    timestamp TEXT PRIMARY KEY, -- ISO8601形式の日時  
    battery_soc INTEGER, -- バッテリーSOC (%)  
    battery_voltage REAL, -- バッテリー電圧 (V)  
    battery_current REAL, -- バッテリー電流 (A)  
    pv_voltage REAL, -- PV電圧 (V)  
    pv_current REAL, -- PV電流 (A)  
    pv_power REAL, -- PV発電量 (W)  
    load_power REAL, -- 負荷電力 (W)  
    grid_power REAL, -- グリッド電力 (W)  
    temperature REAL -- インバーター温度 (°C)  
);
```

-- 設定パラメーター履歴テーブル

```
CREATE TABLE parameter_history (  
    timestamp TEXT PRIMARY KEY, -- 設定変更日時  
    charge_current INTEGER, -- 充電電流設定  
    charge_time INTEGER, -- 充電時間設定  
    output_soc INTEGER, -- 出力soc設定  
    change_reason TEXT, -- 変更理由  
    weather TEXT, -- 変更時の天気  
    season TEXT -- 変更時の季節  
);
```

-- 天気データテーブル

```
CREATE TABLE weather_data (  
    date TEXT PRIMARY KEY, -- 日付  
    weather TEXT, -- 天気状態  
    temp_high REAL, -- 最高気温  
    temp_low REAL, -- 最低気温  
    precipitation REAL -- 降水量  
);
```

-- 日次サマリーテーブル

```
CREATE TABLE daily_summary (  
    date TEXT PRIMARY KEY, -- 日付  
    total_generation REAL, -- 総発電量 (kWh)  
    total_consumption REAL, -- 総消費量 (kWh)  
    grid_purchase REAL, -- グリッド購入量  
    (kWh)  
    grid_feed_in REAL, -- グリッド売電量  
    (kWh)
```



```

        self_consumption_rate REAL,      -- 自家消費率 (%)
        average_soc REAL,                -- 平均SOC (%)
        min_soc INTEGER,                 -- 最小SOC (%)
        max_soc INTEGER                  -- 最大SOC (%)
    );

-- 月次サマリーテーブル

CREATE TABLE monthly_summary (
    month TEXT PRIMARY KEY,              -- 年月 (YYYY-MM)
    total_generation REAL,                -- 月間総発電量
    (kWh)
    total_consumption REAL,              -- 月間総消費量
    (kWh)
    grid_purchase REAL,                  -- 月間購入量
    (kWh)
    grid_feed_in REAL,                   -- 月間売電量
    (kWh)
    self_consumption_rate REAL,          -- 月間自家消費率
    (%)
    estimated_savings REAL,              -- 推定節約額 (円)
    notes TEXT                           -- 備考
);

-- システムログテーブル

CREATE TABLE system_logs (
    timestamp TEXT,                      -- 発生日時
    log_level TEXT,                      -- ログレベル
    module TEXT,                         -- 発生モジュール

```

```
message TEXT,                                -- メッセージ
details TEXT                                  -- 詳細情報
);
```

3.2.3 データ移行と圧縮戦略

1. データ移行ステップ

- 既存のJSON/CSVファイルから一括インポート
- スキーマ検証とデータクリーニング
- インポート後の整合性チェック

2. データ圧縮ルール

- 15分間隔の詳細データは30日間保持
- 30日経過後は日次サマリーに集約
- 1年経過後は月次サマリーに集約
- システムログは重要度に応じて保持期間を設定

3.2.4 システム健全性モニタリング

1. モニター項目

- CPU使用率とメモリー消費
- ディスク使用量と残容量
- ネットワーク接続状態
- プロセス稼働状況

2. 自己修復戦略

- 通信エラー時の自動再接続
- プロセス異常終了時の自動再起動
- 定期的なデータ整合性チェック

3.3 フェーズ3：予測・分析エンジン（3-4週間）

3.3.1 気象データ統合

タスク	詳細	優先度	状態
天気API連携	気象データ取得機能実装	高	未着手
気温影響分析	気温と発電効率の相関分析	中	未着手
前日予測	翌日の発電量予測モデル	中	未着手
週間予測	7日先までの運用最適化	低	未着手

3.3.2 気象データAPIの選定案

1. 候補API:

- OpenWeatherMap（国際的、多言語対応）
- 気象庁API（日本、より詳細な地域データ）
- WeatherAPI（豊富なデータポイント、無料枠あり）

2. 取得データ:

- 当日および翌日の天気予報（晴れ/曇り/雨など）
- 気温（最高/最低/時間帯別）
- 雲量
- 日射量（可能であれば）

3.3.3 発電量予測モデル

1. 入力データ:

- 過去の天気・気温・発電量データ
- 季節要因
- 曜日/休日要因

2. モデリングアプローチ:

- scikit-learnを使用した機械学習モデル
- 初期はRandomForestなど解釈しやすいモデルから開始
- モデル評価指標：RMSE、MAE、R²

3. 予測結果の活用:

- バッテリー充電戦略の最適化
- ユーザーへの運用アドバイス生成

3.4 フェーズ4：最適化エンジンと推奨システム（4-5週間）

3.4.1 パラメータ最適化エンジン

タスク	詳細	優先度	状態
季節別最適パラメーター	季節に合わせた基本設定	高	未着手
気象条件対応	天候変化に対応する動的調整	高	未着手
使用パターン分析	消費パターンに基づく最適化	中	未着手
経済効果計算	設定変更による経済効果試算	中	未着手

3.4.2 重要パラメーター最適化ロジック

1. 対象パラメーター:

- 最大充電電流(ID 07)
- 最大充電電圧充電時間(ID 10)
- インバータ出力切替SOC(ID 62)

2. 最適化基準:

- ・ 自家消費最大化
- ・ バッテリー寿命最適化
- ・ 電気料金最小化

3. 季節・天候別の推奨設定アルゴリズム:

- ・ 基本は季節区分（冬/春秋/夏）に基づく設定
- ・ 3日以上の特異天候時は天候別調整値を適用
- ・ 負荷予測を考慮した動的調整

3.4.3 通知・レポートシステム強化

1. 日次レポートの拡張:

- ・ 前日の実績サマリー
- ・ 当日の予測と推奨設定
- ・ パラメーター変更提案（必要時）

2. 週次・月次レポート:

- ・ 週間/月間の運用サマリー
- ・ 長期トレンド分析
- ・ 最適化効果の測定結果

3. アラート条件の設定:

- ・ 異常SOC変動検出
- ・ 予想外の電力消費パターン
- ・ システム異常の早期警告

3.5 フェーズ5：ユーザーインターフェース改善（5-6週間）

3.5.1 Webダッシュボード構築

タスク	詳細	優先度	状態
-----	----	-----	----

フレームワーク選定	FlaskかDjangoの選択	中	未着手
データ可視化	リアルタイムチャート実装	中	未着手
モバイル対応	レスポンシブデザイン	低	未着手
ユーザー認証	安全なアクセス制御	低	未着手

3.5.2 Webダッシュボード機能

1. メインダッシュボード:

- ・ 現在のシステム状態概要
- ・ 主要指標のリアルタイム表示
- ・ パラメーター設定の表示と変更機能

2. データ分析ビュー:

- ・ 日/週/月/年単位での発電・消費グラフ
- ・ SOCおよびエネルギーフロー可視化
- ・ 最適化効果のビフォー・アフター比較

3. 設定・管理パネル:

- ・ システム設定の確認・変更
- ・ 通知設定のカスタマイズ
- ・ 手動でのデータエクスポート機能

3.5.3 ゲーミフィケーション要素

1. エコ達成バッジ:

- ・ 省エネ目標達成時の報酬

- ・ 連続達成ストリーク記録

2. 省エネ競争:

- ・ 過去の自己記録との比較
- ・ 季節ごとの効率改善率表示

3. 目標設定と進捗:

- ・ ユーザー設定の省エネ目標
- ・ 目標達成度の視覚的表示

3.6 フェーズ6：外部連携と拡張機能（6週間以降）

3.6.1 外部システム連携

タスク	詳細	優先度	状態
Alexa連携	音声コントロール・レポート	低	未着手
HomeAssistant	スマートホーム統合	低	未着手
MQTT	IoT標準プロトコル対応	低	未着手
APIエンドポイント	外部連携インターフェース	低	未着手

3.6.2 Alexaスキル機能案

1. 情報クエリ:

- ・ 「今日の発電量は？」
- ・ 「バッテリー残量は？」
- ・ 「今月の省エネ効果は？」

2. レポート要求:

- ・ 「昨日のエネルギーレポートを教えて」
- ・ 「今週の発電量サマリーを教えて」

3. 設定確認・変更:

- ・ 「現在のシステム設定は？」
- ・ 「明日の天気に合わせて設定を教えて」

3.6.3 高度分析機能

1. バッテリー寿命予測:

- ・ 充放電パターンの分析
- ・ 劣化モデルの適用
- ・ 最適化提案

2. 電力料金プランシミュレーション:

- ・ 異なる料金プランでの効果試算
- ・ 最適プランの提案

3. CO2削減効果計算:

- ・ 環境貢献度の定量化
- ・ 年間CO2削減量の表示

4. 技術詳細と実装ガイド

4.1 インバーターModbus通信仕様

4.1.1 通信パラメーター

- ・ シリアルポートレート: 9600bps
- ・ チェックビット: None
- ・ データビット: 8bit
- ・ ストップビット: 1bit
- ・ 通信プロトコル: Modbus TCP/RTU

4.1.2 主要レジスタマップ

レジスタアドレス	説明	単位	アクセス
0x0100	バッテリーSOC	%	読取専用
0x0101	バッテリー電圧	V × 0.1	読取専用
0x0102	バッテリー電流	A × 0.1	読取専用
0x0103	PVパネル電圧	V × 0.1	読取専用
0x0104	PVパネル電流	A × 0.1	読取専用
0x0105	PV電力	W	読取専用
0x0200	インバーター状態	ビット	読取専用
0x0201	グリッド電圧	V × 0.1	読取専用
0xE001	最大充電電流	A	読書可 (※)
0xE011	最大充電時間	分	読書可 (※)
0xE062	インバータ出力切替SOC	%	読書可 (※)

※実装上は読取のみ使用（書込は手動操作）

4.1.3 Modbus通信サンプルコード

Copy

```
from pysolarmanv5 import PySolarmanV5

def connect_inverter(ip_address, serial_number,
port=8899, mb_slave_id=1, verbose=False):
    """インバーターに接続するPySolarmanV5オブジェクトを作
```

成"""

```
    try:
        modbus = PySolarmanV5(
            address=ip_address,
            serial=serial_number,
            port=port,
            mb_slave_id=mb_slave_id,
            verbose=verbose
        )
        return modbus
    except Exception as e:
        logging.error(f"インバーター接続エラー:
{str(e)}")
        return None

def read_battery_status(modbus):
    """バッテリー状態の取得"""

    try:
        # 0x0100(SOC)、0x0101(電圧)、0x0102(電流)を一度に
読み取り

        registers =
modbus.read_holding_registers(0x0100, 3)
        if registers:
            soc = registers[0] # SOC (%)
            voltage = registers[1] * 0.1 # 電圧 (V)
            current = registers[2] * 0.1 # 電流 (A)
            return {
                "soc": soc,
                "voltage": voltage,
                "current": current
            }
    except Exception as e:
        logging.error(f"バッテリー状態取得エラー:
```

```
{str(e)}")
```

```
    return None
```

4.2 データベース設計と移行ガイド

4.2.1 SQLiteデータベース接続サンプル

Copy

```
import sqlite3
```

```
import os
```

```
def get_db_connection(db_path='data/db/  
solar_data.db'):
```

```
    """データベース接続を取得"""
```

```
    # データベースディレクトリの確保
```

```
    os.makedirs(os.path.dirname(db_path),  
exist_ok=True)
```

```
    conn = sqlite3.connect(db_path)
```

```
    conn.row_factory = sqlite3.Row # 列名でアクセス可能
```

に

```
    return conn
```

```
def setup_database():
```

```
    """データベースの初期セットアップ"""
```

```
    conn = get_db_connection()
```

```
    cursor = conn.cursor()
```

```
    # 計測データテーブル作成
```

```
    cursor.execute('''
```

```
CREATE TABLE IF NOT EXISTS measurements (  
    timestamp TEXT PRIMARY KEY,  
    battery_soc INTEGER,  
    battery_voltage REAL,  
    battery_current REAL,  
    pv_voltage REAL,
```

```

        pv_current REAL,
        pv_power REAL,
        load_power REAL,
        grid_power REAL,
        temperature REAL
    )
    '''

# その他のテーブル作成

# ...

conn.commit()
conn.close()

```

4.2.2 データ移行スクリプト概要

1. JSONからSQLiteへの変換:

- ・ 既存のJSONファイルを読み込む
- ・ データをパースして適切なテーブルに挿入
- ・ 整合性チェックを実施

2. 日次・月次サマリーの生成:

- ・ 詳細データから日次サマリーを集計
- ・ 日次データから月次サマリーを集計
- ・ 長期トレンド分析のベース作成

4.3 主要パラメーター設定ガイド

4.3.1 パラメーター変更の基本手順

1. インバーターパネルで「ENT」ボタン長押し
2. パラメーターID入力
3. 新しい値を設定
4. ID 00を選択し「ENT」で終了

4.3.2 季節・天候別の推奨設定値（タイプB）

| 季節 | 充電電流(ID 07) | 充電時間(ID 10) | 出力切替SOC(ID 62) |
|-----|-----|-----|

コピー

続きをお願いします

4.3.2 季節・天候別の推奨設定値（タイプB）

季節	充電電流 (ID 07)	充電時間 (ID 10)	出力切替 SOC(ID 62)
冬季(12-3月)	60A	60分	60%
春秋季(4-6 月,10-11月)	50A	45分	45%
夏季(7-9月)	35A	30分	35%

4.3.3 特殊条件時の調整（タイプA）

条件	充電 電流 調整	充電 時間 調整	出力 SOC 調整	適用 期間
3日以上晴天予 報	-5A	-5分	-5%	晴天期間 中
3日以上雨天予 報	+5A	+10分	+10%	雨天期間 中
猛暑日予報 (35℃以上)	-10A	-10分	-10%	猛暑期間 中
高負荷予測(来 客・イベント等)	+10A	+15分	+15%	イベント 前日～

パネル増設時	-5A	-5分	-5%	2週間観察後調整
--------	-----	-----	-----	----------

4.3.4 パラメーター変更記録テンプレート

変更日時: YYYY/MM/DD HH:MM

変更者: _____

変更理由: _____ [季節切替/天候対応/その他]

【変更前設定値】

- 充電電流 (ID 07): _____A
- 充電時間 (ID 10): _____分
- 出力SOC (ID 62): _____%

【変更後設定値】

- 充電電流 (ID 07): _____A
- 充電時間 (ID 10): _____分
- 出力SOC (ID 62): _____%

【備考】

5. 気象データと発電予測モデル

5.1 気象データ収集戦略

5.1.1 API選定基準

- **精度:** 地域特化した予報精度の高さ
- **更新頻度:** 少なくとも3時間ごとの更新
- **データ点:** 天気、気温、雲量、日射量のデータ提供
- **対応言語:** 日本語対応があるもの
- **無料枠:** 適切な無料利用枠があること

5.1.2 推奨API候補一覧

API名	長所	短所	価格体系	優先度
気象庁API	日本特化、高精度	更新遅延あり	無料	高
OpenWeatherMap	全世界対応、多機能	地域によっては粒度低	無料枠＋従量課金	中
Weather API	柔軟なデータ提供	一部地域で精度課題	無料枠＋従量課金	中
AccuWeather	予報精度が高い	API制限厳しい	従量課金のみ	低

5.1.3 気象データ取得基本実装

Copy

```
import requests
import json
import logging
from datetime import datetime, timedelta

class WeatherAPI:
    """気象データ取得クラス"""

    def __init__(self, api_key, location, settings):
        """初期化

        Args:
            api_key: API認証キー
```

```

        location: 場所情報 (緯度経度または地域コード)
        settings: 設定情報
    """
    self.api_key = api_key
    self.location = location
    self.settings = settings
    self.base_url = "https://api.example.com/v1/
forecast"
    self.cache = {}
    self.cache_expiry = {}

    def get_current_weather(self):
        """現在の天気を取得"""

        endpoint = f"{self.base_url}/current"
        params = {
            "key": self.api_key,
            "q": self.location,
            "lang": "ja"
        }

        try:
            response = requests.get(endpoint,
params=params)
            response.raise_for_status()
            data = response.json()

            # レスポンス形式によって異なる処理
            # ...

            # 必要なデータを抽出
            weather = {
                "timestamp":
datetime.now().isoformat(),
                "condition": data["current"]

```



```

["condition"] ["text"],
                "temp_c": data["current"] ["temp_c"],
                "cloud": data["current"] ["cloud"],
                "humidity": data["current"]
["humidity"],
                "wind_kph": data["current"]
["wind_kph"],
            }

            return weather

        except Exception as e:
            logging.error(f"天気データ取得エラー:
{str(e)}")
            return None

    def get_forecast(self, days=7):
        """天気予報の取得

        Args:
            days: 取得する日数 (1-14)

        Returns:
            天気予報データ辞書またはNone
        """
        # キャッシュ確認
        cache_key = f"forecast_{days}"
        if cache_key in self.cache and datetime.now()
< self.cache_expiry.get(cache_key, datetime.min):
            return self.cache[cache_key]

        endpoint = f"{self.base_url}/forecast"
        params = {
            "key": self.api_key,

```

```

        "q": self.location,
        "days": days,
        "lang": "ja"
    }

    try:
        response = requests.get(endpoint,
params=params)
        response.raise_for_status()
        data = response.json()

        # 予報データを整形
        forecast = []
        for day in data["forecast"]
["forecastday"]:
            forecast_day = {
                "date": day["date"],
                "max_temp": day["day"]
["maxtemp_c"],
                "min_temp": day["day"]
["mintemp_c"],
                "condition": day["day"]
["condition"]["text"],
                "rain_chance": day["day"]
["daily_chance_of_rain"],
                "hourly": []
            }

            # 時間別データがあれば追加
            if "hour" in day:
                for hour in day["hour"]:

forecast_day["hourly"].append({
                    "time": hour["time"],
                    "temp": hour["temp_c"],

```

```

        "condition":
hour["condition"]["text"],
        "cloud": hour["cloud"]
    })

    forecast.append(forecast_day)

    # キャッシュに保存 (1時間有効)

    self.cache[cache_key] = forecast
    self.cache_expiry[cache_key] =
datetime.now() + timedelta(hours=1)

    return forecast

except Exception as e:
    logging.error(f"天気予報取得エラー:
{str(e)}")

    return None

```

5.2 発電予測モデル設計

5.2.1 必要なデータ特徴量

1. 時間関連特徴:

- 日付
- 時刻
- 季節
- 年間通算日
- 曜日

2. 気象関連特徴:

- 天気状態 (晴れ/曇り/雨など)
- 気温
- 雲量

- 降水確率

3. 過去実績特徴:

- 同時刻帯の過去発電実績
- 同気象条件の過去発電実績
- 前日の発電パターン

5.2.2 予測モデル実装アプローチ

Copy

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import mean_squared_error,
mean_absolute_error, r2_score
import joblib
```

```
class SolarProductionPredictor:
```

```
    """太陽光発電量予測モデル"""
```

```
    def __init__(self):
```

```
        """初期化"""
```

```
        self.model = None
```

```
        self.feature_columns = None
```

```
        self.encoder = None
```

```
    def preprocess_data(self, data):
```

```
        """データ前処理
```

Args:

data: 予測に使うデータフレーム

Returns:

特徴量と目的変数のデータフレーム

```
"""
# 特徴エンジニアリング

df = data.copy()

# 時間関連特徴量

df['hour'] =
pd.to_datetime(df['timestamp']).dt.hour
df['day_of_week'] =
pd.to_datetime(df['timestamp']).dt.dayofweek
df['day_of_year'] =
pd.to_datetime(df['timestamp']).dt.dayofyear
df['month'] =
pd.to_datetime(df['timestamp']).dt.month
df['is_weekend'] = df['day_of_week'].isin([5,
6]).astype(int)

# カテゴリカル変数のエンコード

if self.encoder is None:
    self.encoder =
OneHotEncoder(sparse=False, handle_unknown='ignore')
    weather_encoded =
self.encoder.fit_transform(df[['weather']])
else:
    weather_encoded =
self.encoder.transform(df[['weather']])

weather_df = pd.DataFrame(
    weather_encoded,
    columns=[f'weather_{cat}' for cat in
self.encoder.categories_[0]]
)

df = pd.concat([df.reset_index(drop=True),
```

```

weather_df], axis=1)

    # 特徴量と目的変数を分離
    X = df.drop(['timestamp', 'weather',
'pv_power'], axis=1, errors='ignore')
    y = df['pv_power']

    self.feature_columns = X.columns

    return X, y

def train(self, data):
    """モデルの訓練

    Args:
        data: 訓練データフレーム

    Returns:
        訓練済みモデル
    """
    X, y = self.preprocess_data(data)

    # 訓練・検証データ分割
    X_train, X_val, y_train, y_val =
train_test_split(X, y, test_size=0.2,
random_state=42)

    # モデル訓練
    self.model = RandomForestRegressor(
        n_estimators=100,
        max_depth=10,
        random_state=42,
        n_jobs=-1

```

```

    )
    self.model.fit(X_train, y_train)

    # 検証

    val_pred = self.model.predict(X_val)
    mse = mean_squared_error(y_val, val_pred)
    mae = mean_absolute_error(y_val, val_pred)
    r2 = r2_score(y_val, val_pred)

    print(f"検証結果: MSE={mse:.2f}, MAE={mae:.2f},
R²={r2:.2f}")

    return self.model

def predict(self, data):
    """発電量予測

    Args:
        data: 予測用データフレーム

    Returns:
        予測発電量

    """
    if self.model is None:
        raise ValueError("モデルが訓練されていません。
train()メソッドを先に実行してください。")

    X, _ = self.preprocess_data(data)

    # 必要な特徴量のみ使用
    X = X[self.feature_columns]

    predictions = self.model.predict(X)

```

```

        return predictions

def save_model(self, filename):
    """モデルの保存"""

    if self.model is None:
        raise ValueError("保存するモデルがありません。")

    model_data = {
        'model': self.model,
        'feature_columns': self.feature_columns,
        'encoder': self.encoder
    }

    joblib.dump(model_data, filename)

def load_model(self, filename):
    """モデルの読み込み"""

    model_data = joblib.load(filename)

    self.model = model_data['model']
    self.feature_columns =
model_data['feature_columns']
    self.encoder = model_data['encoder']

```

6. 拡張機能と開発ロードマップ

6.1 Webインターフェース設計

6.1.1 フレームワーク選定

- 推奨: Flask (軽量で開発が容易)
 - Raspberry Pi Zero 2 W のリソース制約に最適
 - 必要に応じて非同期処理対応

- ・ 最小限の依存関係

6.1.2 ダッシュボード構成

1. メインダッシュボード:

- ・ リアルタイムステータス表示
- ・ 主要KPI (バッテリーSOC、発電量、消費量)
- ・ 当日の発電/消費グラフ

2. 詳細分析画面:

- ・ 期間選択による発電/消費データ表示
- ・ 比較グラフ (期間/条件別)
- ・ パラメーター変更の効果分析

3. 設定画面:

- ・ システム設定の表示/編集
- ・ 通知設定
- ・ バックアップ/リストア機能

4. 予測/推奨画面:

- ・ 発電/消費予測表示
- ・ パラメーター推奨値と理由表示
- ・ シナリオ分析 (What-if分析)

6.1.3 データ可視化ライブラリ

- ・ 推奨: Chart.js
 - ・ 軽量でレスポンス
 - ・ 多彩なチャートタイプ
 - ・ インタラクティブ機能

6.2 拡張モジュールとAPI設計

6.2.1 外部連携API

1. RESTful API:

- システム状態取得
- 履歴データ取得
- 設定変更指示

2. Webhook:

- イベント通知
- アラート発報

6.2.2 Alexaスキル設計

1. インテント設計:

- GetSystemStatus
- GetProductionSummary
- GetConsumptionSummary
- GetRecommendation
- GetAlerts

2. サンプルスクリプト:

- 「バッテリー残量はどのくらい？」
- 「今日の発電量は？」
- 「今週の省エネ効果は？」

6.3 セキュリティ対策と品質保証

6.3.1 セキュリティ対策

1. 認証と権限管理:

- ウェブインターフェース用ログイン認証
- API呼び出し用のトークン認証
- 操作権限の段階分け

2. 通信セキュリティ:

- TLS/SSL暗号化
- API呼び出し制限
- ファイアウォール設定

3. データ保護:

- 定期バックアップ
- 設定情報の暗号化
- アクセスログの保存

6.3.2 テスト戦略

1. 単体テスト:

- 各モジュールの機能テスト
- 異常系テストケース

2. 統合テスト:

- モジュール間連携テスト
- エンドツーエンドテスト

3. 性能テスト:

- 長時間動作テスト
- リソース使用量監視

6.4 開発スケジュールと優先順位

6.4.1 短期目標（1ヶ月以内）

1. メール送信問題の修正

- フォールバックロジックの実装
- エラーハンドリングの強化
- 単体テストの追加

2. システム基盤強化

- ロギング強化

- ・ コード構造の整理
- ・ Tailscaleリモート管理強化

6.4.2 中期目標（3ヶ月以内）

1. データベース移行

- ・ SQLite構造の設計と実装
- ・ 既存データの移行
- ・ データ圧縮ポリシー実装

2. 気象データ連携

- ・ API選定と実装
- ・ 予測モデル基盤構築
- ・ メール通知内容の拡張

6.4.3 長期目標（6ヶ月以内）

1. 最適化エンジン

- ・ パラメーター推奨システム
- ・ 動的調整の自動化
- ・ 経済効果の可視化

2. Webダッシュボード

- ・ 基本UI実装
- ・ データ可視化
- ・ 設定管理機能

7. メンテナンスと運用

7.1 定期メンテナンスチェックリスト

7.1.1 日次確認項目

- ・ システム通信状態
- ・ データ収集・保存状態

- ・ メール送信状況
- ・ エラーログ確認

7.1.2 週次確認項目

- ・ データバックアップの実施
- ・ 予測精度の確認
- ・ システムリソース使用状況
- ・ セキュリティログ確認

7.1.3 月次確認項目

- ・ データベース最適化
- ・ 長期トレンド分析
- ・ 設定パラメーターの適正確認
- ・ システムアップデートの確認

7.2 トラブルシューティングガイド

7.2.1 通信エラー対応

1. インバーター通信エラー:

- ・ ネットワーク接続確認
- ・ WiFiモジュール再起動
- ・ IPアドレス再確認
- ・ Modbus設定パラメーター確認

2. API通信エラー:

- ・ インターネット接続確認
- ・ API制限確認
- ・ 認証情報確認
- ・ プロキシ設定確認

7.2.2 システム障害対応

1. Raspberry Pi再起動:

Copy

```
sudo reboot
```

2.

3. サービス再起動:

Copy

```
sudo systemctl restart lvyuan_collector
```

4. `sudo systemctl restart lvyuan_notifier`

5.

6. ログ確認:

Copy

```
tail -n 100 ~/lvyuan_solar_control/logs/  
system.log
```

7.

8. ディスクスペース確認:

Copy

```
df -h
```

9.

7.3 データメンテナンス

7.3.1 バックアップ戦略

1. 自動バックアップ:

- データベースの週次ダンプ
- 設定ファイルの変更時バックアップ

2. クラウドバックアップ:

- GoogleドライブまたはDropboxへの定期自動バックアップ
- 暗号化バックアップファイルの生成

7.3.2 データクリーニング

1. データ有効期限:

- 詳細データ: 30日
- 日次集計: 365日
- 月次集計: 無期限

2. 自動アーカイブ:

- 古いデータの自動圧縮
- 低頻度アクセスストレージへの移動

8. 拡張と将来計画

8.1 システム拡張案

8.1.1 ハードウェア拡張

• センサー追加:

- 室内/室外温度センサー
- 日射量センサー
- 電力消費モニター

- **計算リソース強化:**

- より高性能なRaspberry Piへのアップグレード
- 分散計算モデルの導入

8.1.2 ソフトウェア拡張

- **高度な予測モデル:**

- ディープラーニングの導入
- 転移学習による精度向上

- **インターフェース拡張:**

- モバイルアプリ開発
- AR/VRでのデータ可視化

8.2 自動化レベル向上

8.2.1 パラメーター自動調整

- **API連携:**

- インバーターAPIでの設定自動変更
- 外部システムとのデータ連携

- **AIベース最適化:**

- 強化学習による自動パラメーターチューニング
- 異常検知と自動対応

8.2.2 プロアクティブ管理

- **予知保全:**

- バッテリー寿命予測
- 故障予測アルゴリズム

- **エネルギーマネジメント:**

- 家電と連携した需要予測
- 柔軟な充放電スケジューリング

8.3 コミュニティと共有

8.3.1 オープンソース計画

- コード共有:
 - GitHub公開リポジトリ
 - ドキュメント整備
- コミュニティ構築:
 - 同様のシステムを持つユーザーとの情報交換
 - フォーラム/Discordグループ

8.3.2 データ分析共有

- 匿名化データ:
 - パフォーマンス比較データベース
 - 地域特性に応じた最適化パターン
- 分析結果報告:
 - エネルギー効率改善レポート
 - 優れたプラクティスのケーススタディ

9. 付録とリファレンス

9.1 重要モジュールリファレンス

9.1.1 LVYUANパラメーター一覧

ID	パラメーター名	標準設定	設定範囲	調整頻度
01	AC出力ソースの優先度	SBU	SBU/SUB/UBS	通常変更なし
02	出力周波数	60Hz	50Hz/60Hz	変更不可

03	商用電源タイプ	UPS	UPS/APL	通常変更なし
04	バッテリー切替電圧	45.2V	44.0-45.2V	通常変更なし
05	商用電源切替電圧	53.2V	51.0-53.2V	通常変更なし
06	充電モード	CSO	CSO/CUB/ OSO/SNU	通常変更なし
07	最大充電電流	季節による	10-80A	季節/天候で変更
08	バッテリータイプ	L16	L16	変更不可
10	充電時間	季節による	5-900分	季節/天候で変更
11	トリクル充電電圧	57.6V	-	変更不可
62	インバータ出力切替SOC	季節による	5-100%	季節/天候で変更

9.2 参考文献・資料

9.2.1 ハードウェアドキュメント

- LVYUAN SPI-10K-Uインバーター操作マニュアル
- FLCD16-10048バッテリー仕様書
- LSW-5A8153-RS485 WiFiモジュール通信仕様書
- Raspberry Pi Zero 2 W 技術仕様書

9.2.2 ソフトウェア関連資料

- PySolarmanV5 ライブラリドキュメント
- ModbusRTU/TCP プロトコル仕様
- SQLite パフォーマンスチューニングガイド
- Flask Web開発ガイド

9.3 謝辞と貢献者

本プロジェクトは以下の協力と貢献によって成り立っています：

- プロジェクト主導：HANAZONO管理チーム
- 技術アドバイス：Solar Engineering Group
- ソフトウェア開発：Python Renewable Energy Community
- ドキュメンテーション：Technical Writing Team

このドキュメントは継続的に更新され、プロジェクト進行に合わせて拡張されます。最新版は常にプロジェクトリポジトリで確認してください。

最終更新日：2025年5月2日

ドキュメントバージョン：1.0