

# ソースコード説明書

## 目次

- [プロジェクト全体構造](#)
- [主要ファイルと役割](#)
- [技術スタック](#)
- [データフロー](#)
- [主要機能の実装箇所](#)
- [コンポーネント/モジュール詳細](#)

## 1. プロジェクト全体構造

このプロジェクトは、軌道復元システムのモダンなWEBアプリケーション実装です。以下の3つの主要ディレクトリで構成されています。

```
rail-track-app/  
├── frontend/      # Reactベースのフロントエンドアプリケーション  
├── backend/       # Express.jsベースのローカル開発用APIサーバー  
├── netlify/       # Netlify Functions用のサーバーレスAPI  
├── netlify.toml   # Netlifyのデプロイ設定  
├── package.json   # ルートレベルの依存関係とスクリプト  
└── sample-data.csv # テスト用のサンプルデータ
```

### 各ディレクトリの役割

#### frontend/

- 役割:** ユーザーインターフェースを提供するReact + TypeScriptアプリケーション
- ビルドツール:** Vite (高速な開発サーバーとビルド)
- 開発サーバー:** ポート3000で起動
- 本番ビルド:** dist/ ディレクトリに静的ファイルを出力
- APIプロキシ:** 開発時は /api/\* リクエストをlocalhost:3001にプロキシ

#### backend/

- 役割:** ローカル開発環境用のRESTful APIサーバー
- ポート:** 3001
- 用途:** 開発時のAPIエンドポイント提供、ファイルアップロード処理
- ストレージ:** ディスクベース ( uploads/ ディレクトリ)
- 注意:** 本番環境では使用されない (Netlify Functionsが代替)

#### netlify/

- 役割:** Netlifyへのデプロイ時に使用するサーバーレスAPI関数
- 実行環境:** AWS Lambda (Netlify Functions)
- ストレージ:** メモリベース (永続化なし)
- 用途:** 本番環境でのAPIエンドポイント提供
- コードの重複:** backend/src/server.js と同様の処理をサーバーレス用に最適化

## 2. 主要ファイルと役割

### ルートレベル

ファイル	役割
netlify.toml	Netlifyのビルド・デプロイ設定、リダイレクトルール定義
package.json	プロジェクト全体の依存関係、デプロイスクリプト定義
sample-data.csv	テスト用の軌道データサンプル

### Backend (backend/)

ファイル	役割
src/server.js	Express.jsアプリケーションのメインファイル。全APIエンドポイント、ミドルウェア、計算ロジックを含む
package.json	バックエンド依存関係の定義 (express, cors, multer)
uploads/	アップロードされたCSVファイルの一時保存先 (処理後削除)

### Frontend (frontend/)

ファイル	役割
index.html	SPAのエントリーHTMLファイル
vite.config.ts	Viteのビルド設定、APIプロキシ設定
tsconfig.json	TypeScriptコンパイラ設定
src/main.tsx	Reactアプリケーションのエントリーポイント
src/App.tsx	メインアプリケーションコンポーネント、状態管理とAPIコール
src/components/FileUpload.tsx	ファイルアップロードUI (ドラッグ&ドロップ対応)
src/components/ChartDisplay.tsx	Chart.jsを使用したグラフ表示コンポーネント (ピーク・異常値表示対応)
src/components/Statistics.tsx	統計情報表示コンポーネント
src/components/SpectrumAnalysis.tsx	FFTスペクトル解析コンポーネント
src/components/CorrectionSettings.tsx	カント・スラック補正設定コンポーネント
src/components/OutlierDetection.tsx	異常値検出コンポーネント
src/components/FFTFilterSettings.tsx	FFTフィルター詳細設定コンポーネント
src/*.css	各コンポーネントのスタイル定義

### Netlify Functions (netlify/)

ファイル	役割
functions/api.js	サーバーレスAPI関数。backend/src/server.jsと同等の機能をserverless-http経由で提供

## 3. 技術スタック

バックエンド

技術	バージョン	用途
Node.js	18+ (推奨)	JavaScriptランタイム環境
Express.js	^4.18.2	WEBアプリケーションフレームワーク
cors	^2.8.5	Cross-Origin Resource Sharing対応
multer	^1.4.5-lts.1	マルチパートフォームデータ(ファイルアップロード)処理
serverless-http	^3.2.0	ExpressアプリをAWS Lambda用に変換

モジュールシステム: ES Modules ( "type": "module" )

フロントエンド

技術	バージョン	用途
React	^18.2.0	UIライブラリ
React DOM	^18.2.0	ReactのDOM操作用ライブラリ
TypeScript	^5.3.3	型安全な開発言語
Vite	^5.0.8	高速ビルドツール・開発サーバー
Chart.js	^4.4.0	データ可視化ライブラリ
react-chartjs-2	^5.2.0	Chart.jsのReactラッパー
recharts	^2.10.0	スペクトル解析・補正グラフ用可視化ライブラリ
file-saver	^2.0.5	ファイルダウンロード機能
@vitejs/plugin-react	^4.2.1	Vite用Reactプラグイン

デプロイ環境

技術	用途
Netlify	静的サイトホスティング + サーバーレス関数実行
Netlify CLI	ローカル開発・デプロイツール
AWS Lambda	Netlify Functions実行基盤

4. データフロー

4.1 CSVファイルアップロードと解析の流れ

[ユーザー]  
↓ (1) CSVファイルを選択/ドロップ  
[FileUpload. tsx]

```
↓ (2) File オブジェクトをApp.tsxに渡す
[App.tsx - handleFileUpload]
↓ (3) FormDataを作成してPOSTリクエスト
[POST /api/upload]
↓ (4) multerがファイルを受信・検証
[backend/server.js or netlify/api.js]
↓ (5) CSVテキストを読み込み
[parseCSV関数]
↓ (6) カンマ区切りでパース → {distance, irregularity}[]
[calculateStatistics関数]
↓ (7) 統計値を計算 (min, max, avg, stdDev)
[APIレスポンス]
↓ (8) JSON形式でデータと統計を返却
[App.tsx]
↓ (9) originalData stateを更新
[ChartDisplay.tsx & Statistics.tsx]
↓ (10) UIを更新・グラフと統計を表示
[ユーザーに表示]
```

#### データ構造の変換

CSVファイル:

```
0.0, 2.5
0.5, 2.8
1.0, 3.2
```

parseCSV処理後:

```
[
  { distance: 0.0, irregularity: 2.5 },
  { distance: 0.5, irregularity: 2.8 },
  { distance: 1.0, irregularity: 3.2 }
]
```

APIレスポンス:

```
{
  success: true,
  filename: "sample.csv",
  dataPoints: 3,
  data: [...],
  statistics: {
    min: 2.5,
    max: 3.2,
    avg: 2.83,
    stdDev: 0.29
  }
}
```

## 4.2 波形復元処理の流れ

```
[ユーザー]
  ↓ (1) 「波形を復元」ボタンをクリック
[App.tsx - handleRestoreWaveform]
  ↓ (2) originalDataをリクエストボディに含めてPOST
[POST /api/restore-waveform]
  ↓ (3) filterType='simple' で3点移動平均フィルタを適用
[backend/server.js or netlify/api.js]
  ↓ (4) 各点について前後のデータを平均化
[復元アルゴリズム]
  ↓ (5) 元データと復元データの統計をそれぞれ計算
[APIレスポンス]
  ↓ (6) 元データと復元データの両方を返却
[App.tsx]
  ↓ (7) restoredData stateを更新
[ChartDisplay.tsx]
  ↓ (8) 元データと復元データの2本のラインを表示
[ユーザーに表示]
```

### 復元アルゴリズム (3点移動平均)

```
// 境界点（最初と最後）はそのまま
if (i === 0 || i === data.length - 1) {
  restoredData[i] = originalData[i]
} else {
  // 前後の点と平均を取る
  restoredData[i].irregularity =
    (data[i-1].irregularity + data[i].irregularity + data[i+1].irregularity) / 3
}
```

## 4.3 状態管理フロー (React)

App.tsxで管理される主要な状態:

```
const [originalData, setOriginalData] = useState<DataSet | null>(null)
const [restoredData, setRestoredData] = useState<DataSet | null>(null)
const [loading, setLoading] = useState(false)
```

状態遷移:

1. 初期状態: すべてnull/false
2. ファイルアップロード開始: loading=true
3. アップロード成功: originalData設定、restoredData=null、loading=false
4. 波形復元開始: loading=true
5. 復元成功: restoredData設定、loading=false

---

## 5. 主要機能の実装箇所

## 5.1 CSVファイルアップロード機能

フロントエンド実装:

- ファイル: frontend/src/components/FileUpload.tsx
- 主要機能:
  - ドラッグ&ドロップ対応 ( onDragOver , onDrop イベント)
  - ファイル選択ダイアログ ( <input type="file"> )
  - CSV拡張子の検証
  - ローディング状態の表示
- Props:

```
interface FileUploadProps {  
  onFileUpload: (file: File) => void // ファイル選択時のコールバック  
  loading: boolean // ローディング状態  
}
```

バックエンド実装:

- ファイル: backend/src/server.js (行18-38)
- 主要機能:
  - multerによるファイル受信設定
  - ディスクストレージ設定 (開発環境)
  - ファイル名の一意化 (タイムスタンプ + ランダム値)
  - MIMEタイプ検証 ( text/csv )

Netlify Functions実装:

- ファイル: netlify/functions/api.js (行10-20)
- 主要機能:
  - メモリストレージ設定 (サーバーレス環境)
  - 同様のMIMEタイプ検証

## 5.2 CSVパース機能

実装箇所:

- backend/src/server.js (行41-57)
- netlify/functions/api.js (行23-39)

処理ロジック:

```
function parseCSV(csvText) {  
  const lines = csvText.split('\n').filter(line => line.trim())  
  const data = []  
  
  for (const line of lines) {  
    const values = line.split(',').map(v => v.trim())  
    if (values.length >= 2) {  
      const distance = parseFloat(values[0])  
      const irregularity = parseFloat(values[1])  
      if (!isNaN(distance) && !isNaN(irregularity)) {
```

```

        data.push({ distance, irregularity })
    }
}
}

return data
}

```

#### 特徴:

- 改行で分割、空行をフィルタリング
- カンマで値を分割、前後の空白を除去
- 2列以上のデータのみ処理
- 数値変換に失敗したデータは除外 (NaNチェック)

### 5.3 統計計算機能

#### 実装箇所:

- backend/src/server.js (行86-101)
- netlify/functions/api.js (行69-82)

#### 計算内容:

```

function calculateStatistics(data) {
    const values = data.map(d => d.irregularity)

    // 基本統計量
    const min = Math.min(...values)
    const max = Math.max(...values)
    const avg = values.reduce((a, b) => a + b, 0) / values.length

    // 標準偏差
    const variance = values.reduce((sum, val) =>
        sum + Math.pow(val - avg, 2), 0
    ) / values.length
    const stdDev = Math.sqrt(variance)

    return { min, max, avg, stdDev }
}

```

#### 数学的定義:

- 平均値:  $\mu = \Sigma x / n$
- 分散:  $\sigma^2 = \Sigma (x - \mu)^2 / n$
- 標準偏差:  $\sigma = \sqrt{\sigma^2}$

### 5.4 グラフ表示機能

#### 実装箇所:

- frontend/src/components/ChartDisplay.tsx

#### 使用ライブラリ:

- Chart.js (コアライブラリ)
- react-chartjs-2 (Reactラッパー)

主要設定:

```
const options = {
  responsive: true,                // レスポンシブ対応
  maintainAspectRatio: false,     // アスペクト比固定を解除
  plugins: {
    legend: { position: 'top' },   // 凡例を上部に配置
    title: {
      display: true,
      text: '軌道狂い量の推移'
    },
    tooltip: {                    // ツールチップのカスタマイズ
      callbacks: {
        label: (context) => `${context.parsed.y.toFixed(2)} mm`
      }
    }
  },
  scales: {
    x: {
      title: { text: '距離 (m)' },
      ticks: { maxTicksLimit: 20 } // X軸の表示数制限
    },
    y: {
      title: { text: '軌道狂い量 (mm)' }
    }
  }
}
```

データセット構成:

- 元データ: 青緑色 (rgb(75, 192, 192))
- 復元データ: ピンク色 (rgb(255, 99, 132))
- 両方とも線グラフ ( Line コンポーネント)

## 5.5 波形復元機能

実装箇所:

- backend/src/server.js (行168-210)
- netlify/functions/api.js (行204-246, 248-289)

復元アルゴリズム:

```
app.post('/api/restore-waveform', (req, res) => {
  const { data, filterType = 'simple' } = req.body

  let restoredData = [...data]

  if (filterType === 'simple') {
    // 3点移動平均フィルタ
```



```

    restoredData = data.map((point, i) => {
      if (i === 0 || i === data.length - 1) {
        return point // 境界点はそのまま
      }
      const avgIrregularity =
        (data[i-1].irregularity + point.irregularity + data[i+1].irregularity) / 3
      return { ...point, irregularity: avgIrregularity }
    })
  }

  // 元データと復元データの統計を計算
  const originalStats = calculateStatistics(data)
  const restoredStats = calculateStatistics(restoredData)

  res.json({
    success: true,
    original: { data, statistics: originalStats },
    restored: { data: restoredData, statistics: restoredStats },
    filterType
  })
})

```

#### フィルタの特性:

- **効果:** 高周波ノイズの除去、波形の平滑化
- **制限:** 境界点は処理しない (データ欠損を防ぐため)
- **将来の拡張:** filterTypeパラメータで異なるフィルタを実装可能

## 5.6 ピーク検出機能

#### 実装箇所:

- backend/src/algorithms/peakDetection.js
- APIエンドポイント: POST /api/detect-peaks

#### 主要機能:

- 極大値・極小値の検出
- ローカルピーク検出
- 統計的異常値の検出

#### フロントエンド実装:

- frontend/src/App.tsx - ピーク検出API呼び出し
- frontend/src/components/ChartDisplay.tsx - ピークの可視化 (赤=極大、青=極小)

## 5.7 異常値検出機能

#### 実装箇所:

- backend/src/algorithms/peakDetection.js
- APIエンドポイント: POST /api/detect-outliers

#### 主要機能:

- 標準偏差ベースの異常値検出

- シグマ倍率の調整可能 (1.0~5.0)
- 重症度判定 (deviation値)

#### フロントエンド実装:

- `frontend/src/components/OutlierDetection.tsx` - UI + API連携
- シグマ倍率スライダー
- 異常値テーブル表示 (色分け: 赤>オレンジ>黄>緑)

## 5.8 FFTスペクトル解析機能

#### 実装箇所:

- `backend/src/algorithms/fft.js`
- APIエンドポイント: `POST /api/analyze-spectrum`

#### 主要機能:

- 高速フーリエ変換 (FFT) による周波数解析
- パワースペクトル計算
- 支配的な周波数成分の抽出

#### フロントエンド実装:

- `frontend/src/components/SpectrumAnalysis.tsx` - スペクトル可視化
- Rechartsによるパワースペクトルグラフ
- 支配的周波数テーブル表示

## 5.9 カント・スラック補正機能

#### 実装箇所:

- `backend/src/algorithms/corrections.js`
- APIエンドポイント: `POST /api/apply-corrections`

#### 主要機能:

- カント補正 (曲線部の傾斜補正)
- スラック補正 (レール間隔補正)
- 補正係数の個別指定

#### フロントエンド実装:

- `frontend/src/components/CorrectionSettings.tsx`
- 補正係数入力フォーム
- 補正前後の比較グラフ
- 補正統計情報表示

## 5.10 MTT軌道評価機能

#### 実装箇所:

- `backend/src/algorithms/mttCalculation.js`
- APIエンドポイント: `POST /api/calculate-mtt`

#### 主要機能:

- BC値計算 (軌道狂いの基準値)
- CD値計算 (軌道変化量の基準値)

- MTT総合評価

#### フロントエンド実装:

- `frontend/src/App.tsx` - MTTタブ
- BC/CD値の表示

### 5.11 FFTフィルター設定機能

#### 実装箇所:

- `backend/src/algorithms/fft.js`
- APIエンドポイント: `POST /api/apply-filter` (`filterType: fft_*`)

#### フィルタータイプ:

- `fft_lowpass` : 低域通過フィルター
- `fft_highpass` : 高域通過フィルター
- `fft_bandpass` : 帯域通過フィルター

#### フロントエンド実装:

- `frontend/src/components/FFTFilterSettings.tsx`
- カットオフ周波数スライダー
- 帯域幅設定

### 5.12 相関係数計算機能

#### 実装箇所:

- `backend/src/server.js` (行59-84, 142-166)
- `netlify/functions/api.js` (行42-66, 153-177, 179-202)

#### 計算式:

```
function calculateCorrelation(data1, data2) {
  const n = Math.min(data1.length, data2.length)

  let sumX = 0, sumY = 0, sumXX = 0, sumYY = 0, sumXY = 0

  for (let i = 0; i < n; i++) {
    const x = data1[i].irregularity
    const y = data2[i].irregularity
    sumX += x
    sumY += y
    sumXX += x * x
    sumYY += y * y
    sumXY += x * y
  }

  const avgX = sumX / n
  const avgY = sumY / n

  const numerator = sumXY - n * avgX * avgY
  const denominator = Math.sqrt((sumXX - n * avgX * avgX) * (sumYY - n * avgY * avgY))
```

```
    if (denominator === 0) return 0
    return numerator / denominator
}
```

数学的定義 (ピアソン相関係数):

$$r = (\text{SUMXY} - n \cdot \mu_x \cdot \mu_y) / \sqrt{[(\text{SUMXX} - n \cdot \mu_x^2)(\text{SUMYY} - n \cdot \mu_y^2)]}$$

相関の判定基準:

- $r > 0.7$ : 強い正の相関
- $0.3 < r \leq 0.7$ : 中程度の正の相関
- $-0.3 \leq r \leq 0.3$ : 弱い相関
- $-0.7 \leq r < -0.3$ : 中程度の負の相関
- $r < -0.7$ : 強い負の相関

---

## 6. コンポーネント/モジュール詳細

### 6.1 Backend: server.js

ファイルパス: backend/src/server.js

構成要素:

ミドルウェア設定 (行14-16)

```
app.use(cors())           // CORS対応
app.use(express.json())    // JSONボディパース
```

multer設定 (行18-38)

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, path.join(__dirname, '../uploads'))
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9)
    cb(null, uniqueSuffix + '-' + file.originalname)
  }
})

const upload = multer({
  storage: storage,
  fileFilter: (req, file, cb) => {
    if (file.mimetype === 'text/csv' || file.originalname.endsWith('.csv')) {
      cb(null, true)
    } else {
      cb(new Error('CSVファイルのみアップロード可能です'))
    }
  }
})
```

```
}  
})
```

## ヘルパー関数

**parseCSV** (行41-57):

- 役割: CSVテキストをJavaScriptオブジェクト配列に変換
- 入力: CSV文字列
- 出力: {distance: number, irregularity: number}[]

**calculateCorrelation** (行59-84):

- 役割: 2つのデータセットの相関係数を計算
- 入力: data1, data2 (TrackData配列)
- 出力: 相関係数 (-1 ~ 1)

**calculateStatistics** (行86-101):

- 役割: 基本統計量を計算
- 入力: data (TrackData配列)
- 出力: {min, max, avg, stdDev}

## APIエンドポイント

**GET /api/health** (行104-106):

- 役割: ヘルスチェック
- レスポンス: {status: 'ok', message: '...'}

**POST /api/upload** (行109-140):

- 役割: CSVファイルのアップロードと解析
- ミドルウェア: upload.single('file')
- 処理:
  1. ファイル受信確認
  2. CSVテキスト読み込み ( fs.readFile )
  3. パース処理
  4. 統計計算
  5. ファイル削除 ( fs.unlink )
- レスポンス: データと統計情報

**POST /api/calculate-correlation** (行143-166):

- 役割: 相関係数の計算
- 入力: {data1, data2}
- 出力: 相関係数と説明文

**POST /api/restore-waveform** (行169-210):

- 役割: 波形復元処理
- 入力: {data, filterType}
- 出力: 元データと復元データの両方

## サーバー起動 (行212-215)

```
app.listen(PORT, () => {
  console.log(`Rail Track API server is running on http://localhost:${PORT}`)
})
```

## 6.2 Netlify Functions: api.js

ファイルパス: `netlify/functions/api.js`

backend/src/server.jsとの違い:

- **serverless-http**: ExpressアプリをLambda関数に変換 (行291)
- **メモリストレージ**: ファイルを `req.file.buffer` で処理 (行101, 130)
- **ファイル削除不要**: 一時ファイルが作成されない
- **重複エンドポイント**: `/api/upload` と `/upload` の両方を提供 (互換性のため)

エクスポート:

```
export const handler = serverless(app)
```

このhandlerがNetlify Functionsによって実行されます。

## 6.3 Frontend: App.tsx

ファイルパス: `frontend/src/App.tsx`

主要な型定義:

```
export interface TrackData {
  distance: number
  irregularity: number
}

export interface DataSet {
  data: TrackData[]
  statistics: {
    min: number
    max: number
    avg: number
    stdDev: number
  }
  filename?: string
}
```

状態管理:

```
const [originalData, setOriginalData] = useState<DataSet | null>(null)
const [restoredData, setRestoredData] = useState<DataSet | null>(null)
const [loading, setLoading] = useState(false)
```

主要関数:

handleFileUpload (行28-57):

```
const handleFileUpload = async (file: File) => {
  setLoading(true)
  const formData = new FormData()
  formData.append('file', file)

  try {
    const response = await fetch('/api/upload', {
      method: 'POST',
      body: formData,
    })

    const result = await response.json()

    if (result.success) {
      setOriginalData({
        data: result.data,
        statistics: result.statistics,
        filename: result.filename
      })
      setRestoredData(null) // 前の復元データをクリア
    } else {
      alert('エラー: ' + (result.error || '不明なエラー'))
    }
  } catch (error) {
    console.error('Upload error:', error)
    alert('アップロードエラーが発生しました')
  } finally {
    setLoading(false)
  }
}
```

handleRestoreWaveform (行59-92):

```
const handleRestoreWaveform = async () => {
  if (!originalData) return

  setLoading(true)
  try {
    const response = await fetch('/api/restore-waveform', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        data: originalData.data,
        filterType: 'simple'
      }),
    })
  }
```

```

const result = await response.json()

if (result.success) {
  setRestoredData({
    data: result.restored.data,
    statistics: result.restored.statistics,
    filename: originalData.filename + ' (復元)'
  })
} else {
  alert('エラー: ' + (result.error || '不明なエラー'))
}
} catch (error) {
  console.error('Restore error:', error)
  alert('波形復元エラーが発生しました')
} finally {
  setLoading(false)
}
}

```

UIレンダリング構造:

```

<div className="App">
  <header>タイトル</header>
  <div className="container">
    /* アップロードセクション */
    <section className="upload-section">
      <FileUpload onFileUpload={handleFileUpload} loading={loading} />
    </section>

    /* データがある場合のみ表示 */
    {originalData && (
      <>
        /* グラフセクション */
        <section className="chart-section">
          <ChartDisplay
            originalData={...}
            restoredData={...}
            peaks={...}      // ピーク表示対応
            outliers={...}   // 異常値表示対応
          />
        </section>

        /* 統計セクション */
        <section className="stats-section">
          <Statistics title="元データ" statistics={...} />
          {restoredData && <Statistics title="復元データ" statistics={...} />}
        </section>

        /* アクションセクション */
        <section className="actions-section">
          <button onClick={handleRestoreWaveform}>波形を復元</button>
        </section>
      </>
    )}
  </div>
</div>

```



```

</section>

{/* フィルター設定セクション */}
{/* FFTフィルターの場合は詳細設定を表示 */}
{(filterType === 'fft_lowpass' || ...) && (
  <FFTFilterSettings onSettingsChange={setFftSettings} />
)}

{/* 高度な解析セクション（タブ形式）*/}
<section className="advanced-section">
  <div className="advanced-tabs">
    <button className={advancedTab === 'peaks' ? 'active' : ''}>
      📌 ピーク検出
    </button>
    <button className={advancedTab === 'outliers' ? 'active' : ''}>
      🚨 異常値検出
    </button>
    <button className={advancedTab === 'spectrum' ? 'active' : ''}>
      📊 スペクトル解析
    </button>
    <button className={advancedTab === 'correction' ? 'active' : ''}>
      🛠️ カント・スラック補正
    </button>
    <button className={advancedTab === 'mtt' ? 'active' : ''}>
      📈 MTT評価
    </button>
    <button className={advancedTab === 'export' ? 'active' : ''}>
      📄 エクスポート
    </button>
  </div>

  {/* タブコンテンツ */}
  {advancedTab === 'peaks' && <PeakDetection ... />}
  {advancedTab === 'outliers' && <OutlierDetection ... />}
  {advancedTab === 'spectrum' && <SpectrumAnalysis ... />}
  {advancedTab === 'correction' && <CorrectionSettings ... />}
  {advancedTab === 'mtt' && <MTTResults ... />}
  {advancedTab === 'export' && <ExportButtons ... />}
</section>
</>
)}

{/* 空の状態 */}
{!originalData && (
  <div className="empty-state">
    <p>CSVファイルをアップロードしてください</p>
  </div>
)}
</div>
<footer>フッター情報</footer>
</div>

```

## 6.4 Frontend: FileUpload.tsx

ファイルパス: frontend/src/components/FileUpload.tsx

Props定義:

```
interface FileUploadProps {  
  onFileUpload: (file: File) => void // ファイル選択時のコールバック  
  loading: boolean // ローディング状態  
}
```

主要機能実装:

ファイル選択ハンドラ (行12-17):

```
const handleFileChange = (e: React.ChangeEvent<HTMLInputElement>) => {  
  const file = e.target.files?.[0]  
  if (file) {  
    onFileUpload(file)  
  }  
}
```

ドラッグオーバーハンドラ (行19-22):

```
const handleDragOver = (e: React.DragEvent) => {  
  e.preventDefault() // デフォルトのドロップ動作を防ぐ  
  e.stopPropagation() // 親要素への伝播を防ぐ  
}
```

ドロップハンドラ (行24-34):

```
const handleDrop = (e: React.DragEvent) => {  
  e.preventDefault()  
  e.stopPropagation()  
  
  const file = e.dataTransfer.files?.[0]  
  if (file && file.name.endsWith('.csv')) {  
    onFileUpload(file)  
  } else {  
    alert('CSVファイルのみアップロード可能です')  
  }  
}
```

クリックハンドラ (行36-38):

```
const handleClick = () => {  
  fileInputRef.current?.click() // 非表示のinputをプログラムでクリック  
}
```

UI構造:

```

<div className="file-upload">
  <div className="drop-zone" onDragOver={...} onDrop={...} onClick={...}>
    <input type="file" ref={fileInputRef} onChange={...} accept=".csv" style={{display: 'none'}} />
    <div className="drop-zone-content">
      {loading ? (
        <div className="spinner"></div> /* ローディング表示 */
      ) : (
        <>
          <svg>アップロードアイコン</svg>
          <p>CSVファイルをドラッグ&ドロップ</p>
          <button>ファイルを選択</button>
        </>
      )}
    </div>
  </div>
</div>

```

## 6.5 Frontend: ChartDisplay.tsx

ファイルパス: frontend/src/components/ChartDisplay.tsx

Chart.js登録 (行15-23):

```

ChartJS.register(
  CategoryScale, // X軸カテゴリスケール
  LinearScale,   // Y軸線形スケール
  PointElement,  // ポイント要素
  LineElement,   // ライン要素
  Title,         // タイトルプラグイン
  Tooltip,       // ツールチッププラグイン
  Legend         // 凡例プラグイン
)

```

Props定義:

```

interface ChartDisplayProps {
  originalData: TrackData[]
  restoredData?: TrackData[] // オプション
}

```

データセット作成ロジック (行36-60):

```

const labels = originalData.map(d => d.distance.toFixed(1))

const datasets = [
  {
    label: '元データ (Original)',
    data: originalData.map(d => d.irregularity),
    borderColor: 'rgb(75, 192, 192)',
    backgroundColor: 'rgba(75, 192, 192, 0.5)',
  },

```

```

    tension: 0.1,      // 線の滑らかさ
    pointRadius: 2,    // ポイントのサイズ
    pointHoverRadius: 5, // ホバー時のポイントサイズ
  }
]

if (restoredData) {
  datasets.push({
    label: '復元データ (Restored)',
    data: restoredData.map(d => d.irregularity),
    borderColor: 'rgb(255, 99, 132)',
    backgroundColor: 'rgba(255, 99, 132, 0.5)',
    tension: 0.1,
    pointRadius: 2,
    pointHoverRadius: 5,
  })
}

```

チャートオプション (行67-111):

- **responsive**: ウィンドウサイズに応じて自動調整
- **maintainAspectRatio: false**: コンテナに合わせてサイズ変更
- **plugins.legend**: 凡例を上部に配置
- **plugins.title**: グラフタイトル設定
- **plugins.tooltip.callbacks**: ツールチップのカスタマイズ (mm単位表示)
- **scales.x**: X軸設定 (距離)
- **scales.y**: Y軸設定 (軌道狂い量)

レンダリング:

```

<div className="chart-display">
  <div className="chart-container">
    <Line data={data} options={options} />
  </div>
</div>

```

## 6.6 Frontend: Statistics.tsx

ファイルパス: frontend/src/components/Statistics.tsx

Props定義:

```

interface StatisticsProps {
  title: string
  statistics: {
    min: number
    max: number
    avg: number
    stdDev: number
  }
}

```

コンポーネント構造:

```
<div className="statistics">
  <h3>{title}</h3>
  <div className="stat-item">
    <span className="stat-label">最小値 (Min):</span>
    <span className="stat-value">{statistics.min.toFixed(2)} mm</span>
  </div>
  <div className="stat-item">
    <span className="stat-label">最大値 (Max):</span>
    <span className="stat-value">{statistics.max.toFixed(2)} mm</span>
  </div>
  <div className="stat-item">
    <span className="stat-label">平均値 (Avg):</span>
    <span className="stat-value">{statistics.avg.toFixed(2)} mm</span>
  </div>
  <div className="stat-item">
    <span className="stat-label">標準偏差 (StdDev):</span>
    <span className="stat-value">{statistics.stdDev.toFixed(2)} mm</span>
  </div>
</div>
```

数値フォーマット: すべての値を小数点2桁で表示 ( `toFixed(2)` )

## 6.7 設定ファイル

`vite.config.ts`

ファイルパス: `frontend/vite.config.ts`

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    port: 3000,           // 開発サーバーポート
    proxy: {
      '/api': {
        target: 'http://localhost:3001', // バックエンドAPI
        changeOrigin: true
      }
    }
  }
})
```

プロキシの役割:

- 開発時に `http://localhost:3000/api/*` へのリクエストを `http://localhost:3001/api/*` に転送
- CORS問題を回避
- 本番環境ではNetlifyのリダイレクトルールが代替

netlify.toml

ファイルパス: netlify.toml

```
[build]
  base = "frontend"      # ビルドのベースディレクトリ
  publish = "dist"        # 公開する静的ファイルのディレクトリ

[functions]
  directory = "netlify/functions" # サーバーレス関数のディレクトリ
  node_bundler = "esbuild"        # バンドラー設定

[[redirects]]
  from = "/api/*"
  to = "/.netlify/functions/api/:splat" # /api/* → Netlify Functions
  status = 200
  force = true

[[redirects]]
  from = "/*"
  to = "/index.html" # SPAのルーティング対応（すべてをindex.htmlに）
  status = 200
```

リダイレクトルールの意味:

- 1. /api/\* へのリクエストはNetlify Functionsに転送
- 2. それ以外のすべてのリクエストは index.html に転送 (React Routerなどのクライアントサイドルーティング対応)

## 補足情報

### 開発とデプロイの違い

項目	開発環境	本番環境 (Netlify)
フロントエンド	Vitedev server (localhost:3000)	Netlifyの静的ホスティング
バックエンド	Express server (localhost:3001)	Netlify Functions (AWS Lambda)
APIプロキシ	Viteのproxy設定	Netlifyのリダイレクトルール
ファイルストレージ	ディスク (uploads/)	メモリ (req.file.buffer)
コード	backend/src/server.js	netlify/functions/api.js

### コードの重複について

backend/src/server.js と netlify/functions/api.js は多くのコードが重複していますが、これは以下の理由によります:

- 1. 開発環境とデプロイ環境の分離: ローカルでの高速な開発とサーバーレス環境での動作を両立
- 2. ストレージ戦略の違い: ディスクベース vs メモリベース
- 3. デプロイの独立性: フロントエンドとバックエンドを個別にデプロイ可能

**改善案:** 共通ロジック (parseCSV, calculateStatistics, calculateCorrelationなど) を別ファイルに抽出し、両方からインポートすることでDRY原則に従うことが可能。

### セキュリティ考慮事項

現在の実装では以下のセキュリティ対策が不足しています:

1. **ファイルサイズ制限:** 大きすぎるファイルのアップロードを防ぐ
2. **レート制限:** API呼び出しの頻度制限
3. **入力検証:** CSVデータの形式・内容の厳密な検証
4. **認証・認可:** ユーザー認証機能がない
5. **CSRFトークン:** クロスサイトリクエストフォージェリ対策

本番環境では上記の対策を追加することを推奨します。

### パフォーマンス最適化のポイント

1. **フロントエンド:**
  - Chart.jsのデータポイント数制限 (1000点以上の場合はサンプリング)
  - React.memoを使用したコンポーネントの再レンダリング抑制
  - 大きなデータセットの場合はvirtualizationを検討
2. **バックエンド:**
  - ストリーミングCSVパース (大ファイル対応)
  - 計算結果のキャッシング
  - Worker Threadsを使用した並列処理
3. **Netlify Functions:**
  - コールドスタート対策 (関数のウォームアップ)
  - メモリ設定の最適化
  - タイムアウト設定の調整

---

---

## 7. 新規追加コンポーネント詳細

### 7.1 SpectrumAnalysis.tsx

ファイルパス: frontend/src/components/SpectrumAnalysis.tsx

機能:

- FFTスペクトル解析の実行と結果表示
- パワースペクトルグラフ (Recharts使用)
- 支配的周波数成分のテーブル表示

Props:

```
interface SpectrumAnalysisProps {
  data: TrackData[]
}
```

### 7.2 CorrectionSettings.tsx

ファイルパス: frontend/src/components/CorrectionSettings.tsx

機能:

- カント補正・スラック補正の係数設定
- 補正前後の比較グラフ (Recharts使用)
- 補正効果の統計情報表示

Props:

```
interface CorrectionSettingsProps {  
  data: TrackData[]  
  onCorrectionApplied: (correctedData: TrackData[]) => void  
}
```

### 7.3 OutlierDetection.tsx

ファイルパス: frontend/src/components/OutlierDetection.tsx

機能:

- 統計的異常値の検出
- シグマ倍率の調整 (1.0~5.0)
- 異常値リストの表示 (重症度別色分け)

Props:

```
interface OutlierDetectionProps {  
  data: TrackData[]  
  onOutliersDetected: (outliers: Outlier[]) => void  
}
```

### 7.4 FFTFilterSettings.tsx

ファイルパス: frontend/src/components/FFTFilterSettings.tsx

機能:

- FFTフィルタータイプ選択 (lowpass/highpass/bandpass)
- カットオフ周波数の設定
- 帯域幅の調整 (bandpass時)

Props:

```
interface FFTFilterSettingsProps {  
  onSettingsChange: (settings: FFTSettings) => void  
  disabled: boolean  
}  
  
interface FFTSettings {  
  filterType: 'fft_lowpass' | 'fft_highpass' | 'fft_bandpass'  
  cutoffFreq: number  
  lowCutoff: number  
}
```



```
highCutoff: number
}
```

## 8. VB版との機能比較と未実装機能

このセクションでは、元のVB6.0システムと現在のWebアプリケーションの機能を比較し、まだ実装できていない機能を明示します。

### 8.1 実装済み機能

機能カテゴリ	VB版	Web版 (v2.0)	実装状況
基本データ処理			
CSVデータ読み込み	✓	✓	✓ 完全実装
統計計算 (min, max, avg, stdDev)	✓	✓	✓ 完全実装
グラフ表示	✓	✓	✓ Chart.js使用
フィルタリング			
移動平均フィルタ (3点)	✓	✓	✓ 完全実装
FFT処理	✓	✓	✓ 完全実装
FFTローパスフィルタ	✓	✓	✓ 完全実装
FFTハイパスフィルタ	✓	✓	✓ 完全実装
FFTバンドパスフィルタ	✓	✓	✓ 完全実装
解析機能			
ピーク検出 (極大・極小)	✓	✓	✓ 完全実装
異常値検出 (統計的)	✓	✓	✓ 完全実装
スペクトル解析 (FFT)	✓	✓	✓ 完全実装
カント補正	✓	✓	✓ 完全実装
スラック補正	✓	✓	✓ 完全実装
MTT評価 (BC/CD値)	✓	✓	✓ 完全実装
エクスポート			
CSV出力	✓	✓	✓ 完全実装
Excel出力	✓	✓	✓ 完全実装

### 8.2 部分実装機能

機能カテゴリ	VB版	Web版 (v2.0)	実装状況	備考
データ形式対応				

LABOCS形式	✓	X	⚠ 未対応	VB版の主要形式の1つ
DCP形式	✓	X	⚠ 未対応	検測車データの標準形式
RSQ形式	✓	X	⚠ 未対応	Rail Sequence形式
DDB形式	✓	X	⚠ 未対応	Data Description Block
MDT形式	✓	X	⚠ 未対応	Management Data Transfer
T3形式	✓	X	⚠ 未対応	新軌道T3形式
データベース連携				
Oracle Database接続	✓	X	⚠ 未実装	VB版はADO経由で接続
DAO Database	✓	X	⚠ 未実装	KCDW用のローカルDB
高度な波形処理				
Bs05系波形処理 (17種類)	✓	X	⚠ 未実装	Bs0510~Bs0640MKcdw
HSJ系波形補正	✓	X	⚠ 未実装	区間復元アルゴリズム
Y1Y2系2次元解析	✓	X	⚠ 未実装	縦横同時処理
複雑なピーク検出				
HPP区間連続ピーク検出	✓	△	⚠ 簡易版のみ	基本的なピーク検出は実装済み
HPP要注意箇所抽出	✓	X	⚠ 未実装	高度な評価基準

## 8.3 未実装機能リスト ❌

### 8.3.1 データ形式・変換機能

優先度: 高

- ☐ LABOCS形式対応 (DCP2S, Ora2Lab2プロジェクト)
  - VB版の主要データ形式
  - ヘッダー + データブロック構造
  - テキストベース
  - 実装箇所: backend/src/parsers/labocs.js (新規)
- ☐ DCP形式対応 (DCP2S, DCPZWプロジェクト)
  - 軌道検測車の生データ形式
  - Z (高さ)、W (横)、S (スラック)、K (キロ程) チャンネル
  - バイナリまたはテキスト
  - 実装箇所: backend/src/parsers/dcp.js (新規)

優先度: 中

- ☐ RSQ形式対応
  - Rail Sequence (レールシーケンス)
  - DDB (メタデータ) + データ配列

- 実装箇所: backend/src/parsers/rsq.js (新規)
- ☐ DDB形式対応
  - Data Description Block
  - 軌道データのメタデータ構造体
  - 実装箇所: backend/src/parsers/ddb.js (新規)

優先度: 低

- ☐ MDT形式対応
  - Management Data Transfer
  - 管理データ転送用
  - 実装箇所: backend/src/parsers/mdt.js (新規)
- ☐ T3形式対応
  - 新軌道T3形式
  - バイナリ形式
  - 実装箇所: backend/src/parsers/t3.js (新規)

### 8.3.2 データベース連携機能

優先度: 高

- ☐ Oracle Database連携 (Ora2Lab2, Ora2LaS2プロジェクト)
  - Oracle軌道データベースからのデータ抽出
  - SQL実行・レコード取得
  - LABOCS形式への変換
  - 実装技術: node-oracledb
  - 実装箇所: backend/src/database/oracle.js (新規)

優先度: 中

- ☐ ローカルデータベース (KCDW用)
  - 履歴データの永続化
  - 検索・フィルタリング機能
  - 実装技術: SQLite または PostgreSQL
  - 実装箇所: backend/src/database/local.js (新規)

### 8.3.3 高度な波形処理アルゴリズム

優先度: 高

- ☐ Bs05系波形処理アルゴリズム (CmdLib.bas)
  - Bs0510MKcdw: 基本軌道変位計算
  - Bs0520MKcdw: 簡易波形処理
  - Bs0530MKcdw: 値以上の内方処理
  - Bs0540~Bs0640MKcdw: 各種波形処理 (全17種類)
  - 実装箇所: backend/src/algorithms/waveformProcessing.js (新規)

優先度: 中

- ☐ HSJ系波形補正アルゴリズム (CmdLib.bas)

- HSJ5\_GETBC: 区間のBCデータ取得
  - HSJ5\_SAIHI: 再帰的な補正処理
  - 軌道の連続性を考慮した波形復元
  - 異常値の自動除外
  - 実装箇所: `backend/src/algorithms/hsjCorrection.js` (新規)
- ☐ Y1Y2系2次元解析 (CmdLib.bas)
  - Y1 (縦方向)、Y2 (横方向) の同時処理
  - 曲線区間の補正
  - 実装箇所: `backend/src/algorithms/y1y2Analysis.js` (新規)

#### 8.3.4 高度なピーク検出機能

優先度: 中

- ☐ HPP区間連続ピーク検出 (CmdLib.bas)
  - 区間連続的なピーク検出
  - 前後関係を考慮したピーク抽出
  - 実装箇所: `backend/src/algorithms/peakDetection.js` (拡張)
- ☐ HPP要注意箇所抽出 (CmdLib.bas)
  - 高度な評価基準に基づく要注意箇所の自動抽出
  - 閾値判定 (ABSYN="YES"で絶対値判定)
  - 実装箇所: `backend/src/algorithms/peakDetection.js` (拡張)

#### 8.3.5 ユーザーインターフェース機能

優先度: 高

- ☐ 複数ファイル一括処理
  - バッチ変換処理
  - 進捗表示
  - 実装箇所: `frontend/src/components/BatchProcessing.tsx` (新規)

優先度: 中

- ☐ キロ程変換ツール (KANA3.frm - Command14)
  - 距離単位の変換
  - キロ程とメートルの相互変換
  - 実装箇所: `frontend/src/components/KilometerConverter.tsx` (新規)
- ☐ 詳細パラメータ設定画面 (KANA3B~KANA3F)
  - 計算パラメータの詳細設定
  - 波形表示設定
  - 補正処理設定
  - 実装箇所: `frontend/src/components/AdvancedSettings.tsx` (新規)

優先度: 低

- ☐ MTT検査データフロッピーコピー (KANA3G.frm)
  - レガシー機能のため優先度低

- 現代的な代替: USBドライブへのエクスポート

### 8.3.6 レポート・出力機能

優先度: 中

- ☐ PDFレポート生成
  - グラフと統計を含む総合レポート
  - 実装技術: pdfkit または puppeteer
  - 実装箇所: backend/src/reports/pdf.js (新規)
- ☐ 地点単位表/区間系表出力 (CmdLib.bas)
  - TD地点単位表ファイル生成
  - TD区間系表ファイル生成
  - 実装箇所: backend/src/reports/tables.js (新規)

### 8.3.7 その他の機能

優先度: 低

- ☐ Hgs専用ライブラリ対応
  - HgsLACCommon, HgsLZCommon
  - HgsLAClient, HgsLZClient
  - 独自通信プロトコル
  - 代替案: REST API化
- ☐ フロッピードライブ対応
  - レガシー機能のため不要
  - 現代的な代替: クラウドストレージ

## 8.4 実装優先度と推奨ロードマップ

### フェーズ1 (v2.1) - 3ヶ月

目標: 主要データ形式対応

- LABOCS形式読み込み
- DCP形式読み込み
- 複数ファイル一括処理
- バッチ変換UI

### フェーズ2 (v2.5) - 6ヶ月

目標: データベース連携

- Oracle Database接続
- SQLiteローカルDB実装
- 履歴管理機能
- データ検索・フィルタリング

### フェーズ3 (v3.0) - 12ヶ月

目標: 高度な解析機能

- Bs05系波形処理アルゴリズム (全17種類)

- HSJ系波形補正
- Y1Y2系2次元解析
- HPP区間連続ピーク検出

#### フェーズ4 (v3.5) - 18ヶ月

目標: エンタープライズ機能

- PDFレポート生成
- 詳細パラメータ設定UI
- 地点単位表/区間系表出力
- ユーザー認証・権限管理

### 8.5 技術的課題と対応方針

#### 課題1: バイナリデータ形式の処理

VB版: バイナリファイルを直接読み込み Web版の課題: Node.jsでのバイナリ処理 対応方針: Buffer API、ArrayBuffer使用

#### 課題2: Oracle Database接続

VB版: ADO 2.5経由で接続 Web版の課題: Node.jsからのOracle接続 対応方針: node-oracledb (Oracle公式ライブラリ)

#### 課題3: 複雑な波形処理アルゴリズム

VB版: 33,843行のCmdLib.basに実装 Web版の課題: VBコードのJavaScript移植 対応方針:





- アルゴリズムの理解と文書化
- テストケース作成
- 段階的な移植と検証

#### 課題4: Hgs専用ライブラリ






VB版: 独自通信プロトコルを使用 Web版の課題: ライブラリの仕様不明 対応方針: REST API化、または仕様調査

### 8.6 VB版との互換性

#### データ互換性

-  CSV形式: 完全互換
-  LABOCS形式: 未対応 (v2.1で対応予定)
-  DCP形式: 未対応 (v2.1で対応予定)
-  RSQ/DDB形式: 未対応 (v2.5で対応予定)

#### 計算結果の互換性

-  基本統計: 完全互換
-  FFTフィルタ: 完全互換
-  ピーク検出: 基本機能は互換
-  MTT計算: 基本的に互換 (詳細パラメータは未対応)
-  Bs05系処理: 未対応 (v3.0で対応予定)

---

ドキュメントバージョン: 2.0 最終更新日: 2025年10月 対象ソースコード: rail-track-app v2.0.0 (高度な解析機能追加版) VB版参照: VB6.0 ソースコード説明書 (2025-10-15版)