JAEA-Technology
2025-007
DOI:10.11484/jaea-technology-2025-007

JAEA-Technology

広域放射線サーベイのための リアルタイムマッピングソフトウェアの開発

Development of Real-time Mapping Software for Wide-area Radiation Survey

髙橋 時音　小泉 光生　吉見 優希　持丸 貴則

Tohn TAKAHASHI, Mitsuo KOIZUMI, Yuki YOSHIMI and Takanori MOCHIMARU

原子力人材育成・核不拡散・核セキュリティ総合支援センター

Integrated Support Center for Nuclear Nonproliferation, Security and Human Resource Development

November 2025

Japan Atomic Energy Agency　日本原子力研究開発機構

国立研究開発法人日本原子力研究開発機構　研究開発推進部　科学技術情報課
〒 319-1112　茨城県那珂郡東海村大字村松 4 番地 49
E-mail: ird-support@jaea.go.jp

JAEA-Technology 2025-007

広域放射線サーベイのためのリアルタイムマッピングソフトウェアの開発

日本原子力研究開発機構 原子力人材育成・核不拡散・核セキュリティ総合支援センター
髙橋　時音，小泉　光生，吉見　優希*1，持丸　貴則*2

（2025 年 7 月 16 日受理）

イベント会場等にテロ行為目的で核・放射性物質が持ち込まれることを防ぐため、放射線検出器により、出入りする人や車両を個別に検査する手法が一般的に用いられている。しかし、こうした検査をすり抜ける可能性があるため、補完的にゲート内の広範囲にわたる放射線サーベイを行い、核・放射性物質が持ち込まれていないことを確認する必要がある。

広いエリアを効率的に放射線サーベイする手法として、GPS を搭載したガンマ線検出器を用い、移動しながら測定した位置情報と線量を記録する「放射線マッピング」が有効である。ネットワークを利用すると、複数台の検出器からの GPS と測定データを指揮所で集計し、測定の進行状況や、測定した放射線量マップをリアルタイムで確認することが可能となる。このような仕組みを導入することにより、測定の重複や抜け落ちを防ぐとともに、不審な放射線源を迅速に検出できるようにできる。さらに、ガンマ線検出器にスペクトロメーターを導入すると、放射性同位体の同定に基づく適切な対処が可能となる。

このような広域放射線サーベイを行うため、リアルタイムマッピングソフトウェアを開発した。開発したソフトウェアは、GPS 付ガンマ線スペクトルメーターから送信される測定データを受け、リアルタイムで逐次処理し、あらかじめダウンロードしておいた地図データ上に描画する。また、線量の上昇した領域でスペクトルを積算することにより放射性同位元素の同定が行え、それに基づいて対処法が決定できるようになった。さらに、本ソフトウェアは、情報セキュリティを向上させるため、ローカルネットワークのみでも利用できるようになっている。

本報告書では、開発したソフトウェアの概要を紹介するとともに、エッセンスを簡易化したコードを付録で提供する。提供したコードは、オープンかつフリーの OS、ライブラリ、環境で開発しており、誰でも導入して使用可能である。

本報告書は、文部科学省の核セキュリティ強化等推進事業費補助金の下で進められた研究成果に関するものである。

原子力科学研究所：〒319-1195　茨城県那珂郡東海村大字白方 2 番地 4
＊1 BREXA Technology
＊2 NAIS

JAEA-Technology 2025-007

Development of Real-time Mapping Software for Wide-area Radiation Survey

Tohn TAKAHASHI, Mitsuo KOIZUMI, Yuki YOSHIMI[*1] and Takanori MOCHIMARU[*2]

Integrated Support Center for Nuclear Nonproliferation, Security and Human Resource Development

Japan Atomic Energy Agency

Tokai-mura, Naka-gun, Ibaraki-ken

(Received July 16, 2025)

To prevent the smuggling of nuclear and radioactive materials into event venues for the purpose of terrorism, it is common practice to individually inspect people and vehicles entering and exiting using radiation detectors. However, since there remains a risk of such inspections being bypassed, it is necessary to complement them with a wide-area radiation survey to ensure that no nuclear or radioactive materials have been brought in.

Radiation mapping is an effective method for efficiently surveying large areas. In this method, a gamma-ray detector equipped with GPS is used to record location data and radiation dose rates while moving. By utilizing network connectivity, measurement data from multiple detectors can be aggregated at a central command post, allowing real-time monitoring of survey progress. This system helps to prevent both redundant and missing measurements and enables the prompt detection of suspicious radiation sources. Furthermore, by incorporating spectrometers into the gamma-ray detectors, it becomes possible to identify radioactive isotopes, thereby enabling appropriate responses.

To enable such wide-area radiation surveys, we developed real-time mapping software. The developed software receives measurement data transmitted from GPS-equipped gamma-ray spectrometers, processes it sequentially in real time, and plots it onto pre-downloaded map data. Additionally, by integrating the spectral data collected from regions showing abnormal radiation levels can be displayed immediately. To enhance information security, the software is designed to function within local networks without requiring internet connectivity.

In this report, we introduce an overview of the developed software and provide a simplified version of the source code as an appendix. The provided code is developed using open and free operating systems, libraries, and environments, making it freely available and usable by anyone.

Keywords: Radiation Mapping, Global Positioning System, Real-time Monitoring, Visualization Software Development

---

This development was implemented under the subsidiary for nuclear security promotion of MEXT.

*1 BREXA Technology

*2 NAIS Co. Inc.,

目 次

Contents

図リスト

1. はじめに

放射線や原子力の利用が進む中、発電所などの施設から核・放射性物質が不法な手段で持ち出され、それらが核テロに用いられることが懸念されている。オリンピックなどの大規模イベントは、その注目度の高さからテロの標的とされる可能性が指摘されており、会場での核テロの発生を未然に防ぐための対策の強化が求められている。核テロの未然防止策として、イベント会場周辺を区画し、出入りゲートに放射線検出器を設置して通過する人や車両を個別にサーベイする方法がとられている。しかし、定められたゲート以外からの侵入やあらかじめ持ち込まれた場合などを考慮すると、核・放射性物質がゲート内に持ち込まれていないことを確認する補完的な手段が必要である。このような状況に対応するためには、ゲート内にバックグラウンドレベルを超える放射線量となるエリアがないことを継続的にサーベイする技術が必要である。

広い領域を効率よくサーベイするためには、可搬型放射線検出器に測位センサーを取り付け、放射線量と位置情報を紐づけて記録する方法が効果的である。測定結果を地図に重ねて表示することで、放射線量の高いエリアを容易に特定することができる。また、ネットワークデバイスを通じて測定データを転送可能にすれば、複数台の検出器が連携する際、指揮所でサーベイの状況を一元的に確認ができるようになる。これにより、不審な放射線源の検知に対し、迅速な判断・対処が可能となることに加えて、測定の重複や見逃しを低減することができる。ガンマ線検出器については、スペクトロメーターを導入すれば、検知した放射性物質の同定が可能となり、その後の対応の判断材料を迅速に得ることが可能となる。

日本原子力研究開発機構 原子力人材育成・核不拡散・核セキュリティ総合支援センターでは、核セキュリティ強化等推進事業費補助金の下、以下の機能を備えた広域サーベイシステムの開発を進めている[1,2]。

(1) 屋外のサーベイに対応するための GPS を組み込んだ放射線検出器の開発
(2) ネットワークデバイスの性能試験
(3) 測定データを確認するためのリアルタイムマッピングソフトの開発
(4) 屋内のサーベイに対応するための SLAM を組み込んだ放射線検出器の開発
(5) 放射性核種の同定能力向上を目指した可搬型ガンマ線検出器の高度化

GPS を利用した屋外の放射線マッピングツールとして、測定した緯度・経度・放射線量を地図上に表示するソフトウェアがすでに存在している。例えば、日本原子力研究開発機構が「放射性物質モニタリングデータの情報公開サイト」[3]で公開しているツールは、測定したデータを Google Earth などで読み込むことができる形式に変換するものである。しかし、データの集計、一括変換などの操作が必要であるため、リアルタイムで逐次処理しながら地図上に表示する用途には適さない。また、様々な放射線測定機器メーカーから供給される核セキュリティ向けのソフトウェアは、特定の検出器と紐づけられているため、汎用的な利用が難しい。さらに、情報セキュリティの観点から、公共ネットワークを利用せずローカルネットワークのみで通信を行う環境が求められる場合も想定される。

そこで、我々は、公共のネットワークから切り離されたローカルネットワーク環境で、GPS付スペクトロメーターから送られてくるデータをリアルタイムに線量マッピングするソフトウェアを独自に開発した。本稿では、開発したリアルタイムマッピングソフトウェアの概要について報告する。

本ソフトウェアは、核セキュリティ分野のみならず、放射線マッピングや、災害対応、環境モニタリングなど幅広い分野での活用が期待される。そこで、本報告の巻末には、機能を簡略化したサンプルプログラムを付録として掲載した。このプログラムが技術開発の参考となることを期待する。

2. リアルタイムマッピングソフトウェアの開発

2.1 リアルタイムマッピングソフトウェアの概要

図 1 は、開発したリアルタイムマッピングソフトウェアの概要を示している。本ソフトウェアは、大きく分けて Monitoring system、Map browser、Spectrum viewer の 3 つで構成される。

Monitoring system は通信機能を備えた GPS 付スペクトロメーターから、設定した任意の時間間隔で周期的に送られてくるデータを取得・記録する機能を有する。検出器から送信される測定データは、測定時間、緯度・経度の位置情報、ガンマ線スペクトルが含まれている。データ転送の頻度は、毎秒 1 回程度を想定している。この設定は、毎秒 1 m で歩行した場合、1〜2 m 程度の範囲内で放射線量が高くなる地点を特定できるようにするためである。

Map browser は随時記録されるデータの緯度・経度に対応する位置に、放射線量（計数もしくは線量データ）に応じた色のマーカーをプロットする機能を提供する。ソフトウェア起動時に保存されているすべての計数データを読み込み、さらに 5 秒ごとに追加されるデータを読み込み再描画する。使用する地図はあらかじめダウンロードしたオフライン地図を利用する形式とした。

放射線量が多くなる地点が見つかった場合、Spectrum viewer を利用して、特定の時間帯を指定し、その期間の積算で得られるスペクトルを表示することが可能である。この機能により、放射線量上昇の原因となる放射性同位体を迅速に同定することができる。

開発したソフトウェアの内、Network interface は C 言語を用いており、その他のプログラムは JavaScript を用いて記述されている。各プログラムをコマンドで起動すると、指定のポートにマップとスペクトルを確認するための Web サーバーが立ち上がり、Web ブラウザを用いてアクセスして表示することが可能になる。各プログラムのソースコード、インストール方法、起動方法などは、付録 A にて詳しく説明している。

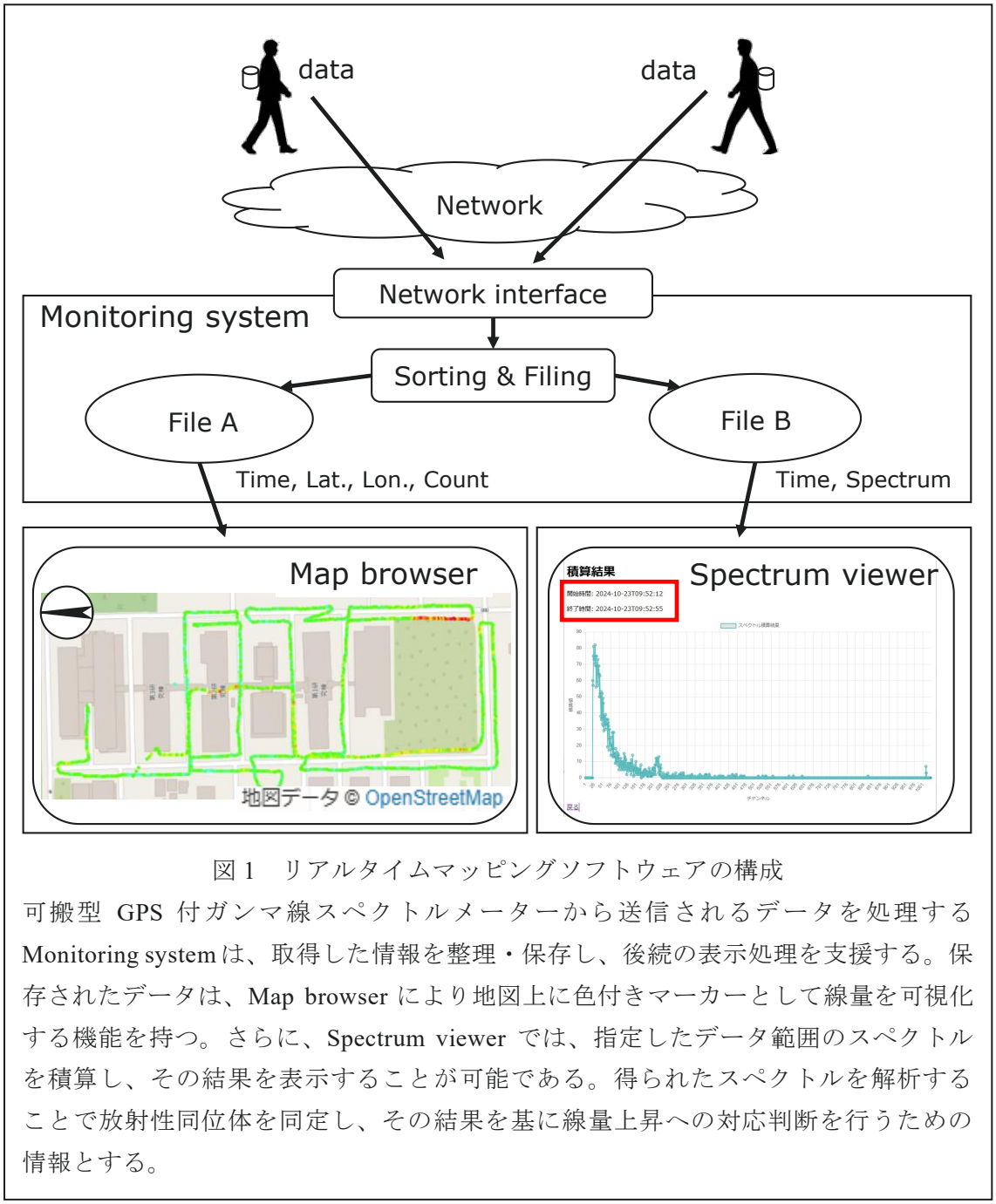

図 1　リアルタイムマッピングソフトウェアの構成

可搬型 GPS 付ガンマ線スペクトルメーターから送信されるデータを処理する Monitoring system は、取得した情報を整理・保存し、後続の表示処理を支援する。保存されたデータは、Map browser により地図上に色付きマーカーとして線量を可視化する機能を持つ。さらに、Spectrum viewer では、指定したデータ範囲のスペクトルを積算し、その結果を表示することが可能である。得られたスペクトルを解析することで放射性同位体を同定し、その結果を基に線量上昇への対応判断を行うための情報とする。

2.2　各プログラムの概要

1) Monitoring system

Monitoring system は、検出器からのデータを受け取る Network interface と、データを整理してファイルに振り分ける Sorting & Filing の 2 つのモジュールで構成される。

Network interface は、ネットワークを介して受信したデータの取り込みを担当する。受信データは、送信元の検出器の種類によって形式が異なるため、そのままでは後の可視化処理のプ

ロセスが複雑になる。この問題を解決するため、Network interface は、検出器の型を判定し、受け取ったデータから時刻、緯度、経度、スペクトルを抽出するとともに、放射線量としてスペクトルの全チャンネルの積分値（Count/s）を計算、形式を揃えてデータを Sorting & Filing に送信する。

Sorting & Filing は、Network interface から送られたデータを 2 つに分け、それぞれを別ファイルに保存する機能を有する。地図上にマーカーを表示するためには緯度、経度、放射線量が必要であり、これらのデータは測定時刻とともに File A に追記・保存される。一方、ガンマ線スペクトルデータは、同様に測定時刻を付加して File B に保存される。データを別のファイルに分けることで、地図の表示とスペクトル表示を同時に行う場合に、同じファイルへの同時アクセスを回避できる。この分離機能を組み込むことにより、地図表示時には不要なスペクトルデータを読み込む必要がなくなるため、メモリ使用量が減少し、パソコンへの負荷を軽減することができた。

2) Map browser

Map browser は、保存された緯度、経度、放射線計数のデータ（File A）を読み込み、地図上にマーカーをプロットして表示するためのプログラムである。本プログラムは JavaScript で記述され、ブラウザ上で動作するよう設計されている。地図表示とマーカー描画には、JavaScript のオープンソース地図ライブラリである Leaflet[4]を使用した。Leaflet は、ブラウザ上でインタラクティブに操作可能な地図表示するためのライブラリであり、マウスドラッグによるスクロールやマウスホイールによる拡大・縮小などの機能を備えている。

地図の表示は、マップタイルと呼ばれる画像ファイルを繋ぎ合わせることで実現される。スクロールや拡大・縮小などで地図表示範囲を切り替えると、その範囲の表示に必要なマップタイルが読み込まれる仕組みである。マップタイルは様々なスタイルのものが利用可能である、本システムでは、OpenStreetMap[5]が提供しているオープンデータを使用している。

サーベイを行う現場においては、通信の秘匿性を高めるため、インターネットに接続できない端末で使用することも想定される。この場合は、必要なマップタイルをあらかじめローカルに保存し、それを利用する方法をとることができる。OpenStreetMap のマップタイルをローカルに保存する手順については、付録 B に記載している。

図 2 に Map browser を起動して測定データを表示した例を示す。この例では、直線状の道に沿って移動しながら行い、1 秒間隔でデータを記録したもので、マーカーの色スケールは少しの空間線量の上昇でもわかるよう調整している。ちなみに、緑のマーカーで示されている地点は、周辺線量当量で約 0.08 μSv/h の線量に相当し、赤のマーカーで示されている地点は、約 0.16 μSv/h に相当する。これらの線量はバックグラウンドレベルを大きく超えていない。

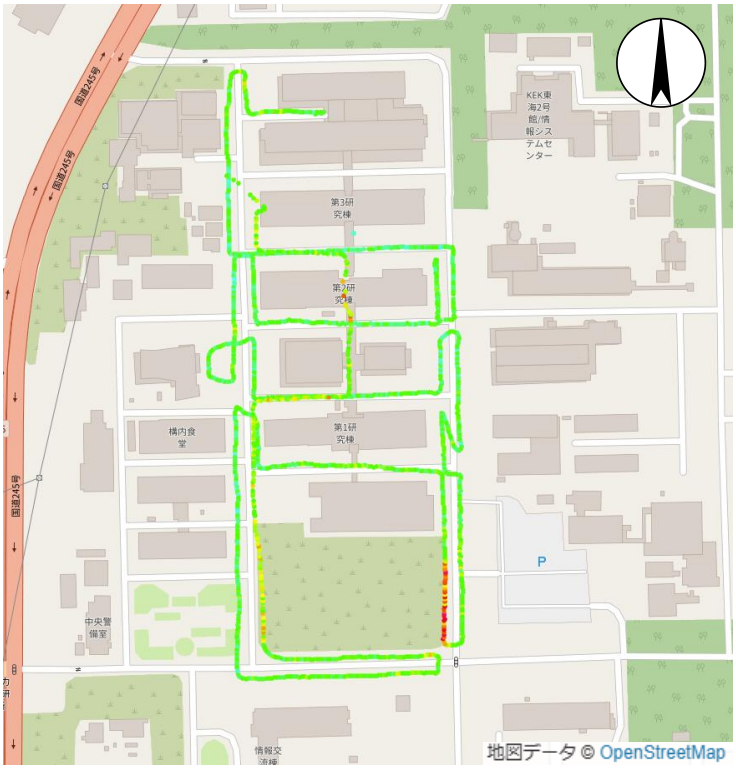

図 2　Map browser の使用例

1 秒ごとに測定されたデータを色付きマーカーとして地図上にリアルタイムにプロットした。緑色のマーカーの地点では放射線量はバックグラウンドレベルからの変動はなかった。赤色のマーカーの地点では、バックグラウンドレベルではあるものの若干の放射線量の上昇がみられた。

3) Spectrum viewer

Map browser を用いると線量が上昇した場所がわかる。例えば、図 2 では、南東の端に放射線計数が高いエリアが観測されている。このような場合、そのエリアで測定されたガンマ線スペクトルを積算し、得られたスペクトルから核種同定を行うことで、適切な対応方法を決定するための判断材料とすることができる。

Spectrum viewer は、指定した時間帯で測定されたガンマ線スペクトルを積算し、その結果を表示するためのプログラムである。図 3(a)は、Spectrum viewer の起動画面を示しており、測定の開始時刻と終了時刻を指定する入力欄が設けられている。この欄に積算を行いたい時間範囲を指定し、「積算する」のボタンをクリックすることで、指定した時刻範囲内のスペクトルが File B から読み込まれ、積算されたスペクトルが図 3(b)のように表示される。

図 3(b)は、積算後のスペクトル表示例を示している。積算スペクトルに表れたピークのエネルギーを確認することで、観測された放射線源の核種を同定することが可能である。この例の場合、ピークのエネルギーが 662 keV であるため、東日本大震災に伴う福島第一原子力発電所

事故時に放出された Cs-137 が集まったホットスポットの近くを通過したためであると判断できる。

(a)

(b)

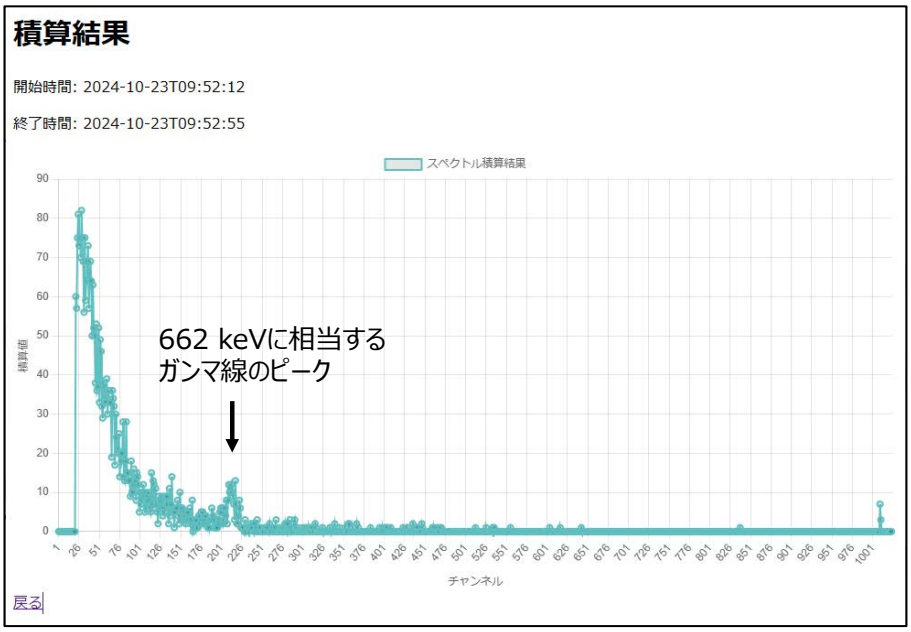

図 3　Spectrum viewer の表示画面

(a)の画面で開始時間と終了時間を設定した後、「積算する」ボタンをクリックすると、指定した時間範囲内に含まれるスペクトルデータが積算され、その結果が(b)の画面に表示される。放射線量の上昇の原因となる放射性同位体の同定は、迅速かつ適切な対応を行うための判断材料となる。

3．まとめ

大規模イベント等での核テロを未然に防ぐための技術として、GPS 付可搬型ガンマ線スペクトロメーターからローカルネットワーク経由で送られてくるデータを処理し、結果をリアルタイムで地図上に表示するためのソフトウェアの開発を行った。

開発したソフトウェアを利用することにより、検出器が移動した場所の放射線量をリアルタイムで地図上に可視化し、広範囲の放射線状況を的確に把握することが可能となる。また、各検出器の移動位置を監視しつつ、放射線量が増加したエリアに対して迅速に対応することもできる。さらに、取得したガンマ線スペクトルを任意の範囲で積算して表示する機能を備えており、放射線量の増加が確認されたエリアで核種を同定する際に有用な情報を提供することができる。

本ソフトウェアは核セキュリティ分野での利用にとどまらず、廃止措置や除染における放射線マッピングや、災害対応や環境モニタリングなど、幅広い分野での活用が期待される。開発に際しては、オープンソースで無料のものを採用しており、コストを抑えつつ高い汎用性を実現している。付録 A に記載したソースコードは、簡素な構成とし、導入しやすい設計としているため、利用者が各自の用途に合わせ改良し、効果的に活用されることを期待している。

謝　辞

本技術開発は、文部科学省の核セキュリティ強化等推進事業費補助金の下で進められた。また、本技術開発は、日本原子力研究開発機構と国際原子力機関（IAEA）との共同研究（Coordinated Research Program (CRP), J02015 (Facilitation of Safe and Secure Trade Using Nuclear Detection Technology - Detection of RN and Other Contraband)）に対しても寄与するものである。

参考文献

[1] 高橋時音ほか, 「大規模公共イベント等における核・放射性物質モニタリング技術開発」, 日本核物質管理学会第 44 回年次大会予稿集, 2023, 東海.

[2] M.Koizumi, et al., “Current status of a JAEA development program on nuclear and radioactive materials detection techniques in major public events”, Proc. of the INMM and ESARDA joint Annual Meeting (2023, Viena, Austria).

[3] 日本原子力研究開発機構「放射性物質モニタリングデータの情報公開サイト」, マッピングツール, https://emdb.jaea.go.jp/emdb_old/mappingtool.html (accessed 2025/10/01).

[4] Leaflet, https://leafletjs.com/index.html (accessed 2025/10/01).

[5] OpenStreetMap, https://www.openstreetmap.org/ (accessed 2025/10/01).

付録 A

リアルタイムマッピングソフトウェアのインストール及び実行手順

1） 構成

リアルタイムマッピングソフトウェアは、次の 4 つのソースファイルで構成される。

1. server_test.cpp
2. server.js
3. index.html
4. map.js
5. spectrum_sum.js

各ソースファイルの関係は図 A-1 の通りである。

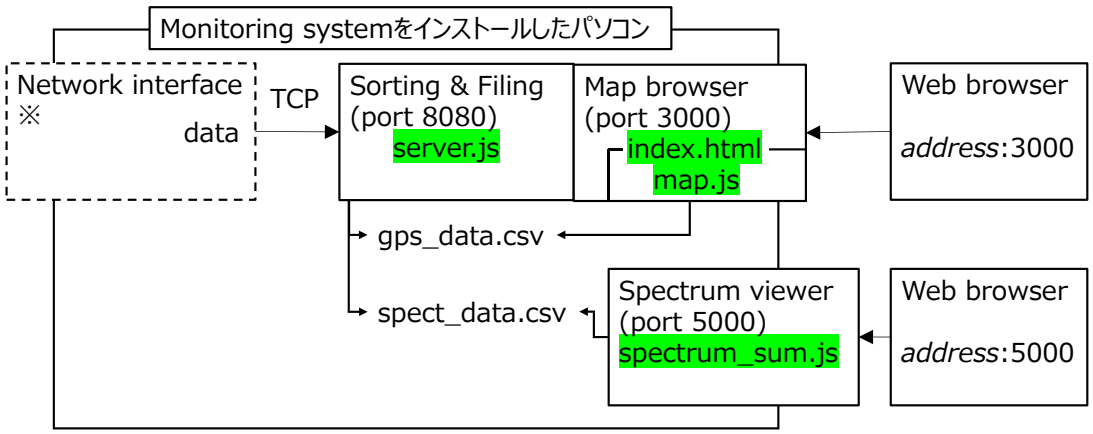

図 A-1 リアルタイムマッピングソフトウェアを校正するソースファイルの関係図
緑でハイライトしたものは、各プログラムのソースファイル名を示している。
※測定データを受け取る Network interface については、環境に依存するものであるため、付録 A ではサンプルデータを作成する server_test.cpp を記載している。

server_test.cpp は、ソフトウェアの動作をテストするために用いたコードで、サンプルデータを作成してポート 8080 にデータを送信する。（本来は、ネットワークデバイスがマウントされているポートからデータを読み出し、統一した形式に整える機能を持つがここでは省略している。）データは、表 A-1 に示した 6 つのものを含んでおり、JSON 形式に準じた文字列で送信される。

{latitude:*緯度*,longitude:*経度*,timestamp:*時間*,value:*計数*,gpsNumber:*ID*,spectrumData:*スペクトル*}

表 A-1　ポート 8080 に送信するデータに含まれる各パラメーターの形式

データ	形式	例
緯度	小数点以下 6 桁の数値	36.461537
経度	小数点以下 6 桁の数値	140.600006
時間	日付 T 時刻 yyyy-mm-ddTHH:MM:SS	2024-08-09T14:19:20
計数	整数	47
ID	整数	2
スペクトル	[カンマで区切られた 1024 個の整数]	[0,0,2,6,3,4,0,1,…,0,0,0]

server.js は、コマンドによって起動し、ポート 8080 に送られてくるデータを分類し、2 つのファイルに分けて保存する Sorting & Filing の機能を担う。起動時に、ポート 3000 で Map browser を起動する。

Map browser のソースファイルは、index.html 及び map.js である。index.html には、地図のサイズや表示スタイル等、ブラウザ上でのレイアウトに関する設定が記述されている。map.js は、地図や測定データの表示に関するコードであり、index.html から読み込まれる。

spectrum_sum.js は、スペクトルデータを積算して確認する Spectrum viewer のソースファイルであり、画面のレイアウトを設定するための html コードが含まれている。

2） 実行環境

Node.js 10.19.0 以上

（サンプルデータを生成するプログラムを利用する場合、C++をコンパイル、実行できる環境が必要）

3） インストール手順（環境を作る手順・コマンドリスト）

事前に JavaScript の実行環境である Node.js 及びパッケージマネージャーである npm をインストールしておく。Ubuntu では、次のコマンドを実行することでインストールできる。

- apt update
- apt install nodejs
- apt install npm

3-1) プログラムを実行するディレクトリを作成する。

（ここでは、ディレクトリ名を realtime_mapping_tool とする。）

3-2) 作成したディレクトリに移動し、package.json 及び server.js を作成する。それぞれのファイルに次のソースコードをコピーする。

<ファイル名：package.json>

```
{
  "name": "leaflet_csv_simple_plot",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": " takahashi.tohn ",
  "license": "ISC",
  "dependencies": {
    "chart.js": "^2.8.0",
    "cors": "^2.8.5",
    "express": "^4.19.2",
    "ws": "^8.18.0"
  }
}
```

<ファイル名：server.js>

```
const fs = require('fs');
const path = require('path');
const net = require('net');

const http = require('http');
const express = require('express');
const WebSocket = require('ws');
const app = express();
const cors = require('cors');
app.use(cors());
app.use(express.static(path.join(__dirname, 'public')));
const server = http.createServer(app);
const wss = new WebSocket.Server({ server });

// Web ソケット接続
wss.on('connection', function connection(ws) {
   console.log('Client connected');
   ws.on('message', function incoming(message) {
      //console.log('received: %s', message);
   });
});

// GPS データの保存パス
const gpsDataFilePath = path.join(__dirname, './public/gps_data.csv');
// スペクトルデータの保存パス
```

```
const spectDataFilePath = path.join(__dirname, './public/spect_data.csv');

// GPS データを CSV に保存する関数
function saveGpsData(latitude, longitude, timestamp, count, gpsNumber) {
    const data = `${timestamp},${latitude},${longitude},${count}\n`;
    fs.appendFile(gpsDataFilePath, data, (err) => {
        if (err) {
            console.error('Error saving GPS data:', err);
        }
    });
}

// スペクトルデータを CSV に保存する関数
function saveSpectData(timestamp, gpsNumber, spectrumData) {
    const data = `${timestamp},${gpsNumber},${spectrumData.join(',')}\n`;
    fs.appendFile(spectDataFilePath, data, (err) => {
        if (err) {
            console.error('Error saving Spectrum data:', err);
        }
    });
}

// TCP サーバーを作成
const tcpServer = net.createServer((socket) => {
    console.log('TCP client connected');

    socket.on('data', (data) => {
        try {
            const jsonData = JSON.parse(data.toString());

            // GPS データを保存
            saveGpsData(jsonData.latitude, jsonData.longitude, jsonData.timestamp, jsonData.value,
jsonData.gpsNumber);

            // スペクトルデータを保存
            saveSpectData(jsonData.timestamp, jsonData.gpsNumber, jsonData.spectrumData);

        } catch (err) {
            console.error('Error processing data:', err);
        }
    });

    socket.on('end', () => {
        console.log('TCP client disconnected');
    });
});

// TCP サーバーを指定ポートでリスン開始
tcpServer.listen(8080, () => {
    console.log('TCP Server listening on port 8080');
});

server.listen(3000, function() {
```

```
    console.log('Web server listening on port 3000');
});
```

3-3) 「public」というサブディレクトリを作成し、その中に index.html と map.js を作成する。それぞれのファイルに次のソースコードをコピーする。

<ファイル名：index.html>

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>GPS Mapping</title>
    <link rel="icon" href="data:,">
    <!-- Leaflet CSS -->
    <link rel="stylesheet" href="leaflet/leaflet.css" />
    <style>
        /* 地図の表示領域を確保 */
        #map {
            height: 100vh; /* 画面全体の高さを使う */
            width: 100vw;
        }
    </style>
</head>
<body>
    <!-- 地図を表示するための div 要素 -->
    <div id="map"></div>

    <!-- Leaflet JS -->
    <script src="leaflet/leaflet.js"></script>
    <!-- 自作の map.js を読み込む -->
    <script src="map.js"></script>
</body>
</html>
```

<ファイル名：map.js>

```
// 地図を初期化する
var map = L.map('map').setView([36.461537, 140.600006], 15); // 初期表示の中心位置とズームレベルを設定
L.tileLayer('map_tile/{z}/{x}/{y}.png', { maxZoom: 19, minZoom: 14}).addTo(map);

// カラーマッピングのための関数
function calculateColor(value, minValue, maxValue) {
```

```
    const normalized = (value - minValue) / (maxValue - minValue);
    const hue = (1 - normalized) * 240; // 青（240 度）から赤（0 度）へのグラデーション
    return `hsl(${hue}, 100%, 50%)`;
}

// CSV ファイルを読み込む関数
 async function loadCsvData(filePath) {
    try {
        const response = await fetch(filePath);

        // ファイルが存在しない場合は 404 エラーを検出
        if (!response.ok) {
            if (response.status === 404) {
                console.log(`ファイルが存在しません: ${filePath}`);
                return []; // 空の配列を返して処理を続行
            }
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const text = await response.text();

        // データが存在する場合にのみパースする
        if (text.trim().length === 0) {
            console.log(`ファイルが空です: ${filePath}`);
            return []; // 空の配列を返す
        }

        return parseCsv(text);
    } catch (error) {
        console.error(`データの読み込み中にエラーが発生しました: ${error}`);
        return []; // エラーが発生した場合でも空の配列を返す
    }
}

// CSV テキストをパースする関数
function parseCsv(text) {
    const lines = text.trim().split('\n');
    // ファイルが空かどうかを確認
    if (lines.length === 0) {
        console.error("CSV ファイルが空です");
        return [];
    }

    const headers = ["timestamp", "latitude", "longitude", "count"]; // ヘッダーを手動で指定

    return lines.map(line => {
        const values = line.split(',');
        const obj = {};

        headers.forEach((header, index) => {
            obj[header] = values[index].trim();
        });
```

```
        return obj;
    });
}

// 地図上に GPS データをプロットする関数
async function plotGpsData() {
    const gpsData = await loadCsvData('gps_data.csv');

    if (gpsData.length === 0) {
        console.log("データがありません。再試行します。");
        return; // データがない場合、プロット処理をスキップ
    }

    gpsData.forEach((data) => {
        const lat = parseFloat(data.latitude);
        const lon = parseFloat(data.longitude);

        if (isNaN(lat) || isNaN(lon)) {
            console.error(`Invalid LatLng object: (${lat}, ${lon}) at data:`, data);
            return; // 無効なデータをスキップ
        }

        const count = parseInt(data.count);
        const color = calculateColor(count, 0, 100);

        L.circleMarker([lat, lon], {
            color: color,
            radius: 1,
            fillOpacity: 0.7
        }).addTo(map);
    });
}

// ページロード時にデータをプロットし、定期的に更新
document.addEventListener('DOMContentLoaded', () => {
    plotGpsData();
    setInterval(plotGpsData, 5000); // 5 秒ごとにプロットを更新
});
```

ここまでで、メインディレクトリの構成は次の図 A-2 のようになる。

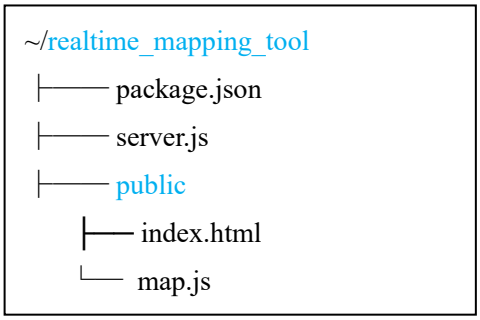

図 A-2　3-3)まで終えた時のメインディレクトリの構成

3-4) public の中にサブディレクトリ「leaflet」を作成し、その中でターミナルを開いて次のコマンドを実行する。

- wget https://unpkg.com/leaflet@1.9.4/dist/leaflet.css
- wget https://unpkg.com/leaflet@1.9.4/dist/leaflet.js

3-5) マップタイルファイルを入手し、public の中のサブディレクトリ「map_tile」に保存する。（マップタイルを入手する例は、付録 B を参照。）

3-6) メインディレクトリ（「realtime_mapping_tool」）でターミナルを開き、次のコマンドを実行する。

- npm install

ここまでで、メインディレクトリ以下の構成は次の図 A-3 のようになる。但し、「node_modules」及び「map_tile」の中は、省略している。

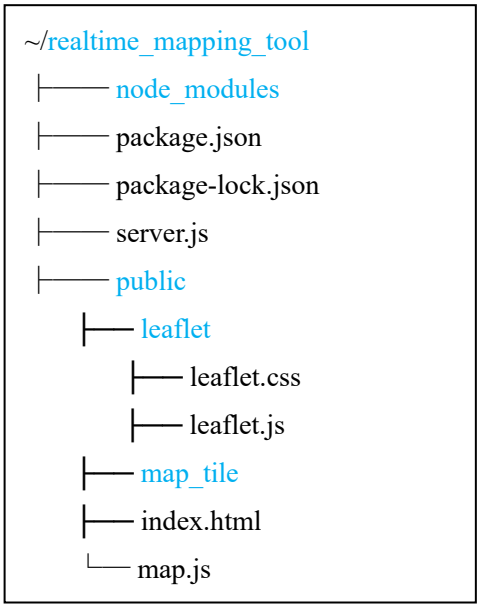

図 A-3　3-6)まで終えた時のメインディレクトリの構成

4） 実行

メインディレクトリでターミナルを開き、次のコマンドを実行する。

- node server.js

実行した結果、下記が出力される。

- TCP Server listening on port 8080
- Web server listening on port 3000

この状態でブラウザを立ち上げて、localhost:3000 にアクセスすると、マップ表示画面を開くことができる。このとき、TCP/IP 通信で、ポート 8080 にデータを文字列で送信すると、サブディレクトリ「public」の中に、「gps_data.csv」と「spect_data.csv」が作成され、送信したデータが保存される。5 秒に 1 度 gps_data.csv が読み込まれ、測定データがマップに反映される。

下記に示す server_test.cpp は、動作テストのために仮のデータを作成し送信するためのコードである。server.js を実行した状態で、これをコンパイル、実行すると、動作確認できる。（コンパイルするには、gcc など C++のコンパイラが必要。）

<ファイル名：server_test.cpp>

```
/*

compile:

g++ server_test.cpp

exec:

./a.out

*/

#include <iostream>
#include <string>
#include <sstream>
#include <iomanip>
#include <vector>
#include <cstring>
#include <arpa/inet.h> // Unix/Linux のソケット通信用ヘッダー
#include <unistd.h>
#include <chrono>
#include <ctime>

// 現在の時間を ISO 8601 形式で取得する関数
std::string getCurrentTimestamp() {
    auto now = std::chrono::system_clock::now();
    std::time_t now_c = std::chrono::system_clock::to_time_t(now);
    std::tm now_tm = *std::localtime(&now_c);

    std::ostringstream oss;
    oss << std::put_time(&now_tm, "%Y-%m-%dT%H:%M:%S");
    return oss.str();
}

// ダミーデータを生成する関数
std::string generateDummyJsonData(double& latitude, double& longitude) {
    std::ostringstream oss;
    std::vector<int> spectrumData;

    // ランダムなスペクトルデータを生成
    for (int i = 0; i < 1024; ++i) {
        spectrumData.push_back(rand() % 256); // 0-255 の範囲の乱数
    }

    // 現在の時間を取得
    std::string timestamp = getCurrentTimestamp();

    // ダミーデータの生成
    oss << "{"
        << "\"latitude\": " << std::fixed << std::setprecision(6) << latitude << ","
```

```
        << "\"longitude\": " << std::fixed << std::setprecision(6) << longitude << ","
        << "\"timestamp\": \"" << timestamp << "\","
        << "\"value\": 45,"
        << "\"gpsNumber\": 1,"
        << "\"spectrumData\": [";

    for (size_t i = 0; i < spectrumData.size(); ++i) {
        oss << spectrumData[i];
        if (i < spectrumData.size() - 1) {
            oss << ",";
        }
    }
    oss << "]"
        << "}\n";

    // 次回の送信に備えて GPS の位置を少しずらす
    latitude += 0.00001; // 緯度を少し増加させる
    longitude += 0.00001; // 経度を少し増加させる

    return oss.str();
}

int main() {
    // サーバーの IP アドレスとポート
    const char* serverIp = "127.0.0.1";
    int serverPort = 8080;

    // 初期 GPS 位置
    double latitude = 36.461537;
    double longitude = 140.600006;

    // ソケットの作成
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        std::cerr << "Error creating socket" << std::endl;
        return 1;
    }

    // サーバーアドレスの設定
    sockaddr_in serverAddr;
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(serverPort);
    inet_pton(AF_INET, serverIp, &serverAddr.sin_addr);

    // サーバーへの接続
    if (connect(sock, (sockaddr*)&serverAddr, sizeof(serverAddr)) < 0) {
        std::cerr << "Error connecting to server" << std::endl;
        close(sock);
        return 1;
    }

    std::cout << "Connected to server" << std::endl;
```

```
    while(1) {
        // ダミーデータの送信
        std::string jsonData = generateDummyJsonData(latitude, longitude);
        send(sock, jsonData.c_str(), jsonData.size(), 0);

        std::cout << "Dummy data sent: " << jsonData << std::endl;
        usleep(1000000); // 1 秒間隔でデータを送信
    }

    // ソケットを閉じる
    close(sock);

    return 0;
}
```

付録 B

オフラインでマップを描画するための画像の作成手順

本文では、地図を描画するためのオープンソースライブラリとして、Leaflet を紹介した。Leaflet をはじめ、多くの地図描画アプリケーションでは、地図の表示範囲に合わせてインターネット上から画像ファイルをロードし、描画する方法をとっている。インターネットへの接続がない状態で地図の描画を利用するには、画像ファイルをローカルに保存しておく必要がある。

ここでは、QGIS を利用した地図描画用の画像ファイルの作成方法を紹介する。QGIS は、無料で利用可能なオープンソースの地理情報システムソフトで、地図の作成やデータの可視化などを行う機能が備えられている。

手順

1) QGIS をダウンロード／インストールする。
   ホームページの Download から QGIS を入手し、手順に従ってインストールする。
   https://www.qgis.org/

2) QGIS を起動し、図 B-1 に示す起動画面で任意の地図を開く。ここでは、インストール時にデフォルトで設定が済んでいる OpenStreetMap を使用する。

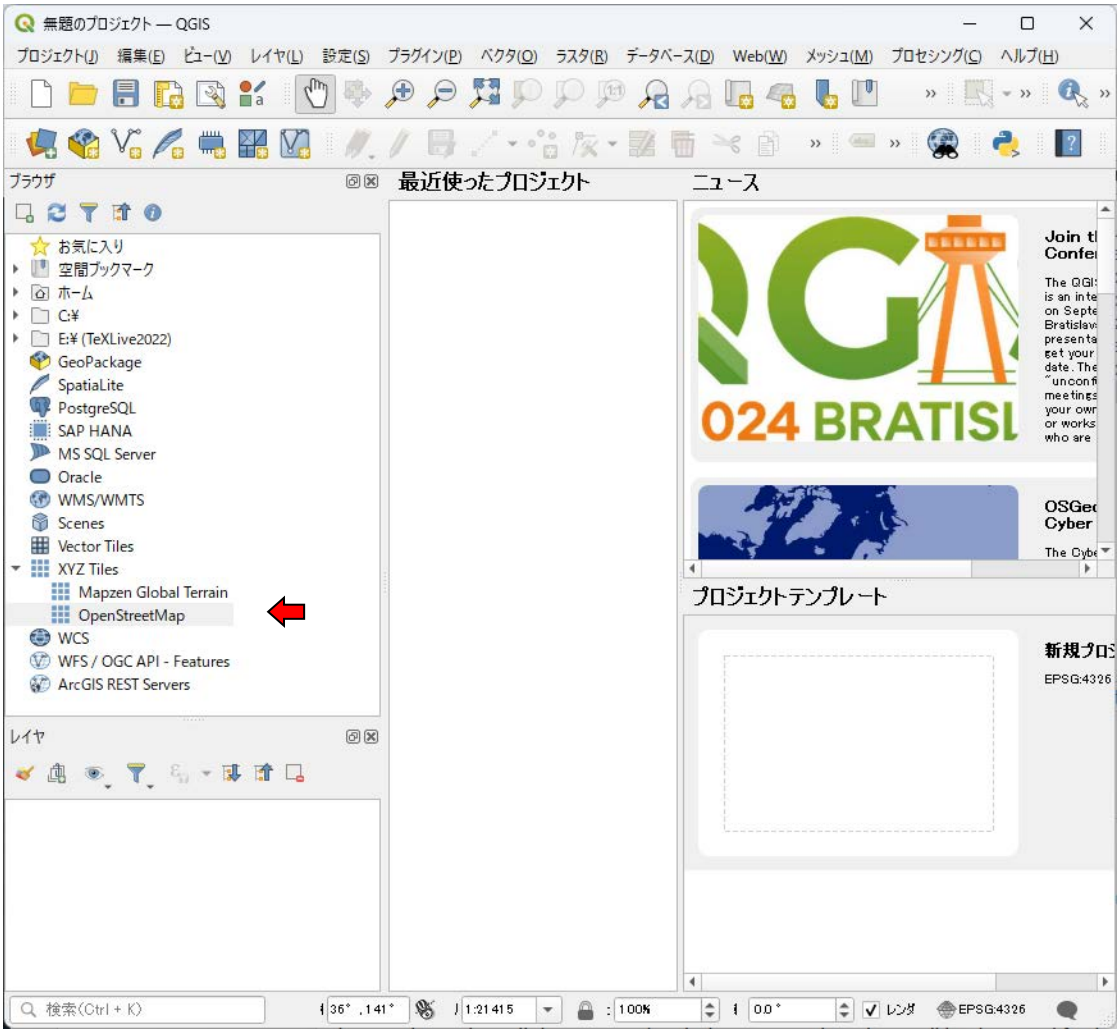

図 B-1　QGIS の起動画面

画面左端「ブラウザ」の「XYZ Tiles」の中に「OpenStreetMap」が設定されている。

3) オフラインでマップを描画したい範囲を画面上に表示する。

4) メニューバーの「プロセシング」→「ツールボックス」を選択し、プロセシングツールボックスを開く。

5) 図 B-2 の図中赤矢印の通り、プロセシングツールボックスの中の「ラスタツール」→「Generate XYZ tiles (Directory)」を選択する。

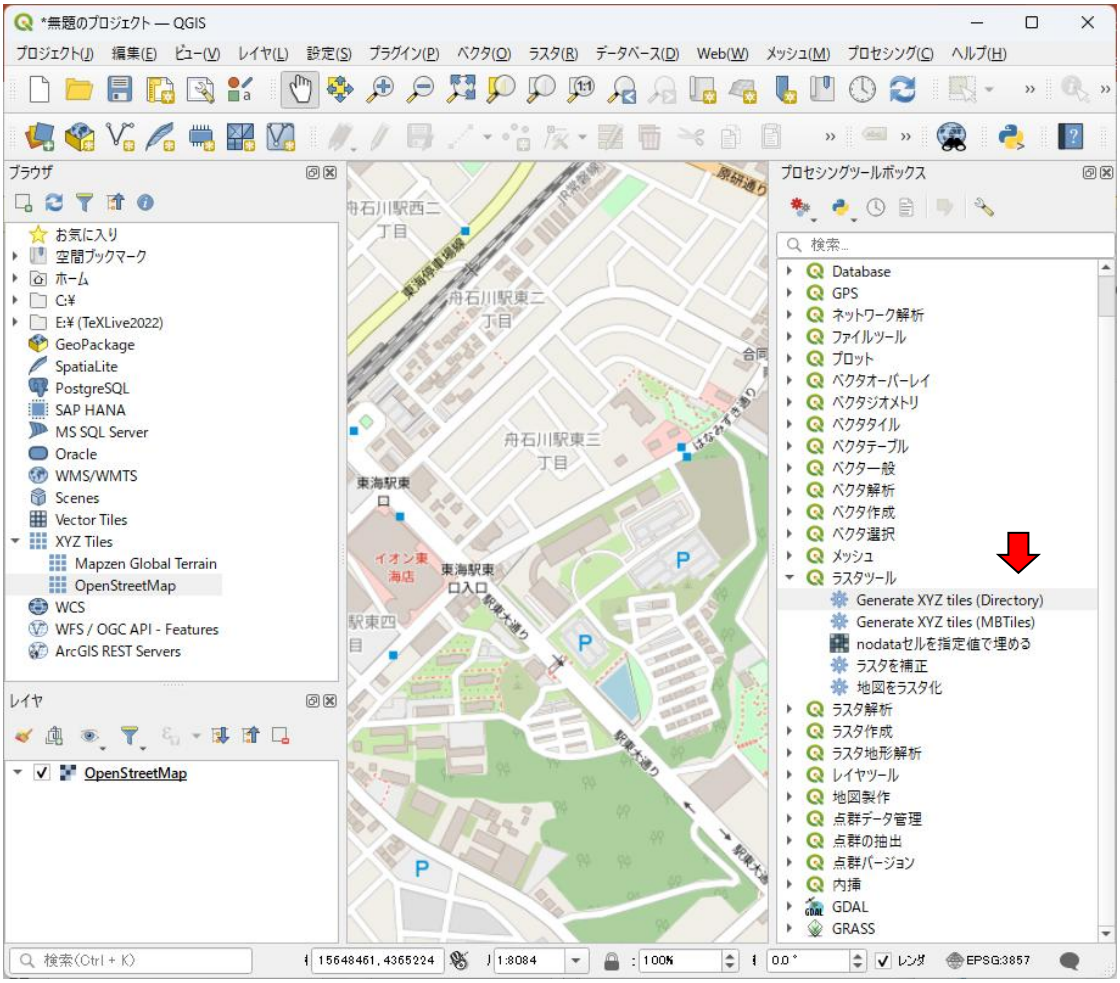

図 B-2　プロセシングツールボックスの Generate XYZ tiles の選択画面
中央の地図にオフラインでマップを描画したい範囲を表示しておく。
（ここでは日本原子力研究開発機構本部周辺を表示している。）

6) 図 B-3 に示すラスタツールの画面で、「領域」、「最小ズーム」、「最大ズーム」、「Tile format」、「出力フォルダ」など、必要なパラメーターを入力する。「領域」の設定では、「現在のキャンバス領域を利用」を選択すると、表示している地図の範囲が自動的に入力される。

図 B-3　ラスタツールのパラメーター入力画面

「領域」の入力では、赤枠で囲ったアイコン（またはプルダウン中の「現在のキャンバス領域を利用」）を選択すると、描画中の地図の範囲が自動で入力される。

7) 図 B-4 の通り、出力フォルダにファイルが書き出される。書き出されたフォルダの中身を Leaflet で読み込むフォルダにコピーする。

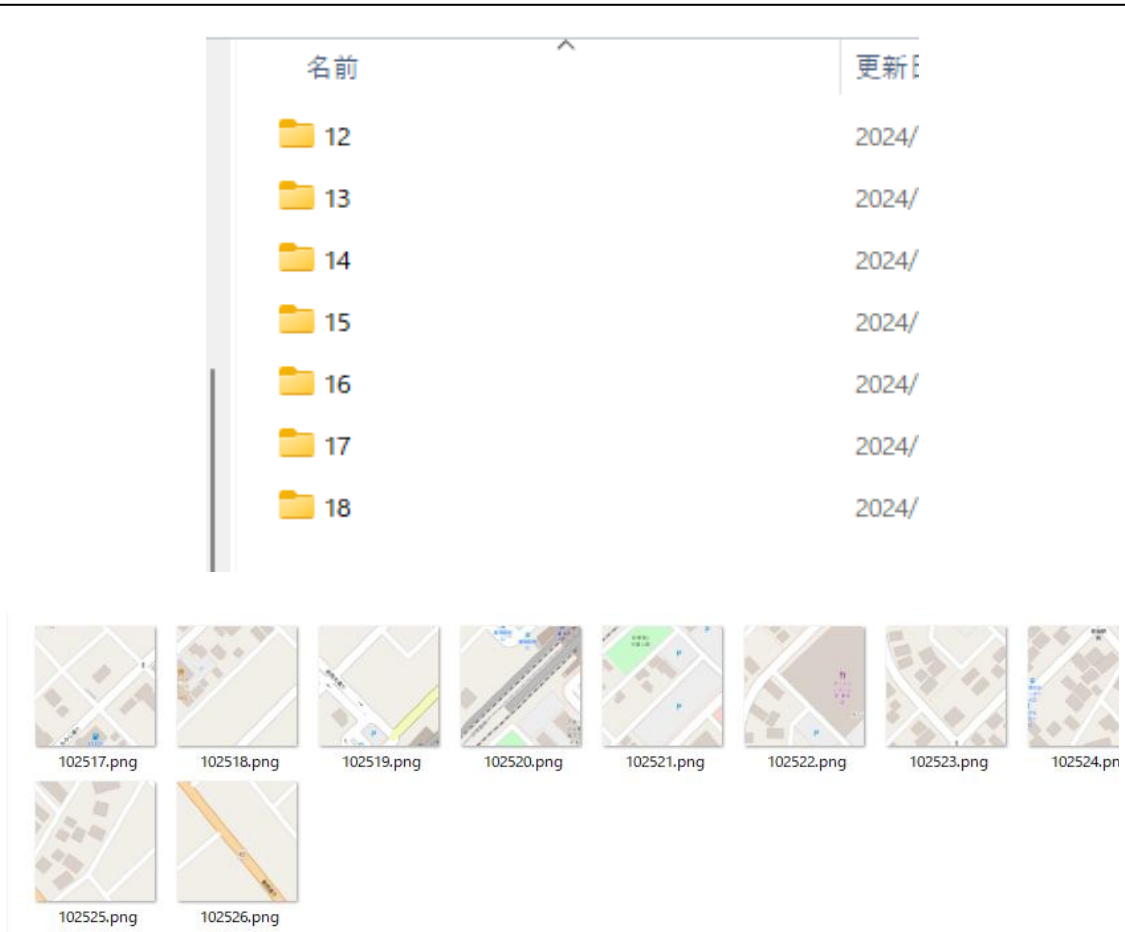

図 B-4　画像ファイルを出力したフォルダの構造
「最小ズーム」を 12、「最大ズーム」を 18 で出力した場合の例（上段）。
各フォルダの中には、画像ファイルが保存される（下段）。

QGIS では、OpenStreetMap に限らず、Web で公開されている様々な形式のマップタイル（ラスタ形式、ベクタ形式問わず）に対応している。