Spectral Data Implementation Plan

Overview

Add support for gamma spectroscopy data from scintillator-based radiation detectors to the Chicha Isotope Map. Users will be able to upload tracks with spectral measurements, view spectrum graphs in marker popups, filter by energy ranges, and download spectrum files.

Requirements Summary

Supported Formats

- RadiaCode .rctrk files with embedded spectrum data
- ANSI N42.42 XML format (https://www.nist.gov/document/annexbn42xml)

Supported Devices

- RadiaCode 101/102/103
- AtomFast
- Other scintillator-based detectors

Spectral Data Characteristics

- 1024 channels covering 0-3000 keV energy range
- Attached to specific measurement points (not all track points)
- Special marker icon to indicate spectral data availability

User Features

1. Click marker with spectral data → popup shows spectrum graph
2. Heatmap overlay showing specific energy ranges
3. Download spectrum files for individual measurements
4. Filter/toggle markers with spectral data

Implementation Plan

Phase 1: Database Schema Extension

1.1 Create `spectra` table

New table to store spectral measurements:

```
CREATE TABLE IF NOT EXISTS spectra (
  id              BIGSERIAL PRIMARY KEY,
  marker_id       BIGINT NOT NULL,            -- Foreign key to
markers.id
  channels        BYTEA,                       -- Binary array of 1024
channel counts (or JSON)
  channel_count   INTEGER DEFAULT 1024,       -- Number of channels
  energy_min_kev  DOUBLE PRECISION,           -- Minimum energy (keV)
  energy_max_kev  DOUBLE PRECISION,           -- Maximum energy (keV)
  live_time_sec   DOUBLE PRECISION,           -- Live time (seconds)
  real_time_sec   DOUBLE PRECISION,           -- Real time (seconds)
  device_model    TEXT,                       -- Detector model (e.g.,
"RadiaCode-102")
  calibration     TEXT,                       -- JSON: energy
calibration coefficients
  source_format   TEXT,                       -- Original format:
"rctrk", "n42"
  raw_data        BYTEA,                      -- Optional: original
spectrum file for download
  created_at      TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_spectra_marker_id ON spectra(marker_id);
```

1.2 Modify `markers` table

Add flag to indicate spectrum availability:

```
ALTER TABLE markers ADD COLUMN has_spectrum BOOLEAN DEFAULT FALSE;
CREATE INDEX idx_markers_has_spectrum ON markers(has_spectrum) WHERE
has_spectrum = TRUE;
```

Phase 2: Data Models (Go)

2.1 Add to `pkg/database/models.go`

```
// Spectrum represents gamma spectroscopy data for a measurement point
type Spectrum struct {
    ID            int64                `json:"id"`
    MarkerID      int64                `json:"markerID"`
    Channels      []int                `json:"channels"`          //
1024 channel counts
    ChannelCount  int                  `json:"channelCount"`      //
Typically 1024
    EnergyMinKeV  float64              `json:"energyMinKeV"`      //
e.g., 0
    EnergyMaxKeV  float64              `json:"energyMaxKeV"`      //
e.g., 3000
    LiveTimeSec   float64              `json:"liveTimeSec"`       //
Measurement live time
```

```
    RealTimeSec   float64            `json:"realTimeSec"`        //
Actual elapsed time
    DeviceModel   string             `json:"deviceModel"`        //
"RadiaCode-102"
    Calibration   *EnergyCalibration `json:"calibration"`        //
Energy calibration
    SourceFormat  string             `json:"sourceFormat"`       //
"rctrk" or "n42"
    RawData       []byte             `json:"-"`                  //
Original file bytes
}

// EnergyCalibration stores polynomial coefficients for energy = a +
b*channel + c*channel^2
type EnergyCalibration struct {
    A float64 `json:"a"` // Offset
    B float64 `json:"b"` // Linear
    C float64 `json:"c"` // Quadratic
}

// Modify existing Marker struct
type Marker struct {
    // ... existing fields ...
    HasSpectrum bool `json:"hasSpectrum,omitempty"` // NEW field
}
```

Phase 3: File Parsers

3.1 Create `pkg/spectrum/` package

pkg/spectrum/rctrk.go - Parse RadiaCode .rctrk with spectrum

```
package spectrum

// ParseRCTRK parses a RadiaCode .rctrk file and extracts spectrum
data
// RadiaCode .rctrk files can contain a "spectra" array with spectrum
objects
func ParseRCTRK(data []byte) ([]Spectrum, error)
```

pkg/spectrum/n42.go - Parse ANSI N42.42 XML

```
package spectrum

// ParseN42 parses ANSI N42.42 XML format
// Reference: https://www.nist.gov/document/annexbn42xml
func ParseN42(data []byte) ([]Spectrum, error)
```

pkg/spectrum/spectrum.go - Common utilities

```
package spectrum

// ConvertToJSON serializes channel data for database storage
func ConvertToJSON(channels []int) ([]byte, error)

// ConvertFromJSON deserializes channel data from database
func ConvertFromJSON(data []byte) ([]int, error)

// ChannelToEnergy converts channel number to energy (keV) using
calibration
func ChannelToEnergy(channel int, cal *EnergyCalibration) float64

// DetectPeaks performs simple peak detection on spectrum data
func DetectPeaks(channels []int, threshold float64) []Peak
```

3.2 Integrate parsers into upload handlers

Modify `chicha-isotope-map.go`:

- In `processRCTRKFile`: Check for embedded spectrum data
- Add new handler `processN42File` for .xml uploads
- Link spectrum records to their corresponding markers

Phase 4: Database Operations

4.1 Add to `pkg/database/spectrum.go` (new file)

```
// InsertSpectrum stores a spectrum record linked to a marker
func (db *Database) InsertSpectrum(ctx context.Context, spectrum
Spectrum) (int64, error)

// GetSpectrum retrieves spectrum data for a marker
func (db *Database) GetSpectrum(ctx context.Context, markerID int64)
(*Spectrum, error)

// GetMarkersWithSpectra returns markers that have spectral data in a
bounding box
func (db *Database) GetMarkersWithSpectra(ctx context.Context, bounds
Bounds) ([]Marker, error)

// DeleteSpectrum removes spectrum data (for track deletion)
func (db *Database) DeleteSpectrum(ctx context.Context, markerID
int64) error
```

Phase 5: Backend API Endpoints

Add to `chicha-isotope-map.go`:

5.1 Get spectrum data

```
GET /api/spectrum/{markerID}
Returns: JSON with spectrum data (channels, calibration, metadata)
```

5.2 Download spectrum file

```
GET /api/spectrum/{markerID}/download?format=n42|json|csv
Returns: Original spectrum file or converted format
```

5.3 Get markers with spectra

```
GET /api/markers/spectra?minLat=...&maxLat=...&minLon=...&maxLon=...
Returns: GeoJSON of markers that have spectral data
```

5.4 Energy range query (for heatmap)

```
GET /api/markers/energy-range?minKeV=...&maxKeV=...&minLat=...
Returns: Markers with integrated counts in specified energy range
```

Phase 6: Frontend - Marker Display

6.1 Special marker icons

Create new marker types in `public_html/map.html`:

- Regular marker: current circle markers
- Spectrum marker: circle with a special icon/color (e.g., star overlay, different color)

Modify `addMarkerToMap()` function to check `marker.hasSpectrum` flag.

6.2 Marker filtering

Add new filter toggle:

- "Show only markers with spectra" checkbox
- Update `loadMarkers()` to filter based on this toggle

Phase 7: Frontend - Spectrum Visualization

7.1 Choose charting library

Options:

- **Chart.js** (lightweight, easy to use) - RECOMMENDED
- Plotly.js (more features, larger bundle)
- D3.js (most flexible, steeper learning curve)

Add Chart.js to `public_html/map.html`:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js@4"></script>
```

7.2 Create spectrum popup component

Add to `public_html/map.html`:

```
// Load spectrum data when marker with spectrum is clicked
async function loadSpectrum(markerID) {
  const resp = await fetch(`/api/spectrum/${markerID}`);
  const spectrum = await resp.json();
  return spectrum;
}

// Render spectrum chart in popup
function renderSpectrumChart(canvasId, spectrum) {
  const ctx = document.getElementById(canvasId).getContext('2d');

  // Convert channels to energy values
  const energies = spectrum.channels.map((count, channel) => {
    return channelToEnergy(channel, spectrum.calibration);
  });

  new Chart(ctx, {
    type: 'line',
    data: {
      labels: energies,
      datasets: [{
        label: 'Counts',
        data: spectrum.channels,
        borderColor: 'rgb(75, 192, 192)',
        tension: 0.1
      }]
    },
    options: {
      scales: {
        x: { title: { display: true, text: 'Energy (keV)' } },
        y: {
          type: 'logarithmic',
          title: { display: true, text: 'Counts' }
        }
      },
      responsive: true,
      maintainAspectRatio: false
    }
  });
}
```

7.3 Modify marker popup

Update popup creation to include spectrum chart when available:

```
function createMarkerPopup(marker) {
  let content = `
    <div class="marker-popup">
      <strong>Dose Rate:</strong> ${marker.doseRate} µSv/h<br>
      <strong>Date:</strong> ${new Date(marker.date *
1000).toLocaleString()}<br>
      <!-- ... existing fields ... -->
  `;

  if (marker.hasSpectrum) {
    content += `
      <hr>
      <div class="spectrum-section">
        <strong>Gamma Spectrum</strong>
        <canvas id="spectrum-chart-${marker.id}" width="400"
height="300"></canvas>
        <button onclick="downloadSpectrum(${marker.id})">Download
Spectrum</button>
      </div>
    `;
  }

  content += `</div>`;
  return content;
}

// Load spectrum when popup opens
marker.on('popupopen', async function() {
  if (marker.hasSpectrum) {
    const spectrum = await loadSpectrum(marker.id);
    renderSpectrumChart(`spectrum-chart-${marker.id}`, spectrum);
  }
});
```

Phase 8: Frontend - Energy Range Heatmap

8.1 Add energy range filter UI

Add controls to select energy range:

```
<div class="energy-filter">
  <label>Energy Range (keV):</label>
  <input type="number" id="energyMin" placeholder="Min" value="0">
  <input type="number" id="energyMax" placeholder="Max" value="3000">
  <button onclick="updateEnergyHeatmap()">Show Heatmap</button>
</div>
```

8.2 Render heatmap overlay

Use Leaflet heatmap plugin or custom implementation:

```
async function updateEnergyHeatmap() {
  const minKeV = document.getElementById('energyMin').value;
  const maxKeV = document.getElementById('energyMax').value;

  const resp = await fetch(`/api/markers/energy-
range?minKeV=${minKeV}&maxKeV=${maxKeV}&minLat=...`);
  const data = await resp.json();

  // Render heatmap layer with intensity based on integrated counts
  // in specified energy range
}
```

Phase 9: Data Export

9.1 Include spectra in track exports

Modify `pkg/jsonarchive/generator.go`:

- When exporting a track, check if markers have spectra
- Include spectrum data in JSON export
- Optionally create separate spectrum files (.n42 or .json) in archive

9.2 Download individual spectrum

Implement download endpoint to return spectrum in multiple formats:

- N42 XML (standard format)
- JSON (easy for web apps)
- CSV (simple tabular format: energy,counts)

Phase 10: Testing & Documentation

10.1 Test cases

- Upload .rctrk file with spectrum data
- Upload N42 XML file
- Verify database storage

- Test spectrum popup display
- Test energy range filtering
- Test spectrum download

10.2 Documentation

- Update README.md with spectral data features
- Add examples of supported spectrum formats
- Document API endpoints for spectrum access

Implementation Order (Recommended)

1. **Database schema** (Phase 1) - Foundation
2. **Data models** (Phase 2) - Go structs
3. **File parsers** (Phase 3) - Start with .rctrk, then N42
4. **Database operations** (Phase 4) - CRUD for spectra
5. **Basic API endpoint** (Phase 5.1) - GET spectrum data
6. **Frontend marker icons** (Phase 6) - Visual indicator
7. **Frontend popup chart** (Phase 7) - Core user feature
8. **Download endpoint** (Phase 5.2) - Export capability
9. **Energy filtering** (Phase 5.4 + Phase 8) - Advanced feature
10. **Data export integration** (Phase 9) - Complete the loop
11. **Testing** (Phase 10) - Verify everything works

Technical Considerations

Storage Format for Channel Data

Option 1: JSON array in TEXT column

- Pros: Easy to query, human-readable
- Cons: Larger storage size

Option 2: Binary BYTEA (packed integers)

- Pros: Compact, efficient
- Cons: Need encoding/decoding logic

Recommendation: Start with JSON for simplicity, optimize to binary if needed.

Energy Calibration

RadiaCode and other devices provide calibration coefficients:

```
Energy(keV) = A + B * channel + C * channel^2
```

Store A, B, C in calibration JSON field.

Spectrum File Size

1024 channels × 4 bytes/int = ~4 KB per spectrum (very manageable)

API Performance

- Spectrum data is not loaded with regular markers (lazy loading)
- Only fetched when user clicks marker
- Consider caching frequently accessed spectra

Example File Formats

RadiaCode .rctrk with spectrum (hypothetical)

```
{
  "id": "ABC123",
  "markers": [
    {
      "lat": 44.08,
      "lon": 43.03,
      "doseRate": 0.33,
      "countRate": 31,
      "date": 1728127340,
      "spectrum": {
        "channels": [0, 2, 5, 10, ...],  // 1024 values
        "liveTime": 60.0,
        "realTime": 61.2,
        "calibration": {
          "a": 0,
          "b": 2.93,
          "c": 0
        }
      }
    }
  ]
}
```

N42 XML (simplified)

```
<RadInstrumentData>
  <Measurement>
```

```
    <Spectrum>
      <LiveTimeDuration>PT60S</LiveTimeDuration>
      <ChannelData>0 2 5 10 15 ...</ChannelData>
      <EnergyCalibration>
        <Coefficients>0 2.93 0</Coefficients>
      </EnergyCalibration>
    </Spectrum>
  </Measurement>
</RadInstrumentData>
```

Future Enhancements (Post-MVP)

1. **Isotope identification**: Auto-detect common isotopes (Cs-137, Co-60, etc.)
2. **Peak labeling**: Annotate energy peaks on spectrum chart
3. **Spectrum comparison**: Compare multiple spectra side-by-side
4. **Background subtraction**: Subtract background spectrum
5. **Region of interest (ROI)**: Define energy windows for specific isotopes
6. **Dose contribution**: Calculate dose contribution from specific energy ranges
7. **3D visualization**: Show spectrum evolution along a track

---

Questions to Resolve During Implementation

1. **RadiaCode .rctrk format**: Need to find/create example .rctrk file with embedded spectrum to understand exact JSON structure
2. **N42 parsing complexity**: ANSI N42.42 is extensive - start with subset of features?
3. **Marker clustering**: How to handle spectrum markers when clustered? Show count of spectra in cluster?
4. **Mobile display**: Spectrum chart rendering on small screens - responsive design?
5. **Real-time spectra**: Should Safecast realtime feed support spectra? (Probably not in MVP)