Санкт-Петербургский государственный университет

КУЗЬМИН Павел Викторович

Выпускная квалификационная работа

Применение методов автоматизации в среде географических информационных систем при оценке характера пространственного распределения населения в трансграничных регионах

Уровень образования: бакалавриат

Направление: 05.03.03 «Картография и геоинформатика»

Образовательная программа: 05.03.03 «Картография и геоинформатика»

Научный руководитель: доцент кафедры
картографии и геоинформатики, к.т.н.
Паниди Евгений Александрович

Рецензент: старший научный сотрудник, Лаборатория геополитических исследований,
ФГБУН Институт географии Российской академии наук, к.г.н.
Себенцов Александр Борисович

Санкт-Петербург
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Плотность населения является одним из важнейших факторов устойчивого развития заселенных территорий, так как она во многом связана с рисками негативных воздействий как на собственно население, так и на окружающую среду. Существует множество подходов к прямой оценке (описанию, моделированию) плотности населения, при этом классическим является применение гравитационной модели (Sen, Smith, 1995) к населенным пунктам. Гравитационная модель описывает взаимодействие между пространственными объектами (населенными пунктами в данном случае). Она предполагает интерполяцию географической переменной, называемой потенциалом поля расселения (settlement field potential) – ППР (SFP). При этом ГИС применяются в качестве инструмента картографирования ППР. Сам термин ППР и формула, по которой он вычисляется, варьируются от автора к автору (Dong и др., 2022; Создаев, Тесленок, 2019). В первом приближении ППР оценивает степень взаимного воздействия объектов (населенных пунктов) друг на друга по численности их населения и прямолинейному расстоянию между ними.

Учитывая, что речь идёт о географическом пространстве, можно сделать вывод, что расстояние по прямой не имеет единой интерпретации. Особенно в обширных регионах, где его оценка будет меняться в зависимости от проекции карты. Более того, связи населения, транспорта или экономические связи между населенными пунктами осуществляются не по прямой, а по дорожной сети. В таком случае близко расположенные объекты могут оказаться разделены географическим барьером любого характера, а расстояние по дорожной сети может быть ощутимо больше прямолинейного. В связи с этим **целью** данной работы является разработка методических и алгоритмических средств для моделирования и картографической визуализации потенциала поля расселения с учётом реальных транспортных связей.

Задачи работы:

- Подбор средства массовой автоматической маршрутизации
- Разработка алгоритма подготовки к использованию и использования средства массовой автоматической маршрутизации
- Разработка алгоритма получения значений потенциала поля расселения
- Разработка средства картографической визуализации потенциала поля расселения

Актуальность данной работы заключается в том, что такая модифицированная гравитационная модель особенно востребована при исследовании трансграничных регионов (Головина и др., 2015), где миграция через государственную границу возможна только через пропускные пункты (рис. 1). Работ с ранее реализованной подобной модификацией выявлено не было.

Объектом исследования выступает гравитационная модель территориального распределения населения, а **предметом исследования** является реализация гравитационной модели территориального распределения населения в среде географических информационных систем.

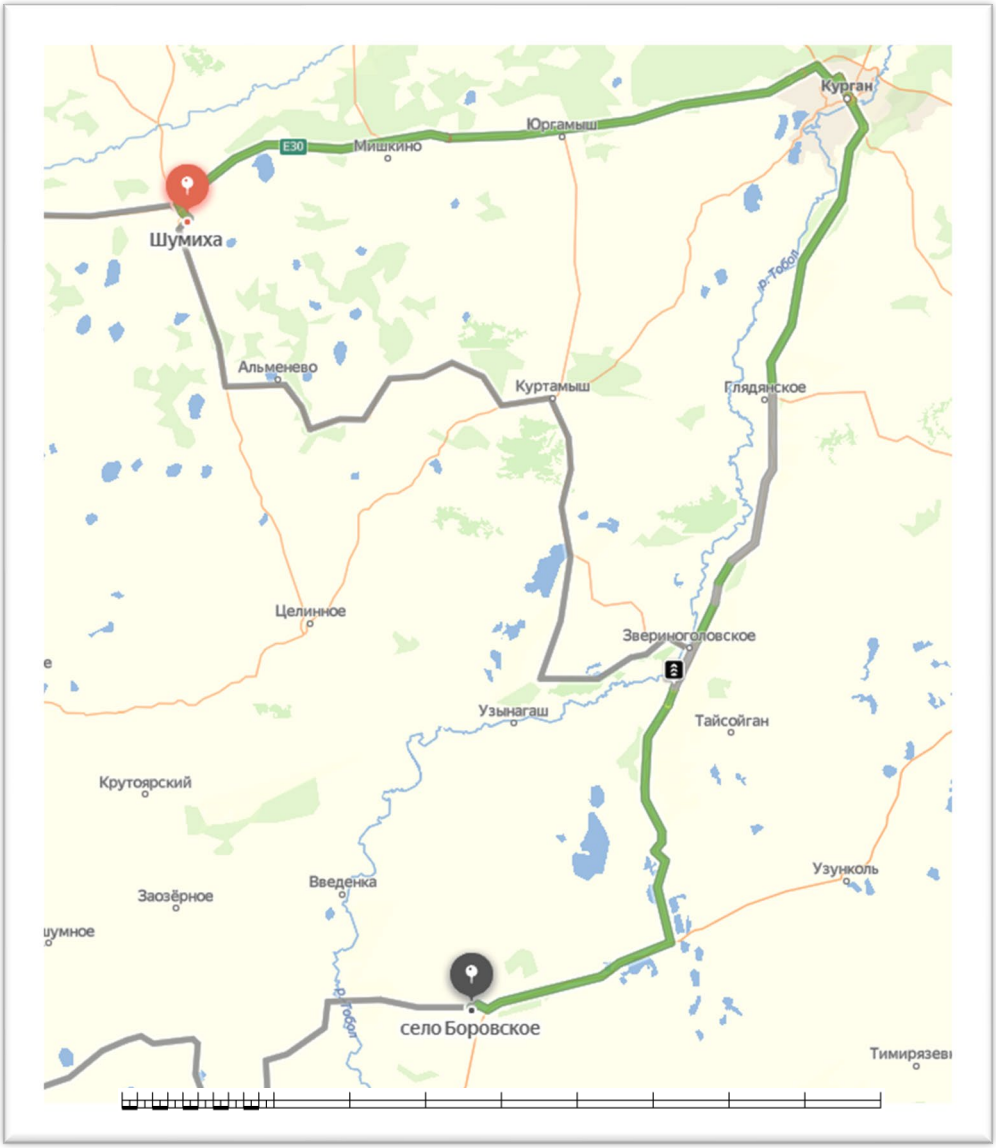

Рисунок 1. Удлиненный дорожной сетью маршрут через АПП Звериноголовское, Курганская обл. (Яндекс.Карты)

1. Используемые данные и методика

Наиболее распространенная формула для расчёта потенциала поля расселения выглядит следующим образом (Dong и др., 2022; Создаев, Тесленок, 2019):

$$V_i = H_i + \sum_{j=2}^{j=n} \frac{H_j}{R_{ij}}$$

n – количество населенных пунктов,

V_i – показатель потенциала поля расселения в данной точке,

H_i – численность населения i-ого населенного пункта,

H_j – численность населения остальных населенных пунктов,

R_{ij} – расстояние между i-ым и j-ым населенным пунктом.

Соответственно, необходимы данные о численности населенных пунктов и расстояния между ними по дорожной сети.

Имеются наборы данных в виде таблиц Excel с координатами (широтой и долготой) и численностью населённых пунктов Казахстана, Белоруссии, Финляндии и России в разные годы. Населенные пункты расположены в трансграничных регионах. Российско-Казахстанская граница – 422 пункта, Российско-Белорусская граница – 290 пунктов, Российско-Финляндская граница – 150 пунктов.

После предварительной обработки данных и ознакомления с ними был оценен географический охват работы. Геоданные визуализировались (рис. 2) при помощи настольной ГИС QGIS (*https://www.qgis.org/ru/site/*).

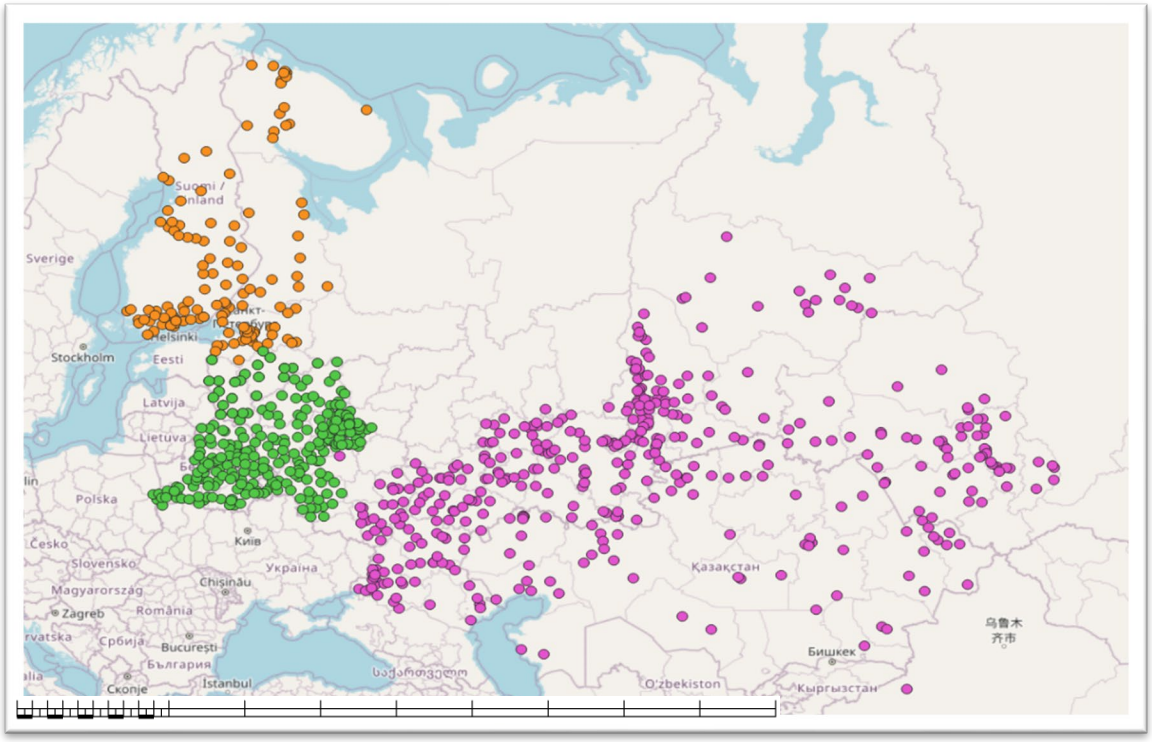

Рисунок 2. Населённые пункты, участвующие в вычислениях.

Каждый населенный пункт соединится с остальными в своей группе расстоянием по дорожной сети и рассчитается потенциал поля расселения. Таким образом, возникает подзадача построения матрицы (квадратной) расстояний по дорожной сети между населенными пунктами. Соответственно, количество расстояний в матрице будет равно квадрату количества населенных пунктов в каждой группе (за вычетом количества расстояний по главной диагонали).

Однако, использование населенных пунктов в качестве узлов будущей интерполяции потенциала поля расселения имеет ряд недостатков. Ввиду нерегулярного расположения и местами малого числа таких узлов на единицу площади, интерполяция будет неустойчивой. Для повышения устойчивости интерполяции на обширных территориях некоторые авторы предлагают применять регулярные сетки (Колосов, 2014), в них узлы интерполяции расположены с постоянным интервалом. Такая сетка создаётся по охвату набора населенных пунктов, но с захватом соседней территории в пределах 300 км. От каждого узла регулярной сетки прокладываются маршруты до населенных пунктов и рассчитывается потенциал поля расселения.

Таким образом, строится не квадратная матрица, где точки из списка координат соединяются каждая с каждой, а прямоугольная с начальными пунктами в виде узлов сетки и конечными в виде населенных пунктов. Основная проблема такого решения заключается в большом количестве вычислений, речь идёт о миллионах расстояний по

кратчайшим маршрутам. Приведенный пример с Российско-Казахстанской границей содержит 422 населенных пункта и регулярную сетку для них с шагом в 10 км. Из этого следует, что сетка состоит из более чем 100 тыс. узлов и, соответственно, матрица расстояний из более чем 40 млн. значений.

Используется следующая формула расчёта потенциала поля расселения:

$$V_i = \sum_{j=1}^{j=n} \begin{cases} \frac{H_j}{R_{ij}}, R_{ij} \leq 50 \\ \frac{H_j}{2R_{ij}}, 50 < R_{ij} \leq 300 \\ 0, R_{ij} > 300 \end{cases}$$

n – количество населенных пунктов,

V_i – показатель потенциала поля расселения в данной точке, чел./м,

H_j – численность населения *j*-ого населенного пункта, тыс. чел.,

R_{ij} – расстояние между данной точкой и *j*-ым населенным пунктом, км.

Расчёт потенциала поля расселения осуществляется на языке программирования Python (*https://www.python.org/*). Python – это высокоуровневый интерпретируемый язык программирования, который был впервые выпущен в 1991 году. Он известен своей простотой и удобочитаемостью. Его безусловное преимущество для данной работы в том, что при своей простоте он имеет очень мощные надстраиваемые библиотеки для манипулирования данными и геоданными в особенности.

Используемые в работе сторонние библиотеки Python:

- Весь программный код построен на возможностях Jupyter (*https://jupyter.org/*), интерактивного блокнота Python, выполняющего отдельные блоки кода в ячейках, позволяющих получать вывод в HTML. Соответственно, вывод может содержать всё, что позволяет HTML (картинки, форматированный текст, JavaScript и т. д.).
- Расчёты по формуле ППР выполняются методами библиотеки NumPy (*https://numpy.org/*), позволяющей оперировать массивами, причём векторизовано – без итераций по ним, итерации и остальная механика осуществляется в низкоуровневых модулях NumPy, написанных на языке C, за счёт этого программа с NumPy становится более быстрой и эффективной по памяти.

- Для чтения данных из разных форматов и их экспорта в работе применяется Pandas (*https://pandas.pydata.org/*), он предоставляет высокоуровневые структуры данных, которые основаны на NumPy. Поэтому с помощью Pandas удобно считывать данные в массивы NumPy, а экспортировать в таблицы Excel.
- Некоторые промежуточные визуализации в блокнотах Jupyter получаются благодаря Matplotlib (*https://matplotlib.org/*), библиотеки для построения графиков. Matplotlib совместима с NumPy и Pandas, и, вместе с библиотекой SciPy, они формируют основополагающий стек для работы с данными и научных вычислений на Python.
- Координатные преобразования осуществляются библиотекой Pyproj (*https://pyproj4.github.io/pyproj*), которая основана на C библиотеке PROJ.
- Для создания и экспорта геопривязанного растра используется Rasterio (*https://rasterio.readthedocs.io*). Библиотека основана на C++ библиотеке GDAL. Благодаря своей интеграции с NumPy, Rasterio позволяет эффективно работать с большими объемами данных и выполнять сложные аналитические задачи.
- Кроме Matplotlib некоторые промежуточные интерактивные картографические визуализации используют библиотеку Folium (*https://python-visualization.github.io/folium/*). Она основана на JavaScript-библиотеке Leaflet и позволяет встраивать интерактивные веб-карты в HTML выводимый Jupyter'ом.
- Наконец, RoutingPy (*https://routingpy.readthedocs.io/*) предоставляет API в Python для HTTP-запросов к различным сервисам маршрутизации. Важно отметить, что он позволяет указать конкретные начальные (sources) и конечные (destinations) пункты для вычисления прямоугольной матрицы расстояний между населенными пунктами и узлами регулярной сетки.

1.1. Краткий обзор сервисов маршрутизации в транспортных сетях

Чтобы построить матрицу расстояний по дорожной сети, в первую очередь необходимо рассмотреть основные инструменты, осуществляющие маршрутизацию в транспортных сетях – механизмы (движки) маршрутизации.

Маршрутизация – одна из распространенных транспортных задач. Как? Сколько? На чём? Это лишь небольшой перечень вопросов, на которые можно получить ответ, пользуясь имеющимися на сегодняшний день способами решения данной задачи. С помощью картографических веб-сервисов современных IT-гигантов, таких как Яндекс, Google задача маршрутизации в транспортных сетях, в общем, успешно решается. Для личных потребностей их сервисов более чем достаточно и в смысле качества, и в смысле удобства. Однако проблема становится более явной, когда речь идёт о промышленных масштабах. Проложить вручную несколько маршрутов на Google Maps обычное дело, но, когда необходимо относительно быстро и не менее качественно автоматически проложить сотни тысяч и даже миллионы маршрутов за раз, проблема становится ощутимой.

Упомянутые сервисы, конечно же, существуют и для массовой маршрутизации, однако использование их в данном случае уже тарифицируется. Геоданные и ресурсы серверов таких компаний, как Google не доступны для свободного пользования (рис. 3).

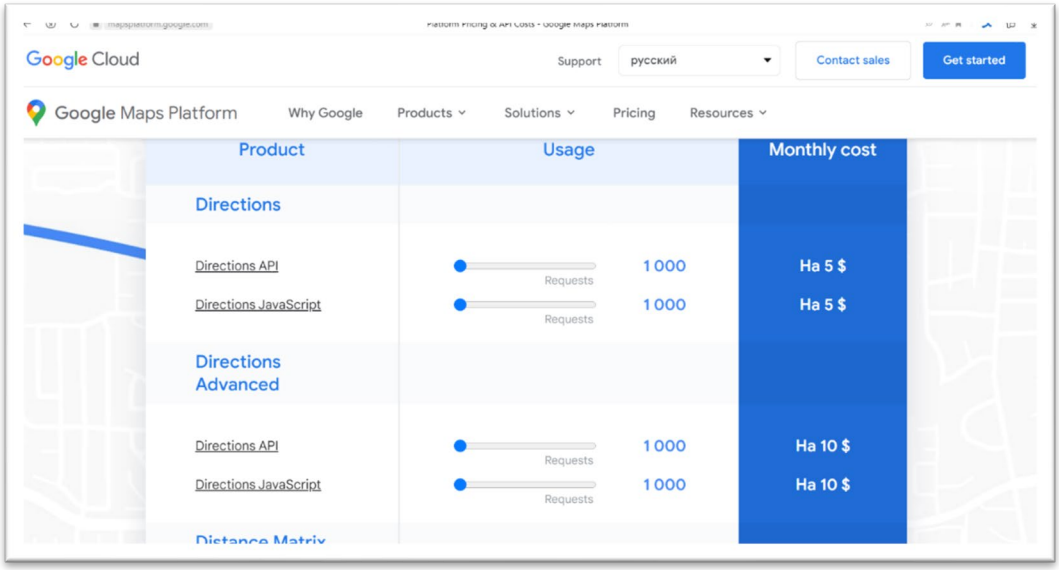

Рисунок 3. Услуги платформы Google Maps API (https://mapsplatform.google.com/intl/ru/pricing/)

Существует и некоммерческий веб-картографический проект – OpenStreetMap (OSM). Геоданные OSM являются публичными и будут использоваться здесь далее. На основе данных OSM есть открытый веб-сервер от FOSSGIS (*https://routing.openstreetmap.de/*), использующий механизм маршрутизации с открытым исходным кодом Open Source Routing Machine (OSRM). Он принимает HTTP-запросы на маршрутизацию и возвращает результаты. Объём в данном случае по-прежнему ограничивается, но есть способы обхода ограничений.

Удачным выглядит использование свободно распространяемых установленных локально механизмов маршрутизации с открытым исходным кодом. Такие решения можно развернуть (установить) с использованием технологии контейнеризации Docker.

В данной работе применяются локальные механизмы маршрутизации OSRM (*https://github.com/Project-OSRM/osrm-backend*) и Valhalla (*https://github.com/valhalla/valhalla*). Оба механизма осуществляют маршрутизацию на основе геоданных OSM. Для извлечения геоданных интересующего охвата понадобится библиотека Osmium, с её установкой в виде утилиты командной строки также поможет Docker. В целом движок OSRM популярнее, чем Valhalla – более 100 млн. скачиваний против более 1 млн. Пользователи находят OSRM более быстрым, тогда как Valhalla более гибким движком. Для данной работы бóльшую важность имеет скорее скорость работы.

1.2. Немного о контейнерах и Docker

Основная проблема открытых механизмов маршрутизации для локальной развёртки – необходимость компиляции их из исходного кода, а также сама последовательность действий для развёртки, зависящая от конкретной операционной системы. Кроме того, предварительно необходимо, как минимум, установить компилятор и зависимости приложения, задать определенные конфигурации. Зачастую эти действия могут быть совсем нетривиальными, так как являются в определенном смысле продолжением программирования приложения.

Технология контейнеров сильно упрощает процесс развёртки (установки). Приложение в контейнере уже является по сути установленным и скомпилированным, так как контейнер содержит в своём основании псевдооперационную систему и все зависимости приложения поверх нее. Docker (*https://www.docker.com/*) же обеспечивает связь между легковесной псевдооперационной системой в контейнере и операционной системой хоста (рис. 4). Это позволяет контейнерам быть мобильными и простыми для запуска в любой среде на базе ПО Docker. При этом контейнеры Docker следует отличать от виртуальных машин.

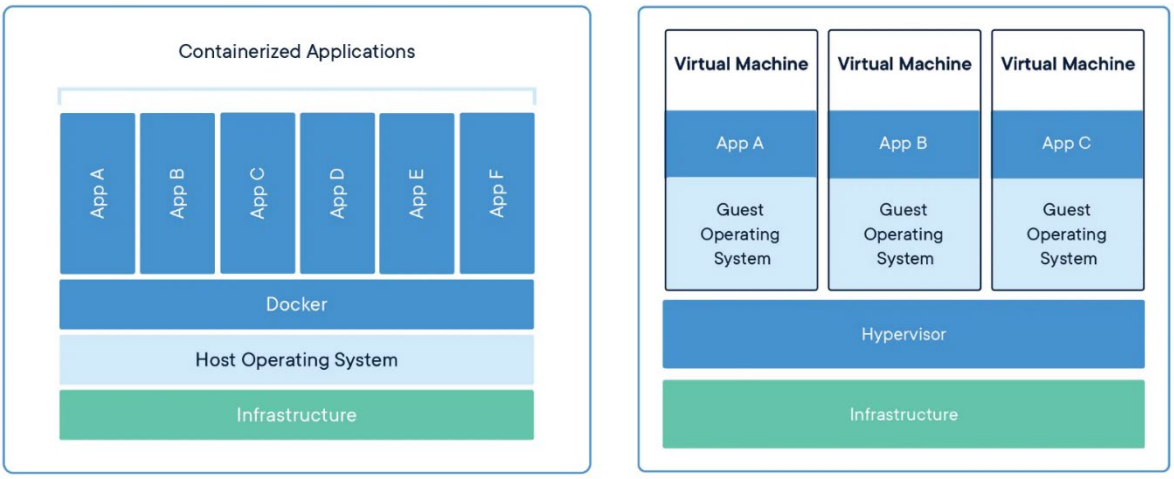

Рисунок 4. Сравнение контейнеров и виртуальных машин (https://www.docker.com/resources/what-container/)

Образы контейнеров Docker размещаются в публичном репозитории Docker Hub (*https://hub.docker.com/*). При наличии доступа к сети Интернет они загружаются на локальный хост посредством командной строки: docker pull. Также при запуске контейнера командой docker run, в случае отсутствия указанного контейнера на локальной машине, Docker попытается загрузить его из Docker Hub. Разумеется, можно создавать и свои образы и загружать их на Docker Hub.

Docker можно установить на Linux, MacOS, Windows 10/11. Для установки на Windows должна быть включена поддержка аппаратной виртуализации на уровне BIOS, а также доступны WSL2 (Windows Subsystem for Linux – это, например, Ubuntu прямо в Windows) или Hyper-V.

1.3. Картографическая визуализация

Картографическая визуализация полученных результатов осуществляется посредством веб-сайта с интерактивной картой, на которой отображены слои подложки и растры значений ППР по каждому году и региону, а также сопутствующая информация. Веб-сайт адаптирован для просмотра на смартфонах и персональных компьютерах.

Для реализации интерактивной карты на веб-сайте используется JavaScript-библиотека OpenLayers (*https://openlayers.org/*). JavaScript – де-факто стандартный язык программирования динамических интерфейсов веб-страниц, тогда как для их просмотра

используется веб-браузер. Любой популярный веб-браузер является средой выполнения JavaScript кода и поддерживает всю его функциональность.

Библиотека OpenLayers позволяет легко разместить динамическую карту на любой веб-странице. Она может отображать тайловые карты, векторные, растровые данные и маркеры. Является полностью бесплатной с открытым исходным кодом, выпущенной по лицензии 2-clause BSD (также известной как FreeBSD).

Переключение слоев на веб-карте осуществляется с помощью плагина OpenLayers LayerSwitcher (*https://github.com/walkermatt/ol-layerswitcher*). Также на сайт добавляется возможность просмотра в 3D на глобусе для устранения искажений проекции карты. Это осуществлется посредством JavaScript-библиотеки Cesium.js (*https://cesium.com/platform/cesiumjs/*) и дополнительного модуля для её связи с OpenLayers (*https://openlayers.org/ol-cesium/*). Cesium.js использует WebGL для отображения высоко детализированных 3D моделей земной поверхности, а также может работать с другими геопространственными данными, такими как снимки спутников, карты высот и векторные данные.

Кроме веб-браузеров средой выполнения JavaScript кода является Node.js (*https://nodejs.org/*) – «программная платформа, основанная на движке V8 (компилирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API, написанный на C++, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода.» – (Википедия. Node.js). Использование Node.js удобно и при программировании на JavaScript для браузеров, так как платформа поставляется со встроенным менеджером пакетов Node Package Manager (npm), с помощью которого удобно создавать шаблоны различных приложений (в том числе приложения с OpenLayers), включающих сборщик и отладчик. В шаблоне OpenLayers в качестве сборщика и отладчика выступает Vite (работает на Node.js).

В качестве картографической основы для растров ППР удачной идей выглядит использование общедоступной подложки Stamen Toner (*http://maps.stamen.com/*), она предоставляется в том числе послойно, что позволяет соблюсти правильный порядок слоёв при сборке карты. При этом необходимо задать правильное смешение слоёв, так как используемый слой Stamen Toner Background не обладает прозрачностью. Технические сложности реализации смешивания слоёв и надлежащей стилизации ППР

растров в OpenLayers заставляют создать архитектурную прослойку в приложении в виде QGIS-Сервера, осуществляющего отрисовку карты и распространяющего её по WMS (протокол Web Map Service), а в OpenLayers только подключить этот WMS сервис. Для развёртки QGIS-Сервера используется Docker.

QGIS Server (*https://hub.docker.com/r/camptocamp/qgis-server*) – надстройка QGIS Desktop для публикации проектов, реализация WMS и WFS с открытым исходным кодом, OGC API для функций 1.0 (WFS3) и реализация WCS, который, кроме того, реализует расширенные картографические функции для тематического отображения. QGIS Server – это приложение FastCGI/CGI (Common Gateway Interface), написанное на C++ и работающее вместе с веб-сервером (в данном случае Apache). QGIS Server использует функциональность QGIS для анализа и отрисовки карт, поэтому публикуемые карты выглядят так же, как и в настольной ГИС.

Собранное OpenLayers приложение (фронтенд) размещается на базе HTTP-сервера Nginx (*https://nginx.org/*). Nginx – это свободный веб-сервер и обратный прокси-сервер, который используется для обслуживания статических и динамических контентных запросов. Nginx также был развёрнут в виде контейнера Docker (*https://hub.docker.com/_/nginx*).

Чтобы веб-сайт был доступен из глобальной сети, необходимо получить от интернет-провайдера внешний IP-адрес, настроить подключение к DNS и переадресацию на маршрутизаторе.

2. Решение подзадач

2.1. Получение матрицы расстояний с использованием механизма маршрутизации OSRM (FOSSGIS)

Модуль RoutingPy содержит целый перечень методов для разных механизмов маршрутизации. Однако для их использования необходим API-ключ. Это механизм маршрутизации предоставленный Google Maps, а также Graphhopper, Mapbox и другими поставщиками. При этом один, уже развернутый на удалённом, общедоступном сервере, механизм из этого перечня всё же допускает полностью свободное использование – OSRM предоставленный FOSSGIS (*https://www.fossgis.de/*). Плюс данного варианта в

том, что он избавляет от необходимости использования Docker и от работы с геоданными OSM. Здесь необходим лишь Python с Routingy и Pandas и доступ в Интернет. Используем рассматриваемый механизм маршрутизации для составления квадратной матрицы расстояний между населенными пунктами.

На механизм маршрутизации OSRM (FOSSGIS) наложены ограничения, которые не позволяют запрашивать обработку больших данных, но будут рассмотрены способы их обхода. В частности, данный сервис отказывает в построении такой большой матрицы как 422x422, но одиночные маршруты он всё же прокладывает и сборку матрицы можно запрограммировать. Разумеется, такое количество запросов к сервису займет много времени, а также повышает риск блокировки. Делая периодические паузы между обращением к серверу в течение сборки матрицы и отказываясь от прокладки обратных маршрутов, можно беспрепятственно получить все необходимые значения расстояния.

Далее рассматривается программный код на Python.

Импортируются необходимые библиотеки:

Time – встроенный модуль для работы со временем.

```
import pandas as pd
import routingpy as rt
from time import sleep
```

Далее считывается Excel-файл с координатами и численностью населённых пунктов в таблицу Pandas и преобразуется в массив NumPy:

```
coords = pd.read_excel('D:\\Downloads\\ППР_БД_1.xlsx').to_numpy()
coords
```

```
array([['Знаменск ЗАТО (Капустин- Яр -1)', 'Астраханская область', 35,
        27.2, 48.586634, 45.736744],
       ['Озерск ЗАТО', 'Челябинская область ', 89.2, 79.5, 55.763184,
        60.707599],
       ['Снежинск ЗАТО', 'Челябинская область ', 47.9, 50.3,
56.085425,
        60.718064],
       ...,
       ['Актау', 'Мангистауская область', 159.2, 185.894, 43.635379,
        51.169135],
       ['Жанаозен', 'Мангистауская область', 56, 144.706, 43.343266,
        52.865792],
       ['Кзылорда', 'Кызылординская область', 151.8, 286.206,
44.842557,
        65.502545]], dtype=object)
```

Создаётся объект клиента механизма маршрутизации Valhalla и указывается адрес сервиса. Дополнительно передаётся параметр, подтверждающий игнорирование некоторых ошибок. Используется вложенный цикл для построения матрицы расстояний. Обращение к OSRM происходит методом directions(), который прокладывает маршрут между двумя пунктами. Для каждой пары координат ставятся в соответствие остальные пары и заполняется строка матрицы. На следующей строке (паре координат) осуществляется сдвиг на k колонок вправо, чтобы пропустить построение обратных маршрутов. При этом запись данных осуществляется не в виде матрицы расстояний, а в виде списка маршрутов: каждое расстояние с соответствующими пунктами записывается на отдельной строке списка, выпрямляя треугольную матрицу расстояний в вектор-столбец с расстояниями. Попутно на каждой строке записываются время в пути и геометрия маршрута. Выводится информация о процессе работы скрипта.

Из-за большого количества запросов сервер OSRM может принудительно разорвать соединение, в таком случае выдаётся исключение со стороны сервера. В коде содержится обработка таких исключений: полученные на момент появления исключения данные записываются в Excel-файл (на случай полной блокировки), а затем скрипт «засыпает» на 5 минут, после чего вновь запрашивает неполученное расстояние и продолжает построение матрицы. В конце все данные экспортируются в таблицу Excel.

```
client = rt.OSRM('https://routing.openstreetmap.de/routed-car',
skip_api_error=True)
dists = []
k = 0
for p1 in coords.tolist():
    k+=1
    print(k)
    for p2 in coords[k:]:
            try:
                route = client.directions(locations=[p1[-2:][::-
1],p2[-2:][::-1]], profile='car')
            except:
                pd.DataFrame(dists).to_excel('distance matrix.xlsx')
                print('сплю')
                sleep(5*60)
                print('работаю')
                route = client.directions(locations=[p1[-2:][::-
1],p2[-2:][::-1]], profile='car')
            dists.append([*p1, *p2, route.distance, route.duration,
route.geometry])
df = pd.DataFrame(dists)
df.to_excel('distance matrix full.xlsx')
```

При выполнении данного скрипта был сделан: $\frac{422 \cdot 422 - 422}{2} = 88831$ запрос. Время выполнения в целом соответствует суммарному времени всех операций при использовании локальных механизмов маршрутизации – около 6 часов. Данный способ подходит для одноразового использования. При многоразовом использовании локальная развертка механизмов маршрутизации предпочтительнее, так как время, потраченное на нее, становится незначительным в масштабе многочисленности запросов. Для Российско-Белорусской и Российско-Финляндской границ, а также для построения прямоугольной матрицы расстояний между населенными пунктами и узлами сетки данный способ не использовался.

2.2. Получение базы геоданных OSM и извлечение интересующей области

Существует несколько вариантов получения геоданных OSM, пожалуй, самым известным и удобным является загрузка с открытого общедоступного сервера компании Geofabrik (*https://download.geofabrik.de/*). На нём периодически выкладываются обновлённые геоданные OSM, представляющие из себя части, определенные территориальные единицы (части света, страны, области/штаты), от всей базы данных OSM в форматах **.osm.pbf* и **.shp*. Соответственно, начинать можно с представленных **.osm.pbf*.

Первым делом необходимо геокодировать исходные табличные данные. Таблицы добавляются в древо слоёв QGIS и подкладывается XYZ-слой OpenStreetMap с помощью дополнительного QGIS-плагина QuickMapServices. В панели инструментов в группе «Вектор – Создание» выбирается инструмент «Создать точечный слой из таблицы», поле X – колонка, соответствующая долготе, поле Y – колонка, соответствующая широте. После создания точечного слоя сравнение с подложкой OSM позволило убедиться в корректном геокодировании. Полученные географические слои и QGIS проект были сохранены в файл GeoPackage.

К сожалению, интересный для данной работы регион отсутствует на указанном сайте в виде одного файла. Из-за опасений некорректной работы механизмов маршрутизации с объединенным **.osm.pbf* было решено прибегнуть к скачиванию всей базы данных OSM – *planet.osm.pbf*. На официальном сайте Planet OSM (*https://planet.openstreetmap.org/*) имеется возможность свободно скачать регулярно

обновляемую базу данных OSM, в том числе через торрент, занимаемый ею объём дискового пространства составляет около 65 Гб в формате **.osm.pbf*.

После загрузки геоданных OSM было произведено извлечение интересующей области. Это необходимо для сокращения затрат времени и вычислительных ресурсов в процессе построения дорожного графа. C++ библиотека Osmium (*https://osmcode.org/*) работает с данными OSM и имеет в своём арсенале функцию извлечения поднабора данных в формате **.osm.pbf* из исходного **.osm.pbf*. Она доступна в Docker Hub как инструмент командной строки osmium-tool (*https://hub.docker.com/r/stefda/osmium-tool*).

Для извлечения поднабора геоданных OSM применяется следующая команда:

```
docker run -it -w /wkd -v ${pwd}:/wkd stefda/osmium-tool osmium
extract --bbox=17.682871,33.679590,30.404538,42.269466 -o
rus_kz.osm.pbf planet.osm.pbf
```

Данная команда запускает контейнер `stefda/osmium-tool` и тут же производит извлечение `osmium extract`, параметр `bbox` задаёт ограничивающий прямоугольник извлекаемых данных, также имеется возможность указать более сложную границу извлечения, например, ограничивающий полигон/мультиполигон в форматах **.poly/*.geojson* или передать ссылку на отношение OSM, в документации по osmium-tool подробно описаны все возможности. Охват взят с небольшим запасом, чтобы дорожная сеть в дальнейшем не обрезалась ровно по крайнему пункту и у механизма маршрутизации оставалась свобода в выборе маршрута. Важной деталью является отображение файловых директорий контейнера и хоста. Командой `${pwd}` задаётся текущая директория хоста, которая в контейнере имеет путь `/wkd`, и где следует расположить исходный файл `planet.osm.pbf`, здесь же будет сгенерирован извлеченный файл `rus_kz.osm.pbf`. Извлечение объёмных данных занимает время, например, файл объёмом 2,5 Гб из полной базы данных весом 65 Гб извлекался около 1 часа на ПК с 16 Гб оперативной памяти и процессором Intel Core i5-4460. После создания файла на интересующую область можно было приступать к построению дорожного графа механизма маршрутизации.

2.3. Получение матрицы расстояний с использованием механизма маршрутизации Valhalla (Local)

Данный механизм маршрутизации использовался для решения задачи построения квадратной матрицы расстояний между населенными пунктами. Разработчики механизма маршрутизации Valhalla разместили его образ на Docker Hub (*https://hub.docker.com/r/valhalla/valhalla*), кроме того, зачастую пользователи Docker Hub размещают модифицированные образы основанные на базовых, и модифицированные нередко оказываются удобнее и практичнее. На Docker Hub размещен модифицированный образ механизма маршрутизации Valhalla (*https://hub.docker.com/r/gisops/valhalla*). В данной работе был применен именно второй, предоставленный GIS-OPS, как наиболее используемый пользователями образ (более миллиона скачиваний против нескольких сотен тысяч). Скорее всего это связано с большей автоматизацией последнего. Образ предоставленный GIS-OPS основан на официальном образе Valhalla.

Для запуска контейнера gisops/valhalla применяется следующая команда:

```
docker run -dt --name valhalla_gis-ops -p 8002:8002 -v
$PWD/custom_files:/custom_files gisops/valhalla:latest
```

- -dt – открывает псевдотерминал в фоновом режиме.
- --name – задаёт имя контейнеру.
- -p – отображает порт контейнера на порт хоста. В дальнейшем по этому порту будут осуществляться подключения к контейнеру с хоста по localhost.
- -v – отображает файловую директорию контейнера на хост. По адресу, указанному до двоеточия, хранятся служебные файлы контейнера на хосте ($PWD возвращает адрес текущего каталога в Windows PowerShell / Linux bash), в частности исходный файл с геоданными необходимо размещать именно по этому пути. После двоеточия указана та же самая директория, но в том виде, какой она имеет внутри ОС контейнера.
- В конце после пробела указывается имя контейнера с версией после двоеточия.

Таким образом, перед запуском контейнера с Valhalla необходимо разместить в директории, отображаемой на контейнер, файл с геоданными OSM. Когда **osm.pbf* расположен в пустой директории `$PWD/custom_files`, Valhalla автоматически начнет процесс предобработки исходных геоданных, как только запустится контейнер.

Отследить процесс можно в журнале, который сохраняется контейнером Valhalla, например, через приложение Docker с пользовательским интерфейсом (Docker Desktop). Файл `rus_kz.osm.pbf` занимает 4,5 Гб дискового пространства, и его предобработка займёт продолжительное время. Предобработка производилась на ПК с 16 Гб оперативной памяти и процессором Intel Core i5-4460.

Основные продолжительные этапы предобработки:

1) Build the initial graph

- Parsing ways (30 мин.)
- Parsing relations (5 мин.)
- Sorting osm way node references by node id (21 мин.)
- Parsing nodes (21 мин.)
- Sorting osm way node references by way index and node shape index (21 мин.)
- Creating graph edges from ways (14 мин.)
- Sorting graph (10 мин.)
- Reclassifying_V2 link graph edges (18 мин.)
- Reclassifying ferry connection graph edges (16 мин.)
- Building 24342 tiles with 4 threads (96 мин.)

2) Enhancing the initial graph

- Enhancing local graph (38 мин.)
- HierarchyBuilder (12 мин.)
- Creating shortcuts (3 мин.)
- Adding Restrictions (7 мин.)
- Validating, finishing and binning tiles (13 мин.)
- Binning inter-tile edges (3 мин.)
- Cleaning up temporary *.bin files, Hashing files (8 мин.)

Finished tarring 28157 tiles

Итого 336 мин.

По итогу предобработки в директории *./custom_files* появились новые данные:

- valhalla_tiles.tar – результат предобработки, архив с тайлами дорожного графа Valhalla и индексом, именно эти данные используются для маршрутизации.
- Папка valhalla_tiles – неархивированные тайлы дорожного графа, при очередном запуске контейнера происходит их упаковка в архив (при

условии, что переменная окружения контейнера BUILD_TAR = True), поэтому при наличии valhalla_tiles.tar данную папку можно удалить.

- valhalla.json – конфигурационный файл механизма маршрутизации, важно определить в нём допуски на различные операции.
- duplicateways.txt
- file_hashes.txt
- исходный файл *rus_kz.osm.pbf*

Если при запуске контейнера переменная среды *use_tiles_ignore_pbf* имела метку *False*, то, при изменении, добавлении или удалении файлов **osm.pbf* и перезапуске этого же контейнера предобработка (перестройка тайлов дорожного графа) начнётся заново. Насколько удалось установить, происходит именно полная перестройка, а не, достройка на основании новых данных в дополнение к старым.

Далее необходимо отредактировать конфигурационный файл *valhalla.json*, в частности значения ключа *service_limits* – это список допусков различных профилей и их функций в Valhalla, все они, в особенности относящиеся к *max_matrix_distance* и *max_matrix_location_pairs*, были увеличены, чтобы в дальнейшем без ограничений запрашивать построение матрицы расстояний размером 422x422 и получать все расстояния в ней. После сохранения конфигурационного файла контейнер следует перезапустить.

Строка журнала контейнера `INFO: Found config file. Starting valhalla service!` сообщает об успешном запуске сервиса. Ниже приведен программный код на Python осуществляющий проверку работоспособности сервера маршрутизации посредством выполнения тестового запроса и построение матрицы расстояний между населенными пунктами.

Импортируются необходимые библиотеки:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import routingpy as rt
```

Далее считывается Excel-файл с координатами и численностью населённых пунктов в таблицу Pandas и преобразуется в массив NumPy:

```
df = pd.read_excel('BY_FI/FI.xlsx', sheet_name='RUS-
FIN_1+2_ПОРЯДОК_FINAL').to_numpy()
df
array([['Мурманская область',
        'Городской округ город Кировск с подведомственной территорией',
        'Кировск', ..., 28832, 67.615, 33.6636111111111],
```

```
       ['Мурманская область', 'Городской округ ЗАТО Александровск',
        'Полярный', ..., 15663, 69.1988888888889, 33.4508333333333],
       ['Мурманская область', 'Городской округ город Мурманск',
        'Мурманск', ..., 298096, 68.9694444444444, 33.0744444444444],
       ...,
       ['Finland', 'North Ostrobothnia', 'Kalajoki', ..., 12549,
        64.252469, 23.919127],
       ['Finland', 'Uusimaa', 'Kerava', ..., 35442, 60.405365, 25.101696],
       ['Finland', 'Uusimaa', 'Kerava', ..., 35442, 60.405365, 25.101696]],
      dtype=object)
```

Создаётся массив NumPy с координатами населённых пунктов, при этом меняются местами долгота и широта. Координаты на вход для запросов маршрутизации ожидаются в порядке долгота (X) / широта (Y):

```
p = df[:,:5:-1]
p
```

```
array([[33.6636111111111, 67.615],
       [33.4508333333333, 69.1988888888889],
       [33.0744444444444, 68.9694444444444],
       [39.4872222222222, 68.0575],
       [32.9358333333333, 67.9383333333333],
       [33.0261111111111, 68.8786111111111],
       [33.315, 69.2488888888889],
       [33.4177777777778, 69.0763888888889],
       [32.4975, 67.3727777777778],
       [33.2380555555556, 69.1919444444444],
       [32.4141666666667, 67.1566666666667],
       [33.2669444444444, 68.1419444444444],
       [30.8202777777778, 69.4261111111111],
       [32.45, 69.4002777777778],
       ...
```

Создаётся объект клиента механизма маршрутизации Valhalla и указывается адрес сервиса. В данном случае сервис работает в контейнере и отображается на указанный ранее порт хоста, поэтому его адрес: localhost:8002:

```
client = rt.Valhalla('http://localhost:8002', timeout=3000)
```

Осуществляется тестирование работы сервиса. Запрашивается маршрут между двумя пунктами методом directions() объекта клиента, при этом указывается профиль (автомобиль). Метод возвращает сам маршрут в виде массива точек, длину в метрах и

длительность маршрута в секундах (единицы измерения можно поменять, указав явно параметр units). Из вывода понятно, что длина соответствует действительности:

```
r = client.directions(locations=p[1:3].tolist(), profile='auto')
r.distance
```
```
62021
```

Также схема маршрута подтверждает корректность работы сервиса (рис. 5):

```
plt.plot(*np.array(r.geometry).T)
```
```
[<matplotlib.lines.Line2D at 0x2357103e460>]
```

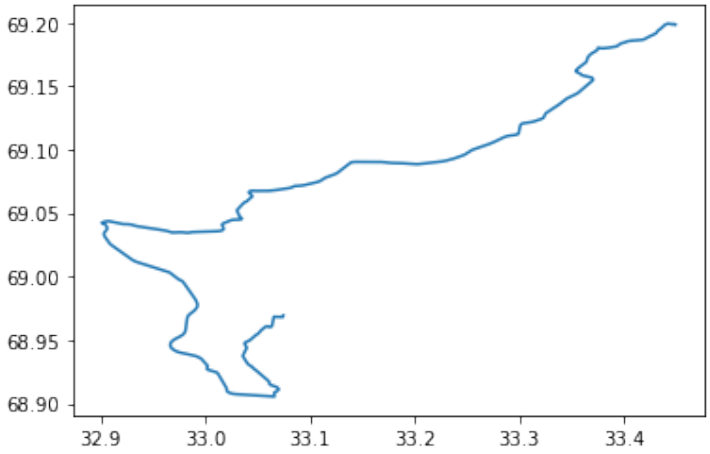

Рисунок 5. Схема тестового маршрута, проложенного механизмом маршрутизации Valhalla.

Далее строится матрица расстояний между населёнными пунктами из имеющегося массива. Здесь параметр locations принимает полный массив координат в формате обычного списка Python. Данная операция требовательна к ресурсам сервера, особенно, с ростом числа пунктов в массиве. Для Российско-Казахстанской границы их количество составило 422, что соответствует 422*422-422 = 177662 значениям расстояния, такая матрица строилась около 15 минут.

Особое внимание стоит уделять возможному переполнению ОЗУ во время построения матрицы. В случае переполнения ОЗУ используется файл подкачки, однако предварительно нужно убедиться, что Docker выделяет достаточно памяти для него, на Windows это можно регулировать через настройку WSL. В данном примере при построении матрицы контейнер по непонятным причинам прервал работу (автоматическая команда: /valhalla/scripts/run.sh: line 9: 28399 Killed sudo -E $1), скорее всего это было связано как раз с переполнением файла подкачки. В файле *.wslconfig* с настройками WSL были увеличены максимальные размеры файла подкачки и после повторного запроса матрица построилась.

После выполнения запроса к механизму маршрутизации создаётся объект с атрибутами distances и durations, в которых хранятся собственно матрицы расстояний и времени. Далее эти матрицы преобразуются в таблицы Pandas и экспортируются в таблицы Excel. Порядок того, какое значение к какому пункту относится задаётся входным массивом координат (если не указаны параметры sources и destinations), в данном примере не происходило никаких сортировок и перемешиваний, поэтому порядок в матрицах соответствует порядку в исходной таблице Excel, что позволяет добавить далее названия населённых пунктов для строк и столбцов.

```
dm = client.matrix(locations=np.delete(p,3,0).tolist(),
profile='auto')
dmd = pd.DataFrame(dm.distances)
dmt = pd.DataFrame(dm.durations)
```

```
dmd.to_excel('dmd.xlsx')
dmt.to_excel('dmt.xlsx')
```

Фрагмент результата работы функции Distance Matrix механизма маршрутизации Valhalla приведен в табл. 1.

city	Новгород	Боровичи	Старая Русса	Валдай	Малая Вишера
Новгород	0	196018	97063	139978	90567
Боровичи	197028	0	203567	79480	149897
Старая Русса	96888	203099	0	166035	167719
Валдай	140199	79051	166023	0	178090
Малая Вишера	90558	149452	167958	178381	0
Окуловка	159913	39773	166452	69680	145358
Пестово	316785	120234	323324	199237	269259
Сольцы	77631	282584	78923	217609	177133
Холм	201107	306944	105323	209989	271564
Чудово	74848	203389	184278	194701	70714

Таблица 1. Фрагмент матрицы расстояний по дорожной сети.

С помощью механизма маршрутизации Valhalla удалось получить квадратную матрицу расстояний между населенными пунктами (422x422), однако построить с его помощью прямоугольную матрицу (с sources и destinations) между населенными пунктами и узлами сетки не удалось: прямоугольная матрица вычисляется ощутимо дольше квадратной, не говоря уже о случае с миллионами расстояний.

2.4. Получение матрицы расстояний с использованием механизма маршрутизации OSRM (Local)

В Docker Hub был обнаружен образ контейнера популярного механизма маршрутизации OSRM для локальной развёртки (*https://hub.docker.com/r/osrm/osrm-backend*). Он был использован для решения подзадачи построения прямоугольной матрицы между населенными пунктами и узлами сетки.

Основные функции механизма маршрутизации OSRM:

- Nearest – привязывает координаты к дорожной сети, возвращая ближайшие совпадения.
- Route – находит самый быстрый маршрут между координатами.
- Table – вычисляет продолжительность или расстояние по самому быстрому маршруту между всеми парами заданных координат.
- Match – привязка зашумленных GPS-трасс к дорожной сети наиболее правдоподобным способом.
- Trip - решает проблему коммивояжера, используя жадный алгоритм.
- Tile - генерирует векторные тайлы Mapbox с внутренними метаданными маршрутизации.

Подготовка OSRM к работе также осуществляется командами Docker, но в отличие от Valhalla осуществляется в несколько отдельных команд. С помощью них на основе образа OSRM создаётся новый контейнер, в котором запускается заданный процесс. По окончании процесса контейнер выключается и удаляется, и для другого, заданного командой процесса, создаётся другой контейнер. При этом рабочая директория всех контейнеров отображается (подключается) на одну и ту же директорию хоста, и, таким образом, каждый контейнер выполняет определенные действия с данными, находящимися в этой директории. Данные, в свою очередь, создаются и преобразуются во время выполнения процессов разных контейнеров и в итоге принимают необходимый для осуществления маршрутизации вид. На главной странице OSRM в Docker Hub приведены команды и их последовательность. Для начала, разумеется, необходимо иметь исходные геоданные OSM, процесс по получению и извлечению геоданных OSM был описан раннее. Время на предобработку в целом соответствует образу механизма маршрутизации Valhalla из соответствующего раздела данной работы.

Первым делом производится первичное извлечение из исходного файла *rus_kz.osm.pbf* объёмом 4,5 ГБ при помощи следующей команды:

```
docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-extract -p /opt/car.lua /data/rus_kz.osm.pbf
```

- -t – открывает псевдотерминал.
- -v – отображает файловую директорию контейнера на хост. По адресу, указанному до двоеточия, хранятся служебные файлы контейнера на хосте ($PWD возвращает адрес текущего каталога в Windows PowerShell / Linux bash), в частности исходный файл с геоданными необходимо размещать именно по этому пути. После двоеточия указана та же самая директория, но в том виде, какой она имеет внутри ОС контейнера.

Команда `osrm-extract` и далее относится уже к происходящему внутри контейнера. Флаг `-p` указывает профиль механизма маршрутизации. Можно создавать пользовательские профили, а в данном случае используется профиль по умолчанию для маршрутов на автомобиле. В конце указывается путь к исходным геоданным внутри контейнера.

Данный этап состоит из следующих продолжительных подэтапов, в скобках указана продолжительность в секундах:

- Parse relations (42)
- Parse ways and nodes (1116)
- Sorting all nodes (117)
- Confirming/Writing used nodes (47)
- Sorting edges by start (283)
- Setting start coords (272)
- Sorting edges by target (60)
- Computing edge weights (408)
- Sorting edges by renumbered start (134)
- Writing used edges (46)
- Generating edge-expanded graph representation (468)
- Writing nodes for nodes-based and edges-based graphs (40)
- Generating edge expanded nodes (415)
- Generating edge-expanded edges (1055)
- Generating guidance turns (565)
- Saving edge-based node weights to file (43)
- Computing strictly connected components (203)
- R-tree construction (119)

– Writing edge-based-graph edges (61)

Всего извлечение заняло 1 час 55 минут 17 секунд по времени и 14053 Мб оперативной памяти (в пиковой нагрузке) на процессоре Intel Core i5-4460 в 3 потоках.

Далее следует второй этап предобработки. Здесь предлагается либо последовательный запуск двух команд `osrm-partition` и `osrm-customize`, которые подготавливают данные для дальнейшей маршрутизации с использованием алгоритма MLD (многоуровневый Дейкстра), ускоряющего маршрутизацию, либо единственной `osrm-contract`, реализующей метод иерархий сжатия (Contraction Hierarchies), который также ускоряет маршрутизацию. Был использован второй вариант, так как он лучше подходит для построения больших матриц расстояний, как отмечено в руководстве по запуску данного контейнера. Метод иерархий сжатия эффективен на графах с небольшим количеством вершин и ребер. В процессе предобработки для дальнейшего использования этого метода дорожный граф существенно упрощается: удаляются избыточные ребра и вершины, а остаются только наиболее значимые. Эта процедура позволяет быстрее вычислять кратчайшие пути и, следовательно, быстрее строить матрицы расстояний.

На втором этапе запускается следующая команда:

```
docker run -t -v "${PWD}:/data" osrm/osrm-backend osrm-contract /data/rus_kz.osrm
```

Следует обратить внимание, что для выполнения данной команды необходим файл, созданный предыдущей командой, он сгенерировался рядом с исходными геоданными. Команда выполнялась 3 часа 36 минут 5 секунд с теми же аппаратными характеристиками.

В итоге запускается контейнер, принимающий запросы на маршрутизацию на основе подготовленных данных, внутри контейнера это делает команда `osrm-routed`, при этом снимаются ограничения на функциональные возможности движка (внутриконтейнерные флаги `--max-…` устанавливаются в `-1`):

```
docker run -t -i -p 5001:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --max-viaroute-size -1 --max-trip-size -1 --max-table-size -1 --max-matching-size -1 --max-nearest-size -1 /data/rus_kz.osrm
```

- `-i` – интерактивный режим.
- `-p` – отображает порт контейнера на порт хоста. В дальнейшем по этому порту будут осуществляться подключения к контейнеру с хоста по localhost.

Сообщение «`running and waiting for requests`» говорит об успешном запуске сервера механизма маршрутизации OSRM, сервер в ожидании запросов. Данный контейнер можно использовать многократно – останавливая и перезапуская, не создавая каждый раз новый.

Ниже приведен программный код на Python осуществляющий проверку работоспособности сервера посредством выполнения тестового запроса и построение матрицы расстояний между населенными пунктами и узлами сетки.

Импортируются необходимые библиотеки:

```
import pyproj
import numpy as np
import pandas as pd
import rasterio
import folium
import routingpy as rt
```

При возникновении проблемы с установкой Rasterio следует воспользоваться интерпретатором Python и менеджером пакетов pip входящим в состав дистрибутива QGIS, так как QGIS уже содержит в своем дистрибутиве библиотеку GDAL, при установке которой в общем случае и возникает проблема.

Задаются необходимые входные параметры регулярной сетки – шаг и охват:

```
res = 1e4

left = -215
right = 209
bottom = 133
top = 385
```

После расчёта потенциала поля расселения каждому узлу сетки будет соответствовать его значение. Ввиду регулярности сетки выглядит целесообразным записать результаты в геопривязанный растр, где её узлы будут центрами пикселей. Разрешение растра – это размер пикселя, и он равен шагу сетки, только не от узла до соседнего, а от середины отрезка между двумя узлами-соседями до соседней такой середины (рис. 6). Значение пикселя растра – значение потенциала поля расселения в узле, к которому относится пиксель.

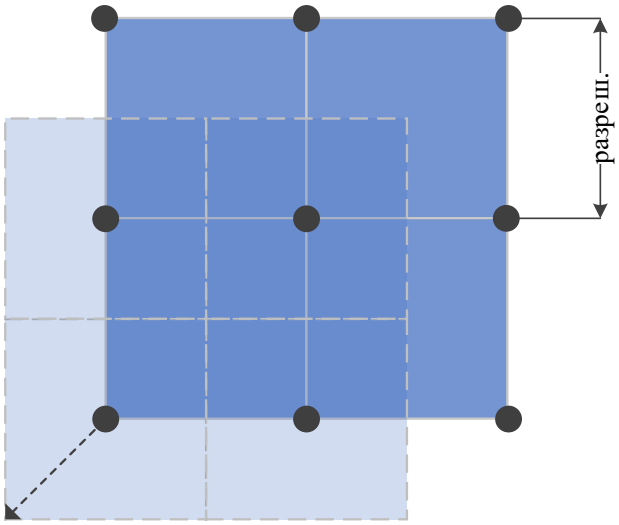

Рисунок 6. Соответствие пикселей и узлов сетки.

При создании геопривязанного растра в Rasterio используются пиксельные координаты, то есть номера строк и столбцов. Для сохранения его в определенной системе координат необходимо задать матрицу аффинного преобразования, которая сдвинет и отмасштабирует растр к этой СК. В следующем блоке кода задаётся матрица аффинного преобразования со сдвигом в угол сетки, но за вычетом половины разрешения, что делает угловой (и, соответственно, любой) узел сетки не углом, а центром пикселя (рис. 6). Затем в матрицу добавляется масштабный коэффициент. Полученное аффинное преобразование будет использовано в конце всей программы при сохранении растра в следующем разделе. Далее создается сетка (рис. 7) по заданным ранее параметрам и сохраняется её форма (количества элементов по каждому измерению массива). Размеры сетки получаются 253x425:

```
rast_transform = rasterio.transform.Affine.translation(res*left -
res/2, res*bottom - res/2) * rasterio.transform.Affine.scale(res,
res)
grid = res*np.mgrid[bottom:top+1, left:right+1]
grid_shape = grid.shape
```

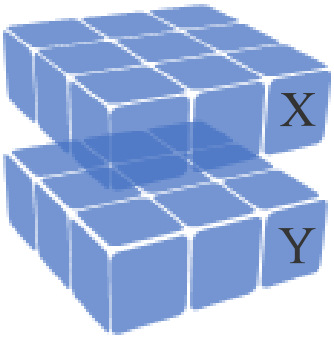

Рисунок 7. Утрированная (3x3) форма массива координат узлов сетки.

Значения координат сетки на самом деле являются значениями в определенной спроецированной системе координат, при этом механизм маршрутизации ожидает точки в широте/долготе. Однако, если создавать сетку сразу в широте/долготе, то на больших пространствах при рассмотрении на сфере сильно исказится её регулярность, что в дальнейшем негативно повлияет на репрезентативность итогового потенциала поля расселения.

Далее в коде задаются параметры системы координат сетки и определяется преобразователь в систему координат с широтой/долготой:

```
source_crs = pyproj.CRS.from_proj4("+proj=aea +lat_0=30 +lon_0=64
+lat_1=47 +lat_2=58 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
+type=crs")
crs_transformer = pyproj.Transformer.from_crs(source_crs,
'epsg:4326')
```

Затем сетка перепроецируется в систему координат с широтой/долготой, при этом форма выходного массива сохраняется. Механизм маршрутизации ожидает точки обычным списком пар координат, поэтому массив с перепроецированной сеткой выпрямляется в двумерный и сохраняется как вложенный список пар координат. Также сохраняется количество узлов:

```
grid = np.array(crs_transformer.transform(grid[1], grid[0]))[::-1]
srs = grid.reshape(2,-1,order='C').T.tolist()
srs_l = len(srs)
```

Далее считывается Excel-файл с координатами и численностью населённых пунктов в таблицу Pandas. Создаются отдельные переменные (векторы-строки NumPy) с численностями населения в населенных пунктах в 1989 и 2017 году. Координаты преобразуются в список destinations (dst) в порядке долгота, широта, как того ожидают методы RoutingPy. Длина исходного списка dst записывается специально для сохранения информации о позициях destinations в общем списке (как и ранее с srs). Список координат узлов сетки (источников) расширяется населенными пунктами (назначениями), так как методу запроса матрицы необходимо подавать один общий список и его индексы на sources и destinations:

```
dst = pd.read_excel('KZ/ППР_БД_1.xlsx')
popul_1989 = dst['Population size, 1989, thousands
people'].to_numpy()[np.newaxis,:]
popul_2017 = dst['Population size,  2017, thousands
people'].to_numpy()[np.newaxis,:]
dst = dst.values[:,:3:-1].tolist()
dst_l = len(dst)
srs.extend(dst)
```

Создаётся объект клиента механизма маршрутизации OSRM и указывается адрес сервиса. В данном случае сервис работает в локальном контейнере и отображается на указанный ранее порт хоста, поэтому его адрес localhost:5000. Максимальное время на выполнение запросов к сервису задается равным одному часу (3600 сек.):

```
client = rt.OSRM('http://localhost:5000', timeout=3600)
```

Работоспособность сервиса тестируется, посредством запроса маршрута между двумя пунктами методом directions() объекта клиента. Метод делает HTTP-запрос функции Route сервера OSRM. Он возвращает сам маршрут в виде списка точек, длину в метрах и длительность маршрута в секундах. Полученная геометрия маршрута записывается в массив. По выводу функции print ниже понятно, что длина соответствует действительности:

```
r = client.directions(locations=[dst[64], dst[157]])
route = np.array(r.geometry).T[::-1]
print('Из {},{} в {},{} ехать {} м за {} сек'.format(*dst[64],
*dst[157], r.distance, r.duration))
```

```
Из 63.28571,55.228726 в 64.183351,53.795151 ехать 323155 м за 20847
сек
```

С помощью Folium создается Leaflet-карта с подложкой Stamen Toner (блокноты Jupyter поддерживают вывод результатов в HTML). `folium.PolyLine` предлагает простой способ обращения к функции Leaflet по построению векторной полилинии на карте. Таким образом выводится схема маршрута, подтверждающая корректность работы сервиса (рис. 8):

```
fig = folium.Figure(width=1200, height=600)
map = folium.Map(tiles="Stamen Toner").add_to(fig)
folium.PolyLine(route.T.tolist(), tooltip="Route").add_to(map)
map.fit_bounds(map.get_bounds(), padding=(5, 5))
map
```

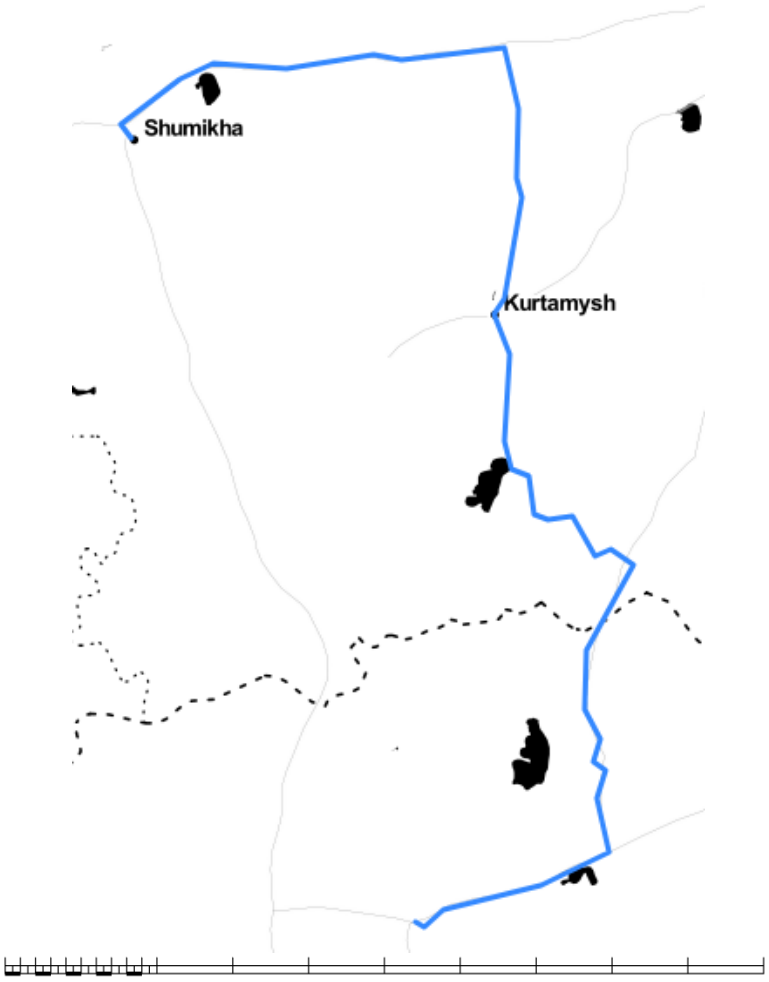

Рисунок 8. Тестовый маршрут, проложенный механизмом OSRM.

Итак, строится прямоугольная матрица расстояний от каждого узла сетки до каждого населенного пункта. Метод объекта клиента matrix() делает HTTP-запрос функции Table сервера OSRM, при этом передается список всех пунктов и указываются индексы исходных и целевых. Для формирования индексов используются заранее сохраненные длины списков. Полученная матрица записывается в массив NumPy. С 16 Гб оперативной памяти, на процессоре Intel Core i5-4460 в 3 потоках запрос выполнялся около 5 минут:

```
dm = client.matrix(locations=srs, sources=list(range(srs_l)),
destinations=list(range(srs_l, srs_l + dst_l)),
annotations=('distance',))
dmd = np.array(dm.distances)

print(dmd, f'\n\nФорма матрицы: {dmd.shape[0]} x {dmd.shape[1]}')
```

```
[[1775113.8 3332430.2 3368652.4 ... 2831070.4 2815088.8 4004044. ]
 [1758603.8 3315920.2 3352142.2 ... 2814560.4 2798578.8 3987534. ]
 [1734161.6 3291478.  3327700.  ... 2790118.4 2774136.4 3963091.6]
 ...
```

```
 [4659214.  2991765.2 3009416.8 ... 5350880.  5334898.  3702155.6]
 [4659214.  2991765.2 3009416.8 ... 5350880.  5334898.  3702155.6]
 [4659214.  2991765.2 3009416.8 ... 5350880.  5334898.  3702155.6]]

Форма матрицы: 107525 x 422
```

С помощью локально установленного механизма маршрутизации OSRM удалось решить подзадачу по получению прямоугольной матрицы расстояний между населенными пунктами и узлами регулярной сетки. Аналогичным образом была получена матрица расстояний для трансграничных регионов Россия-Финляндия и Россия-Белоруссия. Матрицу можно экспортировать в Excel, однако сами по себе расстояния не представляют особого интереса, а экспортированная таблица займёт большой (для таблицы Excel) объем дискового пространства, что приведет к повышенному времени отклика при работе с ней в Excel или в настольных ГИС. Лучше продолжить обработку матрицы в Python и непосредственно рассчитать потенциал поля расселения.

2.5. Вычисление потенциала поля расселения и создание растра вычисленных значений

Благодаря функциональности NumPy расчёт потенциала поля расселения расположится всего на нескольких строчках программного кода и не займет много времени выполнения. Однако прежде, чем рассчитывать собственно ППР, необходимо внести поправки в расстояния в матрице.

Дело в том, что список всех входных точек не привязан к дорожному графу OSRM, то есть каждая точка расположена в пустом пространстве на расстоянии от графа. OSRM привязывает точки, как бы притягивает их на ближайшие места графа, и результаты выдает уже для привязанных положений. Поэтому изначальная матрица расстояний не учитывает концевые отрезки маршрутов, а они могут оказаться внушительными, если по близости совсем нет дорог. Такие «домеры» (прямолинейные расстояния между изначальной и привязанной к графу точкой) всё равно содержатся в ответе на запрос, но отдельно от основной матрицы, остается только прибавить их к ней.

Отметим недостаток, что если расстояние до привязки слишком большое по сравнению с основным расстоянием по графу, то результат сильно искажается: расстояния между пунктами преуменьшаются. Преуменьшенное расстояние

преувеличит итоговое значение поля, показав больший потенциал, что в данном случае категорически неверно. Домеры в отсутствие дорог будут скорее максимально извилистыми, чем прямолинейными. Для учета такого нюанса необходимо вводить штрафные коэффициенты за большое значение прямолинейного «домера». Однако, это остаётся за рамками работы.

Далее продолжается рассмотрение программного кода из предыдущего раздела.

Из необработанного ответа на запрос берется список destinations, в нем содержатся словари с привязанной точкой и расстоянием до нее от исходного населенного пункта. Создается массив (вектор) NumPy содержащий только значения расстояния:

```
dst_new = pd.DataFrame(dm.raw['destinations'])
dst_new = dst_new['distance'].to_numpy()
```

Такие же операции производятся и со списком sources, содержащим словари с привязанной точкой и расстоянием до нее от исходного узла сетки:

```
srs_new = pd.DataFrame(dm.raw['sources'])
srs_new = srs_new['distance'].to_numpy()
```

Следующая операция выполняет внесение поправок в матрицу расстояний. Операция производится согласно правилам транслирования (broadcasting) NumPy: берутся векторы с расстояниями от исходных и целевых точек, при этом один из них транспонируется в строку, а другой определяется как столбец, затем вектор-столбец повторяется по горизонтали столько раз сколько столбцов в векторе-строке, а вектор-строка по вертикали – сколько строк в векторе-столбце, таким образом получается матрица формой равная матрице расстояний, значения матриц будут соответственными и поправка вносится поэлементным сложением (рис. 9). Фактически при транслировании данные не копируются, а работают низкоуровневые эффективные алгоритмы.

```
dmd += srs_new[:,np.newaxis] + dst_new[np.newaxis,:]
```

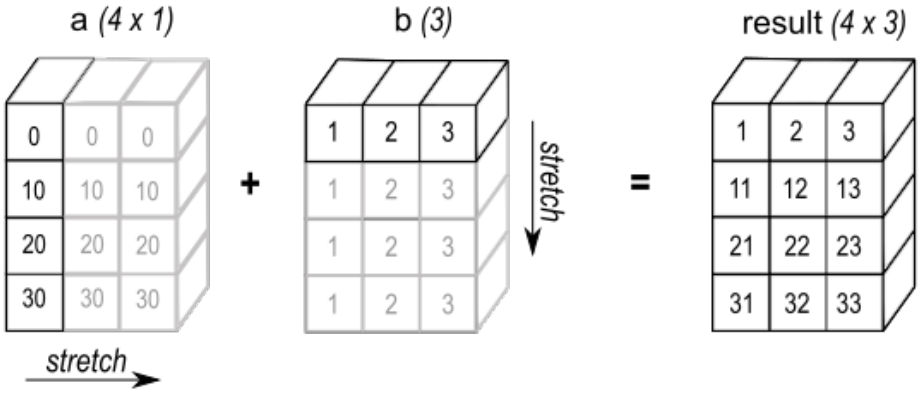

Рисунок 9. Транслирование в NumPy.
(https://numpy.org/doc/stable/user/basics.broadcasting.html, Figure 4)

Далее реализуется формула вычисления потенциала поля расселения. Применяются избирательные удвоения, а также сбросы особо больших расстояний, дабы ослабить влияние далеко расположенных населенных пунктов на итоговое значение ППР. Внутри квадратных скобок находятся булевы маски [7, с. 104-107], и умножение производится только для значений «Истина».

Наконец, вычисляется ППР: вектор-строка с численностями населения населенных пунктов приводится к единицам человек, деление производится также сначала по правилам транслирования (вектор-строка с численностью населения повторяется по вертикали столько раз сколько строк в матрице расстояний), а затем поэлементно. Теперь остаётся свернуть матрицу в вектор-столбец, просуммировав значения населенных пунктов для каждого узла сетки, при этом используется функция nansum игнорирующая значения nan при суммировании.

Полученный вектор-столбец необходимо привести к размеру исходной сетки (253x425), чтобы восстановить соответствие между узлом и относящимся к нему значением ППР. Порядок пар координат в списке узлов сетки сохранялся как при передаче в функцию построения матрицы расстояний, так и на выходе из нее (строки матрицы расстояний расположены ровно по порядку строк списка узлов сетки). Соответственно, порядок значений ППР в векторе-столбце соответствует порядку пар координат в исходном списке узлов сетки. Исходный список узлов сетки был получен выпрямлением массива Numpy mgrid, поэтому обратная операция reshape восстановит форму вектора-столбца ППР в соответствие узлам сетки по сохраненному ранее значению формы (рис. 10).

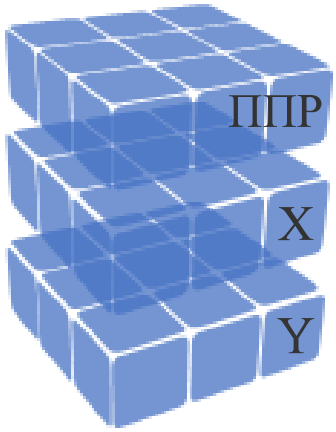

Рисунок 10. Утрированная (3x3) форма массива координат узлов сетки со значениями ППР.

Вычисление ППР выполняется для двух вариантов численности населения одного и того же населенного пункта: на 2017 и на 1989 годы, но с одним вариантом матрицы расстояний. Соответственно, получается два варианта ППР по годам.

```
dmd[(dmd > 300000)] = np.nan
dmd[(dmd > 50000) & (dmd <= 300000)] *= 2

SFP_1989 = np.nansum(popul_1989*1000/dmd, axis=1)
SFP_1989 = SFP_1989.reshape(grid_shape[1:])

SFP_2017 = np.nansum(popul_2017*1000/dmd, axis=1)
SFP_2017 = SFP_2017.reshape(grid_shape[1:])
```

В итоге создается файл GeoTIFF для записи значений потенциала поля расселения в многоканальный растр по годам. Этот растровый формат поддерживает значения с плавающей точкой. Размеры растра в пикселях задаются согласно сохраненной форме сетки (253x425). Задаётся определенная ранее система координат. Нулевые значения ППР соответствуют значению растра «nodata», при такой настройке отображение растра будет прозрачным в пикселях, в которых отсутствует (равен 0) потенциал поля расселения. При создании растра не учитывается сетка узлов, записывается только массив со значениями ППР, имеющий форму равную заданному размеру растра в пикселях. Вместо сетки узлов для сдвига и масштабирования пиксельной системы координат в заданную используется раннее определенная матрица аффинного преобразования. После записи массива ППР каналам задаётся соответствующее название. В конце созданный файл (растр) закрывается.

```
rast = rasterio.open('SFP_KZ_RUS_bnd.tif', 'w', driver='GTiff',
height=grid_shape[1], width=grid_shape[2], count=2, dtype='float32',
crs=source_crs, transform=rast_transform, nodata=0,
photometric='MINISBLACK')
rast.write(SFP_1989, 1)
rast.set_band_description(1, 'SFP_1989')
```

```
rast.write(SFP_2017, 2)
rast.set_band_description(2, 'SFP_2017')
rast.close()
```

Аналогичным образом был рассчитан потенциал поля расселения и получен растр его значений для трансграничных регионов Россия-Финляндия и Россия-Белоруссия на 2021 и 1989 годы.

2.6. Создание веб-сайта для картографической визуализации потенциала поля расселения

В данном разделе рассматриваются этапы создания и архитектура веб-сайта для картографической визуализации потенциала поля расселения. Создается проект QGIS и в него добавляются растры ППР и подложки Stamen Toner Background и Stamen Toner Labels. Затем задаётся порядок отрисовки и стилизация слоям.

Подложки отрисовываются поверх растров ППР, при этом на самом верху располагаются подписи (Stamen Toner Labels), а под ними – картографическая основа Stamen Toner Background. В настройках стиля слоя Stamen Toner Background режим смешивания выбирается «Умножение». Таким образом белые цвета принимают цвета нижележащего слоя, а чёрные полностью поглощают цвета нижележащего слоя и остаются черными. В данном случае такая стилизация особенно эффективна, так как чёрным цветом показываются водные пространства, которые скрывают потенциально ошибочные пиксели со значением ППР на водных объектах, а на месте белых цветов значения ППР, наоборот, как и следует, отображаются в полной мере без цветовых искажений. Все растры ППР расположены под слоями Stamen Toner и их взаимный порядок не важен, так как на веб-сайте они отображаются только по одному.

Создаются отдельные слои для каждого канала каждого растра ППР и задаётся их стилизация (рис. 11). Полученные четыре растровых слоя стилизуются как одноканальные псевдоцветные изображения. Для растра, представляющего трансграничный регион Россия-Казахстан, выбран цветовой градиент «PuBuGn», а для растра Россия-Белоруссия/Россия-Финляндия – «YlGnBu». Интерполяция между цветовыми ступенями выбрана линейная. Растяжка градиента осуществляется между фактическим минимумом и максимумом растров на 15 ступеней. Ступени разделены квантилями (каждый цвет занимает одинаковое количество пикселей). Квантили наиболее точно представляют растр потенциала поля расселения.

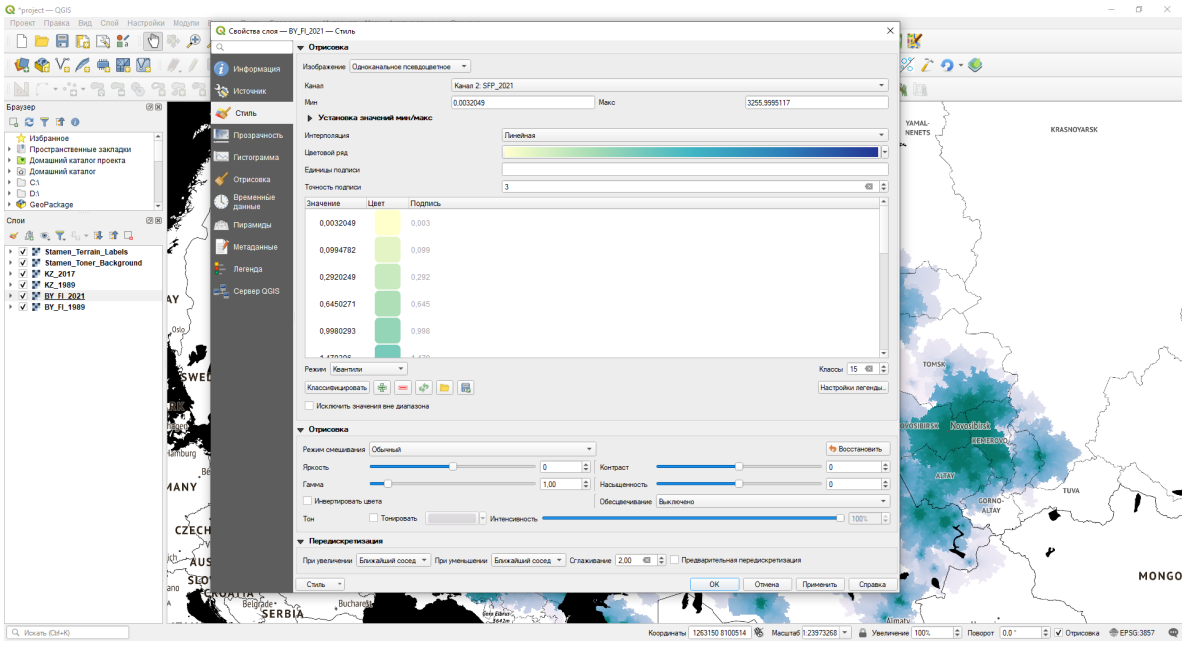

Рисунок 11. Стилизация растра ППР в QGIS.

Проект сохраняется под именем *project.qgs*, и, в дальнейшем, директория, где он располагается, отображается в Docker-контейнер с QGIS-Сервером.

Далее рассматривается настройка и создание Docker-контейнеров с QGIS Server и Nginx. Docker-контейнеры QGIS Server и Nginx взаимодействуют. Для создания взаимодействующих контейнеров используется Docker-Compose. Он позволяет определять и запускать многоконтейнерные Docker-приложения. С помощью Docker Compose описываются все компоненты приложения в файле YAML и запускаются одной командой. Взаимодействие осуществляется внутри изолированной для данной группы контейнеров сети, созданной Docker-Compose.

Команда `docker-compose up` создаст и запустит все контейнеры, определенные в файле *docker-compose.yml*, и свяжет их между собой в соответствии с настройками. Далее приведено содержимое файла *docker-compose.yml*:

```
version: '3.9'

services:
  web:
    image: nginx
    ports:
      - '80:80'
    volumes:
      - 'D:/ol-app/dist:/usr/share/nginx/html'
  server:
    image: camptocamp/qgis-server
    volumes:
      - 'D:/YandexDisk/4 курс/Диплом/QGIS_Server:/etc/qgisserver'
    expose:
```

```
      - '80'
    environment:
      - QGIS_PROJECT_FILE=/etc/qgisserver/project.qgs
```

Также необходимо настроить Nginx как прокси-сервер, перенаправляющий WMS-запросы на QGIS Server в изолированную сеть. Был обновлен конфигурационный файл, а именно блок кода, отвечающий за запросы к корневому URL веб-сервера. Обновленный блок кода выглядит следующим образом:

```
        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
            if ( $arg_SERVICE = WMS) {
                proxy_pass http://server;
            }
        }
```

Внутри изолированной сети адресами контейнеров являются их имена, заданные в файле *docker-compose.yml*. Команда `proxy_pass http://server` перенаправляет WMS-запросы на 80-ый порт (на веб-сервер Apache) контейнера QGIS Server с именем server.

Для разработки веб-сайта необходимо установить Node.js, либо воспользоваться Docker-контейнером Node.js. Далее создается шаблон приложения OpenLayers (с дополнительными модулями OL-Cesium и OL-LayerSwitcher) с помощью менеджера пакетов NPM посредством следующих команд в командной строке:

```
npm create ol-app ol-app
cd ol-app
npm install cesium vite-plugin-cesium vite -D
npm install olcs
npm install ol-layerswitcher
```

Создаётся директория `ol-app`, включающая стартовые шаблонные файлы разработки (.html, .css, .js), файлы конфигураций и необходимые зависимости. Отладка и сборка готового для хостинга приложения осуществляется посредством Vite. Итоговое содержимое файлов разработки приведено в приложении к данной работе.

Функциональность разработанного веб-сайта:

- Просмотр мультимасштабной интерактивной карты потенциала поля расселения трансграничных регионов России и её легенды
- Переключение между растрами ППР
- Получение значения ППР в заданном пикселе растра по карте

– Просмотр в 3D-режиме на глобусе

По окончании разработки приложение было собрано посредством следующей команды:

```
npm run build
```

Создаётся директория `my-app/dist`, содержащая файлы сборки (.html, .css, .js), которые размещаются в дальнейшем как статический контент на веб-сервере Nginx.

Далее рассматривается публикация веб-сервера Nginx, осуществляющего хостинг веб-сайта в глобальной сети. После получения от интернет-провайдера внешнего IP-адреса (в данном случае услуга была бесплатной) появляется возможность сделать локальную сеть доступной из глобальной сети. Подключения приходят на маршрутизатор, который осуществляет управление подключениями, перенаправляя их по адресам в локальной сети, если они были опубликованы. Кроме того, на маршрутизаторе работает DHCP-сервер, который динамически присваивает IP-адреса устройствам в локальной сети. Для публикации необходим постоянный IP-адрес, который будет задан в настройках публикации (переадресации). Соответственно, далее через веб интерфейс настроек маршрутизатора был установлен статический IP-адрес для ПК Docker-хоста (рис. 12).

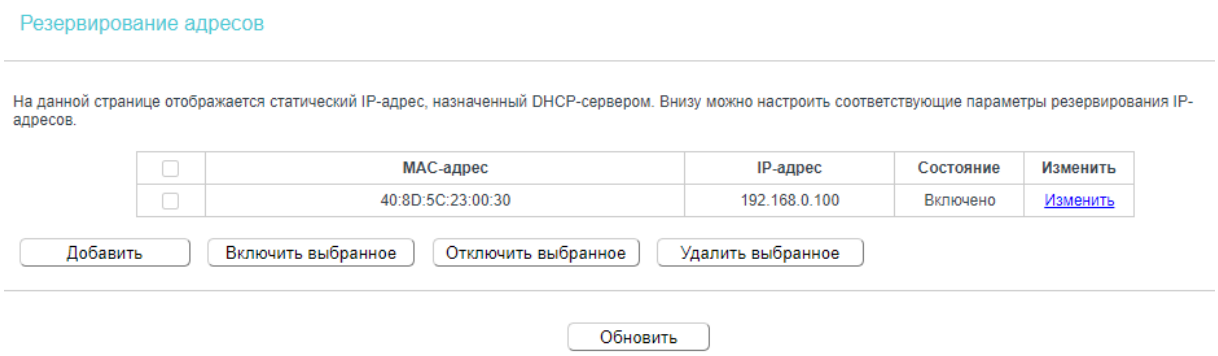

Рисунок 12. Резервирование IP-адресов маршрутизатора.

Затем в этих же настройках маршрутизатора была установлена переадресация подключений на 80 порт IP-адреса ПК Docker-хоста, где работает Nginx.

Рисунок 13. Таблица опубликованных адресов маршрутизатора

Обычно доступ к веб-сайтам не осуществляется напрямую по IP-адресу, а используется доменное имя, при обращении к которому сервер доменных имен (DNS) осуществляет перенаправление на необходимый IP-адрес. Существует бесплатный (для одного доменного имени) сервис предоставления доменных имен No-IP (*https://www.noip.com/*). Данный сервис связывает в том числе динамический внешний IP-адрес с доменным именем и при динамическом внешнем IP-адресе доменное имя будет оставаться статическим. На данном сервисе был зарегистрирован аккаунт и создано доменное имя с настройками переадресации (рис. 14).

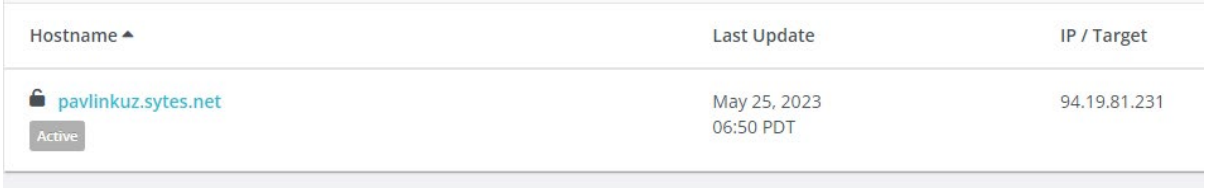

Рисунок 14. Создание доменного имени на сервисе No-IP.

Далее в настройках маршрутизатора была осуществлена связь с созданным аккаунтом и DNS (рис. 15).

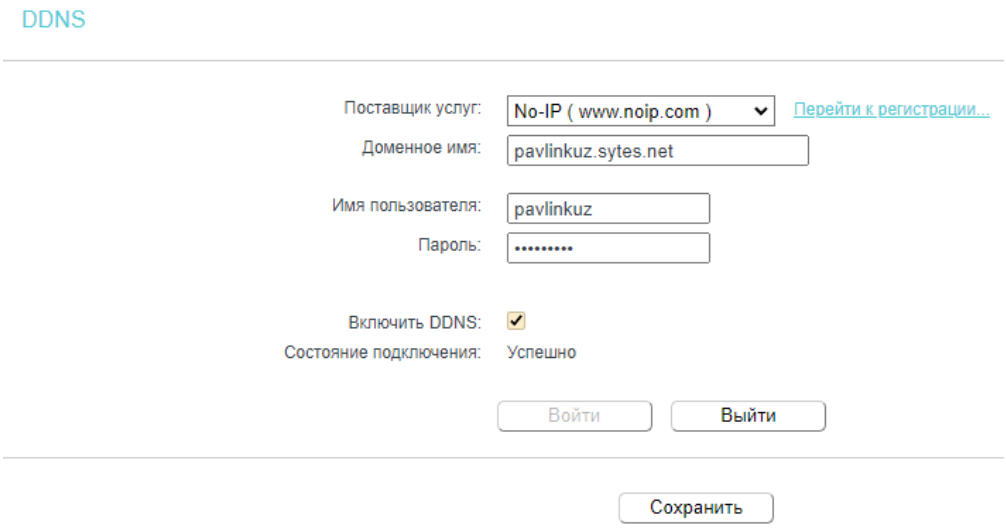

Рисунок 15. Связывание DDNS с маршрутизатором.

Итак, по адресу *http://pavlinkuz.sytes.net/* осуществляется доступ к веб-сайту с картографической визуализацией потенциала поля расселения трансграничных регионов России (рис. 16).

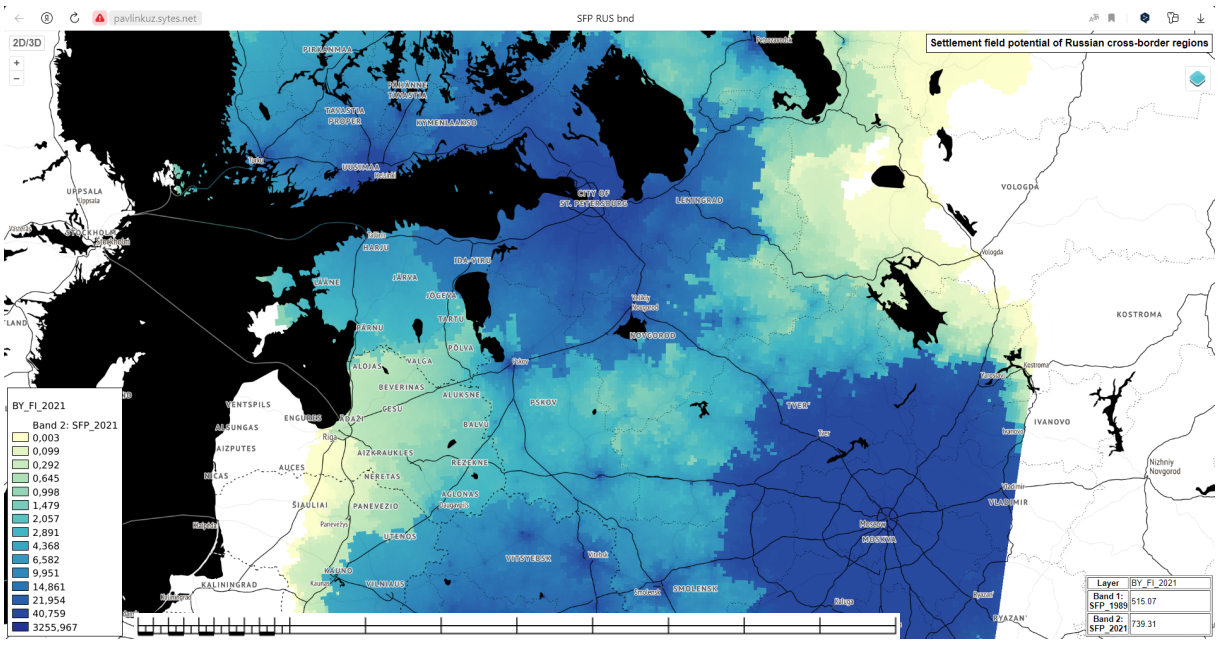

Рисунок 16. Веб-сайт с картографической визуализацией ППР.

Напоследок рассматривается итоговая архитектура полученного веб-приложения (рис. 17). На базе Nginx расположены собранные для хостинга файлы HTML, CSS и JavaScript. Порт 80 контейнера Nginx опубликован на хост-операционную систему и по нему осуществляется доступ к веб-сайту из глобальной сети. В JavaScript-файле по адресу, ведущему на тот же Nginx-контейнер, делаются WMS-запросы к сервису QGIS Сервера на получение тайлов карты, легенды и информации о слоях.

Nginx, как обратный прокси-сервер, перенаправляет WMS-запросы на изолированную сеть Docker Compose, а именно на контейнер с QGIS Сервером, заключенным внутри неё. При всём этом для обоих контейнеров отображены необходимые директории хоста. Для QGIS Сервера это директория с файлом проекта QGIS, а для Nginx – папка html со статическим контентом.

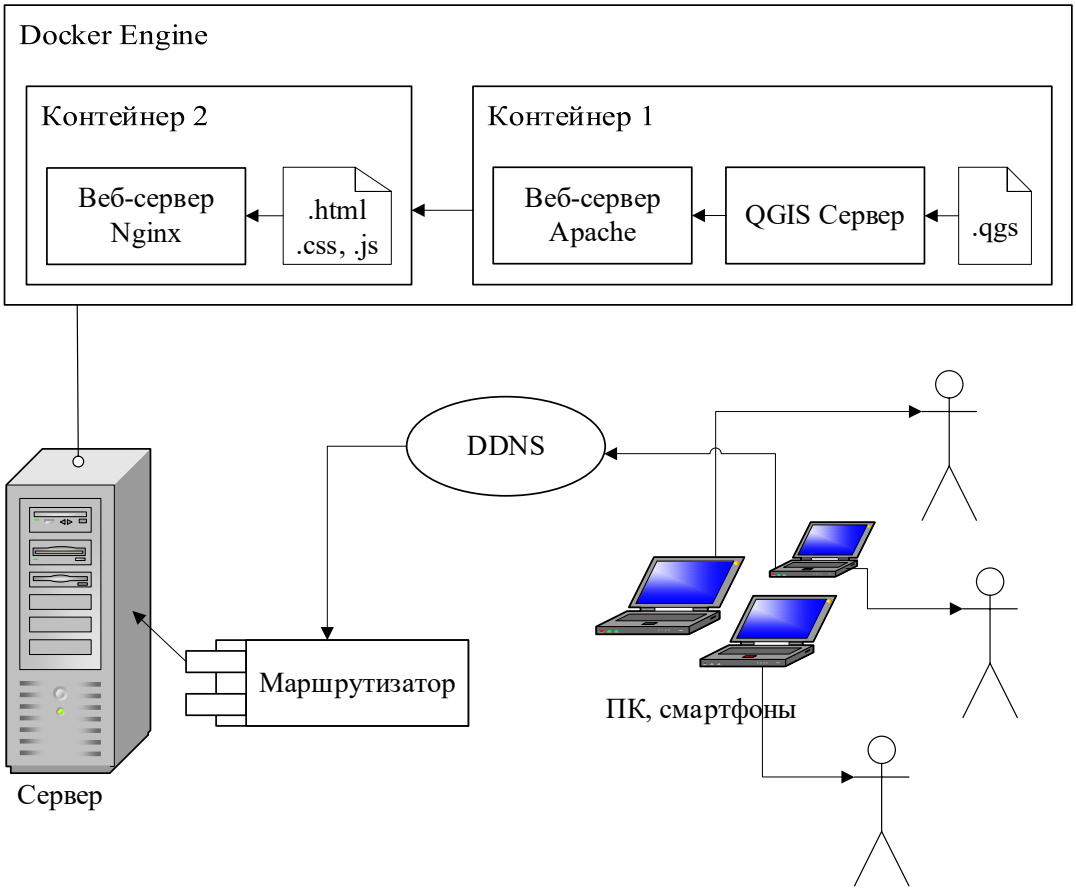

Рисунок 17. Схема веб-приложения картографической визуализации ППР.

ЗАКЛЮЧЕНИЕ

Выполнена алгоритмизация для модифицированной гравитационной модели потенциала поля расселения, сформулирована общая методика отбора и обработки данных при выполнении вычислений с использованием модифицированной гравитационной модели потенциала поля расселения.

В частности, приведены способы решения задачи построения матрицы расстояний по дорожной сети с помощью открытых механизмов массовой автоматической маршрутизации, а именно: локальные механизмы маршрутизации

Valhalla и OSRM с использованием Docker и веб-сервис OSRM от FOSSGIS, наглядно показаны концептуальные и технические различия их использования. На Python произведен расчет потенциала поля расселения и экспорт результатов в виде растров. Создан веб-сайт для картографической визуализации полученных растров.

Общая методика работы:

1. Получение данных о населенных пунктах с их численностями в разные годы, определение географического охвата
2. Построение матрицы расстояний по дорожной сети
   - Получение копии базы геоданных OSM
   - Извлечение интересующей области из полученных данных
   - Построение дорожного графа для механизма маршрутизации
   - Подбор проекции (системы координат) для регулярной сетки
   - Создание регулярной сетки
   - Перепроецирование сетки для механизма маршрутизации
   - Построение матрицы расстояний механизмом маршрутизации
3. Применение фильтров к матрице расстояний, расчет ППР
   - Внесение поправок в расстояния в виде домеров на концах маршрутов
   - Фильтрация и преобразование матрицы расстояний
   - Расчёт ППР
   - Экспорт значений ППР на разные годы в виде многоканального растра на основе регулярной сетки
4. Картографическая визуализация ППР на веб-сайте
   - Настройка проекта QGIS для отображения растров ППР
   - Развёртывание контейнеров Docker-Compose с QGIS Server и Nginx
   - Конфигурирование Nginx
   - Разработка интерактивной веб-карты с использованием OpenLayers
   - Настройка собственного хостинга веб-сайта

Созданные алгоритмы:

- Предобработка данных о населенных пунктах на языке Python и с использованием QGIS

- Построение матрицы расстояний с использованием механизма маршрутизации OSRM (FOSSGIS) на языке Python
- Развертывание механизма маршрутизации Valhalla (Local) с использованием Docker
- Построение матрицы расстояний с использованием механизма маршрутизации Valhalla (Local) на языке Python
- Развертывание механизма маршрутизации OSRM (Local) с использованием Docker
- Построение матрицы расстояний с использованием механизма маршрутизации OSRM (Local) на языке Python
- Вычисление потенциала поля расселения на языке Python
- Экспорт результата расчётов потенциала поля расселения на языке Python
- Создание картографической визуализации потенциала поля расселения в QGIS
- Создание веб-сайта с интерактивной картой потенциала поля расселения
- Развертывание инфраструктуры веб-сайта с использованием Docker-Compose
- Организация собственного хостинга веб-сайта

ЛИТЕРАТУРА

1. Sen, A., Smith, T.E., 1995. Gravity models of spatial interaction behavior. Springer. doi:10.1007/978-3-642-79880-1
2. P. Kuzmin, M. Karpenko, E. Panidi, A. Sebentsov, 2023. Road network accounting when estimating settlement field potential. Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XLVIII-4/W2-2022, 65–70, 2023 https://doi.org/10.5194/isprs-archives-XLVIII-4-W2-2022-65-2023
3. Создаев А.А., Тесленок К.С., 2019. Технология подготовки материалов для картографирования потенциала поля расселения. Огарёв-online, 3(124), 8 с.

4. Dong, Y., Cheng, P., Kong, X., 2022. Spatially explicit restructuring of rural settlements: A dual-scale coupling approach. Journal of Rural Studies, 94, 239-249. doi: 10.1016/j.jrurstud.2022.06.011
5. Головина Е.Д., Зотова М.В., Себенцов А.Б., Тикунов В.С., Картографирование барьерной функции государственных границ // Геодезия и картография. – 2015. – № 3. – С. 29-38. DOI: 10.22389/0016-7126-2015-897-3-29-38
6. Kolosov V.A., Rudenko L.G., Tikunov V.S., Gercen A.A., Golovina E.D., Zotova M.V., Sebentsov A.S. ATLAS INFORMATION SYSTEM OF RUSSIAN-UKRAINIAN BORDERLAND. Proceedings of the International conference “InterCarto. InterGIS”. 2014; 20:24-44.
7. Плас Дж. Вандер. Python для сложных задач: наука о данных и машинное обучение. — СПб.: Питер, 2018. — 576 с.: ил. — (Серия «Бестселлеры O’Reilly»). ISBN 978-5-496-03068-7
8. Кочер П. С. Микросервисы и контейнеры Docker / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2019. – 240 с.: ил.
9. Valhalla in Docker. Container recipes *https://hub.docker.com/r/gisops/valhalla*, 28.05.2023
10. OSRM in Docker. Quick Start *https://hub.docker.com/r/osrm/osrm-backend*, 28.05.2023
11. Osmium-tool in Docker. Example usage *https://hub.docker.com/r/stefda/osmium-tool*, 28.05.2023
12. Библиотека NumPy. Документация *https://numpy.org/doc/stable/reference/*, 28.05.2023
13. Библиотека Pandas. Документация *https://pandas.pydata.org/docs/reference/*, 28.05.2023
14. Библиотека RoutingPy. Документация *https://routingpy.readthedocs.io/*, 28.05.2023
15. Библиотека PyProj. Документация *https://pyproj4.github.io/pyproj/stable/*, 28.05.2023
16. Библиотека Rasterio. Документация *https://rasterio.readthedocs.io/*, 28.05.2023
17. QGIS. Руководство пользователя *https://docs.qgis.org/3.28/en/docs/user_manual/index.html*, 28.05.2023
18. Библиотека OpenLayers. Примеры использования *https://openlayers.org/en/v6.15.1/examples/*, 28.05.2023
19. Модуль OpenLayers LayerSwitcher. Примеры использования *https://github.com/walkermatt/ol-layerswitcher*, 28.05.2023
20. Модуль OpenLayers OL-Cesium. Интеграция *https://openlayers.org/ol-cesium/*, 28.05.2023

21. Картографическая основа Stamen. How to Use These Tiles Elsewhere *http://maps.stamen.com/*, 28.05.2023
22. QGIS Server in Docker. Usage *https://hub.docker.com/r/camptocamp/qgis-server*, 28.05.2023
23. Nginx in Docker. How to use this image *https://hub.docker.com/_/nginx*, 28.05.2023

ПРИЛОЖЕНИЕ А. Программный код веб-сайта с картографической визуализацией потенциала поля расселения.

А1. Содержимое файла index.html.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>SFP RUS bnd</title>
  </head>
  <body>
    <div id="map-container"></div>
    <h4>Settlement field potential of Russian cross-border
regions</h4>
    <img id="legend"/>
    <div id="info"> </div>
    <input id="enable" type="button" value="2D/3D">
    <script src="main.js" type="module"></script>
    <script src="node_modules/cesium/Build/Cesium/Cesium.js"
type="text/javascript"></script>
  </body>
</html>
```

А2. Содержимое файла style.css.

```
@import "node_modules/ol/ol.css";
@import "node_modules/ol-layerswitcher/dist/ol-layerswitcher.css";

html, body, #map-container {
  position: absolute;
  z-index: 1;
  margin: 0;
```

```
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  font-family: sans-serif;
  background-color: #ffffff;
}
h4 {
  position: absolute;
  z-index: 3;
  padding: 5px;
  margin: 5px 5px 90% 25%;
  top: 0;
  right: 0;
  background-color: #ffffff;
  border: 1px solid #000000;
  font-size: medium;
}
img {
  max-height: 50vh;
  max-width: 30vw;
  height: auto;
  width: auto;
}
#legend {
  position: absolute;
  z-index: 2;
  margin: 5px;
  bottom: 0;
  left: 0;
  border: 1px solid #000000;
}
#info {
  display: flex;
  position: absolute;
  z-index: 3;
  margin: 5px;
  bottom: 0;
  right: 0;
  background-color: #ffffff;
  font-size: small;
}
#enable {
  display: block;
  position: absolute;
  z-index: 4;
  top: 0.5em;
  left: 0.5em;
  font-weight: bold;
  border: 1px solid #d2d2d2;
```

```
  padding: 5px;
  color: var(--ol-subtle-foreground-color);
  text-decoration: none;
  font-size: inherit;
  text-align: center;
  height: 1.5em;
  line-height: .4em;
  background-color: var(--ol-background-color);
  border-radius: 3px;
 }
.ol-zoom {
  top: 2.5em;
  left: 0.5em;
}
```

А3. Содержимое файла main.js.

```
import './style.css';
import Map from 'ol/Map';
import OLCesium from 'olcs/OLCesium';
import LayerSwitcher from 'ol-layerswitcher';
import View from 'ol/View';
import TileLayer from 'ol/layer/Tile';
import TileWMS from 'ol/source/TileWMS';


const wms_url = 'http://pavlinkuz.sytes.net/'

const BY_FI_2021 = new TileWMS({
  url: wms_url,
  params: {'LAYERS': ['BY_FI_2021', 'Stamen_Toner_Background',
'Stamen_Terrain_Labels'], 'TILED': true},
});

const BY_FI_1989 = new TileWMS({
  url: wms_url,
  params: {'LAYERS': ['BY_FI_1989', 'Stamen_Toner_Background',
'Stamen_Terrain_Labels'], 'TILED': true},
});

const KZ_2017 = new TileWMS({
  url: wms_url,
  params: {'LAYERS': ['KZ_2017', 'Stamen_Toner_Background',
'Stamen_Terrain_Labels'], 'TILED': true},
});

const KZ_1989 = new TileWMS({
```

```
  url: wms_url,
  params: {'LAYERS': ['KZ_1989', 'Stamen_Toner_Background',
'Stamen_Terrain_Labels'], 'TILED': true},
});

const wmsLayer1 = new TileLayer({
  title: 'BY-FI 2021 (person/m)',
  type: 'base',
  combine: true,
  visible: true,
  source: BY_FI_2021,
});

const wmsLayer2 = new TileLayer({
  title: 'BY-FI 1989 (person/m)',
  type: 'base',
  combine: true,
  visible: false,
  source: BY_FI_1989,
});

const wmsLayer3 = new TileLayer({
  title: 'KZ 2017 (person/m)',
  type: 'base',
  combine: true,
  visible: false,
  source: KZ_2017,
});

const wmsLayer4 = new TileLayer({
  title: 'KZ 1989 (person/m)',
  type: 'base',
  combine: true,
  visible: false,
  source: KZ_1989,
});

const view = new View({
  center: [4188707, 7507612],
  zoom: 6.5,
  enableRotation: false,
});

const map = new Map({
  layers: [wmsLayer1, wmsLayer2, wmsLayer3, wmsLayer4],
  target: 'map-container',
  view: view,
});

const layerSwitcher = new LayerSwitcher({
```

```
  tipLabel: 'Layers',
  groupSelectStyle: 'children'
});
map.addControl(layerSwitcher);

const updateLegend = function (source) {
  const graphicUrl = source.getLegendUrl(
   undefined,
   {
    'LAYER': source.getParams()['LAYERS'][0],
   }
  );
  const img = document.getElementById('legend');
  img.src = graphicUrl;
};

let selectedLayerSource = BY_FI_2021;
updateLegend(selectedLayerSource);
LayerSwitcher.forEachRecursive(map, function(l, idx, a) {
  l.on("change:visible", function(e) {
   if (e.target.getVisible()) {
    selectedLayerSource = e.target.getSource();
    updateLegend(selectedLayerSource);
    document.getElementById('info').innerHTML = '';
   }
  });
});

map.on('singleclick', function (evt) {
  document.getElementById('info').innerHTML = '';
  const viewResolution = /** @type {number} */
(view.getResolution());
  const url = BY_FI_2021.getFeatureInfoUrl(
   evt.coordinate,
   viewResolution,
   'EPSG:3857',
   {
    'INFO_FORMAT': 'text/html',
    'QUERY_LAYERS': [selectedLayerSource.getParams()['LAYERS'][0]],
   },
  );
  if (url) {
   fetch(url)
    .then((response) => response.text())
    .then((html) => {
     document.getElementById('info').innerHTML = html;
    });
  }
});
```

```
const map3d = new OLCesium({map: map});
document.getElementById('enable').addEventListener('click', () =>
map3d.setEnabled(!map3d.getEnabled()));
```

A4. Содержимое файла package.json.

```
{
  "name": "ol-app",
  "version": "1.0.0",
  "scripts": {
    "start": "vite",
    "build": "vite build",
    "serve": "vite preview"
  },
  "devDependencies": {
    "cesium": "^1.105.1",
    "vite": "^4.3.5",
    "vite-plugin-cesium": "^1.2.22"
  },
  "dependencies": {
    "ol": "latest",
    "ol-layerswitcher": "^4.1.1",
    "olcs": "^2.14.0"
  }
}
```

A5. Содержимое файла vite.config.js.

```
import { defineConfig } from 'vite';
import cesium from 'vite-plugin-cesium';

export default defineConfig({
  plugins: [cesium()],
  build: {
    sourcemap: true,
  }
});
```