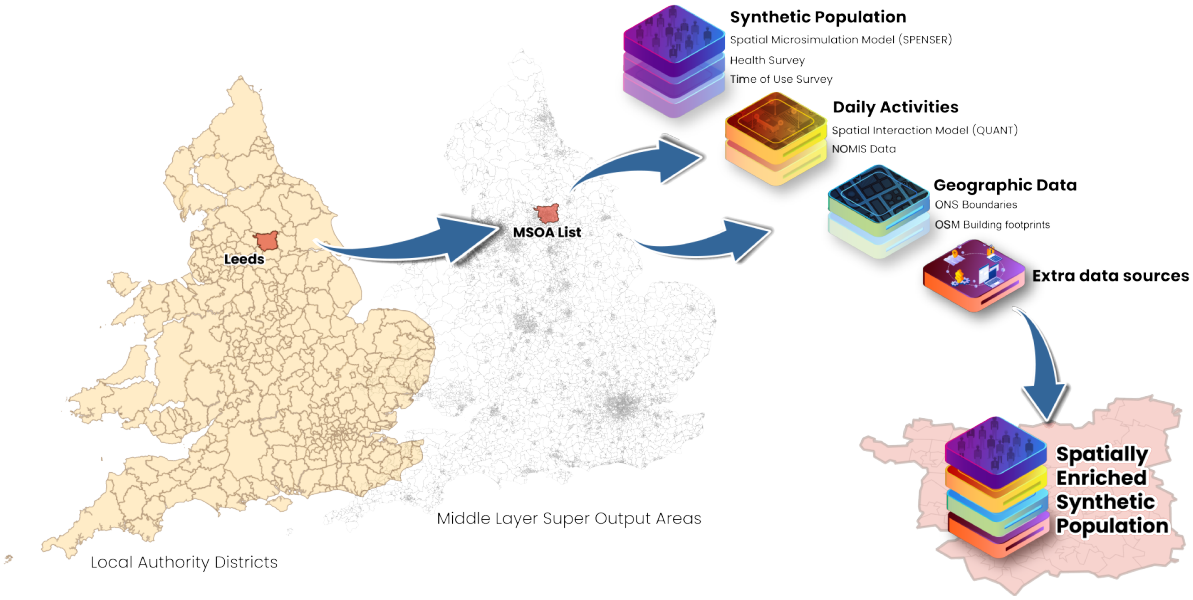
Performance

Table of contents

1 Getting started



The Synthetic Population Catalyst (SPC) makes it easier for researchers to work with synthetic population data in England. It combines a variety of data sources and outputs a single file in protocol buffer format, describing the population in a given study area. The data includes demographic, health, and daily activity data per person, and information about the venues where people conduct activities.

You can use SPC output to catalyze your own project. Rather than join together many raw data sources yourself and deal with missing and messy data, you can leverage SPC's effort and well-documented schema.

To get started:

1. Download sample data for a county in England
2. Explore how to use the data
3. If you need a different study area, build and then run SPC

You can also download this site as a PDF and find all code on Github.

This work was supported by Wave 1 of The UKRI Strategic Priorities Fund under the EPSRC Grant EP/W006022/1, particularly the "Ecosystem of Digital Twins"
and "Shocks and Resilience" themes within that grant & The Alan Turing Institute

Part I

Using SPC

2 Outputs for England Counties

You don't need to run SPC yourself. We provide output data for 47 ceremonial counties in England. (Note this list is missing the City of London.)

- bedfordshire
- berkshire
- bristol
- buckinghamshire
- cambridgeshire
- cheshire
- cornwall
  - Within this area, we're missing the Isles of Scilly (about 2,000 inhabitants)
- cumbria
- derbyshire
- devon
- dorset
- durham
- east_sussex
- east_yorkshire_with_hull
- essex
- gloucestershire
- greater_london
- greater_manchester
- hampshire (Southhampton)
- herefordshire
- hertfordshire
- isle_of_wight
- kent
- lancashire
- leicestershire
- lincolnshire
- merseyside (Liverpool)
- norfolk
- northamptonshire
- northumberland

- north_yorkshire
- nottinghamshire
- oxfordshire
- rutland
- shropshire
- somerset
- south_yorkshire
- staffordshire
- suffolk
- surrey
- tyne_and_wear (Newcastle)
- warwickshire
- west_midlands (Birmingham)
- west_sussex
- west_yorkshire (Leeds)
- wiltshire
- worcestershire

We also have a few special study areas defined:

- northwest_transpennine (Liverpool, Manchester, Leeds)
- oxford_cambridge_arc (Oxford, Milton Keynes, Cambridge)

See config/ for the list of MSOAs covered by each study area. If you want to run SPC for a different list of MSOAs, see here.

2.1 Citing

If you use SPC code or data in your work, please cite using the Zenodo DOI (using the bottom-right tool to generate the citation).

2.2 Versioning

Over time, we may add more data to SPC or change the schema. Protocol buffers are designed to let combinations of new/old code and data files work together, but we don't intend to use this feature. We may make breaking changes, like deleting fields. We'll release a new version of the schema and output data every time and document it here. You should depend on a specific version of the data output in your code, so new releases don't affect you until you decide to update.

- v1: released 25/04/2022, schema

- v1.1, released 27/05/2022, schema
  - added `pwkstat`, `salary_hourly`, `salary_yearly`, and `idp`
  - reorganized `Identifiers` and `Employment` attributes

3 Using the SPC output file

Once you download or generate an SPC output file for your study area, how do you use it? Each study area consists of one `.pb` or protocol buffer file. This file efficiently encodes data following this schema. Read more about what data is contained in the output.

You can read the "protobuf" (shorthand for a protocol buffer file) in any supported language, and then extract and transform just the parts of the data you want for your model.

We have examples for Python below, but feel free to request other languages.

3.1 Python

To work with SPC protobufs in Python, you need two dependencies setup:

- The protobuf library
  - You can install system-wide with `pip install protobuf`
  - Or add as a dependency to a conda, poetry, etc environment
- The generated Python library, synthpop_pb2.py
  - You can download a copy of this file into your codebase, then `import synthpop_pb2`
  - You can also generate the file yourself, following the docs: `protoc --python_out=python/ synthpop.proto`

3.1.1 Converting to Pandas data-frames and CSV

The schema expresses relationships between people, households, and venues that can't all be captured by a simple 2D table. Nevertheless, you can extract per-person information and express as a dataframe or CSV file. See this example Python script for inspiration. You can try it out:

```
# Download a file
wget https://ramp0storage.blob.core.windows.net/spc-output/v1/rutland.pb.gz
# Uncompress
gunzip rutland.pb.gz
# Convert the .pb to JSON
python3 python/protobuf_to_csv.py data/output/rutland.pb
# View the output
less people.csv
```

3.1.2 Converting .pb file to JSON format

To interactively explore the data, viewing JSON is much easier. It shows the same structure as the protobuf, but in a human-readable text format. The example below uses a small Python script:

```
# Download a file
wget https://ramp0storage.blob.core.windows.net/spc-output/v1/rutland.pb.gz
# Uncompress
gunzip rutland.pb.gz
# Convert the .pb to JSON
python3 python/protobuf_to_json.py data/output/rutland.pb > rutland.json
# View the output
less rutland.json
```

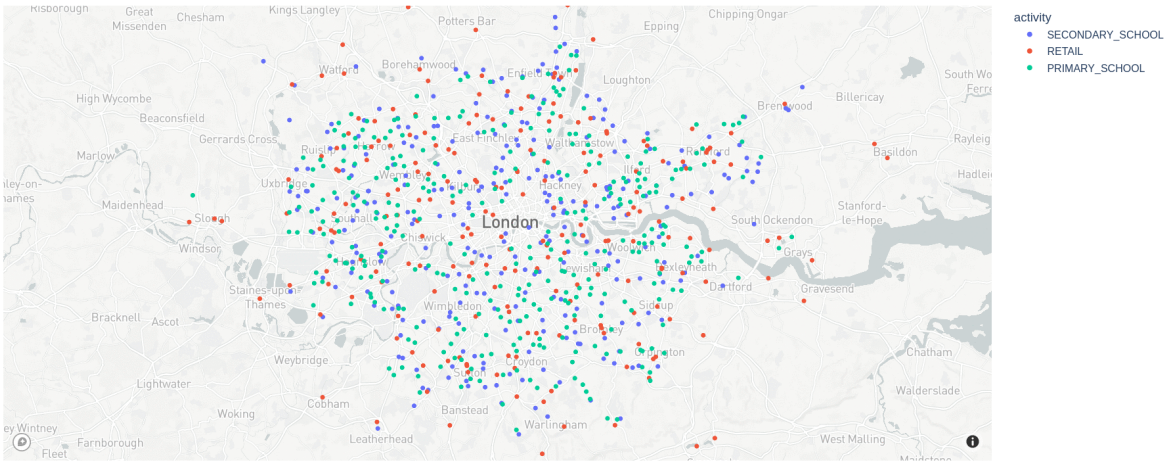
3.1.3 Converting to numpy arrays

The ASPICS project simulates the spread of COVID through a population. The code uses numpy, and this script converts the protobuf to a bunch of different numpy arrays.

Note the ASPICS code doesn't keep using the generated Python protobuf classes for the rest of the pipeline. Data frames and numpy arrays may be more familiar and appropriate. The protobuf is a format optimized for reading and writing; you don't need to use it throughout all of your model code.

3.1.4 Visualizing venues

Use this script to read a protobuf file, then draws a dot for every venue, color-coded by activity.


activity
SECONDARY_SCHOOL
RETAIL
PRIMARY_SCHOOL
London
Chesham
Kings Langley
Potters Bar
Epping
Chipping Ongar
Watford
Borehamwood
Enfield
Loughton
High Wycombe
Beaconsfield
Billericay
Brentwood
Gerrards Cross
East Finchley
Basildon
Marlow
Uxbridge
Hackney
Ilford
Maidenhead
Slough
Stanford-le-Hope
South Ockendon
Windsor
Chiswick
Woolwich
Grays
Staines-upon-Thames
Dartford
Gravesend
Bracknell
Ascot
Wimbledon
Sidcup
Bromley
Chatham
Weybridge
Croydon
Lightwater
Walderslade
Cobham
Woking
Banstead
Warlingham
Farnborough
Leatherhead
West Malling
Fleet

4 Installation

You only need to compile SPC to run for a custom set of MSOAs. Just download existing output if your study area matches what we provide.

4.1 Dependencies

- **Rust**: The latest version of Rust (1.60): https://www.rust-lang.org/tools/install
- A build environment for proj, to transform coordinates.
  - On Ubuntu, run `apt-get install cmake sqlite3 libclang-dev`
  - On Mac, install Homebrew and run `brew install pkg-config cmake proj`

4.2 Compiling SPC

```
git clone https://github.com/alan-turing-institute/uatk-spc/
cd uatk-spc
# The next command will take a few minutes the first time you do it, to build external depen
cargo build --release
```

If you get `error: failed to run custom build command for 'proj-sys v0.18.4'`, then you're likely missing dependencies listed above. Please open an issue if you have any trouble.

4.3 Troubleshooting downloading

If you get an error `No such file or directory (os error 2)` it might be because a previous attempt to run SPC failed, and some necessary files were not fully downloaded. In these cases you could try deleting the `data/raw_data` directory and then running SPC again. It should automatically try to download the big files again.

If you have trouble downloading any of the large files, you can download them manually. The logs will contain a line such as `Downloading https://ramp0storage.blob.core.windows.net/nationaldata/`

`to data/raw_data/nationaldata/QUANT_RAMP_spc.tar.gz`. This tells you the URL to retrieve, and where to put the output file. Note that SPC won't attempt to download files if they already exist, so if you wind up with a partially downloaded file, you have to manually remove it.

5 Creating new study areas

If the area you want to model isn't already generated, then you can follow this guide to run SPC on a custom area. You must first compile SPC.

5.1 Specifying the area

SPC takes a newline-separated list of MSOAs in the `config/` directory as input, like this. You can generate this list from a LAD (local authority district). From the main SPC directory, run `python scripts/select_msoas.py`. Refer to `data/raw_data/referencedata/lookUp.csv` (only available after running SPC once) for all geographies available.

This script will create a new file, `config/your_region.txt`.

5.2 Run SPC for the new area

From the main directory, just run:

```
cargo run --release -- config/your_region.txt
```

This will download some large files the first time. You'll wind up with `data/output/your_region.pb` as output, as well as lots of intermediate files in `data/raw_data/`. The next time you run this command (even on a different study area), it should go much faster.

5.3 (Optional) run SPC for lots of areas

If you want to run the program over lots of areas at once and are using Mac/Linux, you can use a `for` loop in a terminal to repeatedly run SPC over all files in the config directory. For example, this will run SPC on all `.txt` files in the `config` directory:

```
for file in config/*.csv; do cargo run --release -- config/$file; done
```

5.4 Using the output

After you generate the files, see here for how to use them in your project.

If you use SPC code or data in your work, please cite using the Zenodo DOI (using the bottom-right tool to generate the citation).

Part II

Understanding SPC

6 Data schema

6.1 Understanding the schema

Here are some helpful tips for understanding the schema.

Each .pb file contains exactly one `Population` message. In contrast to datasets consisting of multiple `.csv` files, just a single file contains everything. Some of the fields in `Population` are lists (of people and households) or maps (of venues keyed by activity, or of MSOAs). Unlike a flat `.csv` table, there may be more lists embedded later. Each `Household` has a list of `members`, for example.

The different objects refer to each other, forming a graph structure. The protobuf uses `uint64` IDs to index into other lists. For example, if some household has `members = [3, 10]`, then those two people can be found at `population.people[3]` and `population.people[10]`. Each of them will have the same `household` ID, pointing back to something in the `population.households` list.

6.2 Flows: modelling daily activites

SPC models daily travel behavior of people as "flows." Flows are broken down by by an activity – shopping/retail, attending primary or secondary school, working, or staying at home. For each activity type, a person has a list of venues where they may do that activity, weighted by a probability of going to that particular venue.

Note that `flows_per_activity` is stored in `InfoPerMSOA`, not `Person`. The flows for retail and school are only known at the MSOA level, not individually. So given a particular `Person` object, you first look up their household's MSOA – `msoa = population.households[ person.household ].msoa` and then look up flows for that MSOA – `population.info_per_msoa[msoa].flows_per_activity`.

Each person has exactly 1 flow for home – it's just `person.household` with probability 1. A person has 0 or 1 flows to work, based on the value of `person.workplace`.

This doesn't mean that all people in the same MSOA share the same travel behavior. Each person has their own `activity_durations` field, based on time-use survey data. Even if two

people share the same set of places where they may go shopping, one person may spend much more time on that activity than another.

See the ASPICS conversion script for all of this in action – it has a function to collapse a person's flows down into a single weighted list.

Note that per MSOA, very few venues are represented as destinations – 10 for retail and 5 for school. Only the most likely venues from QUANT are used.

6.3 Flow weights

How do you interpret the probabilities/weights for flows? If your model needs people to visit specific places each day, you could randomly sample a venue from the flows, weighting them appropriately. For retail, you may want to repeat this sampling every day of the simulation, so they visit different venues. For primary and secondary school, it may be more appropriate to sample once and store that for the simulation – a student probably doesn't switch schools daily.

Alternatively, you can follow what ASPICS does. Every day, each person logically visits all possible venues, but their interaction there (possibly receiving or transmitting COVID) is weighted by the probability of each venue.

7 Modelling methods

The principles behind the generation of the enriched SPENSER population data and behind the modelling of trips to schools and retail from QUANT are detailed in

> Spooner F et al. A dynamic microsimulation model for epidemics. Soc Sci Med. 291:114461 (2021). (DOI)

7.1 Commuting flows

In order to distribute each individual of the population to a unique physical workplace, we first created a population of all individual workplaces in England, based on a combination of the Nomis UK Business Counts 2020 dataset and the Nomis Business register and Employment Survey 2015 (see Data sources). The first dataset gives the number of individual workplace counts per industry, using the SIC 2007 industry classification, with imprecise size (i.e. number of employees) bands at MSOA level. The second dataset gives the total number of jobs available at LSOA level per SIC 2007 industry category. We found that the distribution of workplace sizes follows closely a simple 1/x distribution, allowing us to draw for each workplace a size within their band, with sum constraints given by the total number of jobs available, according to the second dataset.

The workplace 'population' and individual population are then levelled for each SIC 2007 category by removing the exceeding part of whichever dataset lists more items. This takes into account that people and business companies are likely to over-report their working availability (e.g. part time and seasonal contracts are not counted differently than full time contracts, jobseekers or people on maternity leave might report the SIC of their last job). This process can be controlled by a threshold in the parameter file that defines the maximal total proportion of workers or jobs that can be removed. If the two datasets cannot be levelled accordingly, the categories are dropped and the datasets are levelled globally. Tests in the West Yorkshire area have shown than when the level 1 SIC, containing 21 unique categories, is used, 90% of the volume of commuting flows were recovered compared to the Nomis commuting OD matrices at MSOA level.

The employees for each workplace are drawn according to the 'universal law of visitation', see

Schläpfer M et al. The universal visitation law of human mobility. Nature 593, 522–527 (2021). (DOI)

This framework predicts that visitors to any destination follow a simple

(r,f)= K / (rf)2

distribution, where (r,f) is the density of visitors coming from a distance r with frequency f and K is a balancing constant depending on the specific area. In the context of commuting, it can be assumed that f = 1. Additionally, we only need to weigh potential employees against each other, which removes the necessity to compute explicitly K. In the West Yorkshire test, we found a Pearson coefficient of 0.7 between the predicted flows when aggregated at MSOA level and the OD matrix at MSOA level available from Nomis.

7.2 Income data

This modelling is mainly based on the 2020 revised edition of the Earnings and hours worked, region by occupation by four-digit SOC: ASHE Table 15 database from ONS. Some percentiles for employees' gross hourly salaries are provided for each full-time and part-time job according to their four-digit SOC classification per region, and separated by sex.

7.2.1 Methods

The data are far from complete (only about 15% of all possible values), especially for the highest deciles. We found that an order 3 polynomial fit was satisfactory for most categories (93.11%) to complete the partially filled SOCs. SOCs with too many missing values are given the value for the category that is immediately higher in the SOC hierarchy. Some jobs appear to have a 'ceiling' for the highest percentiles, making the polynomial fit fail. In that case, we have replaced the unknown values by the highest known value in the raw data (as there is no clear and systemic fit for these special cases). In addition, there is no information for the highest decile in all cases, which means that the highest salaries are underestimated (and exceptionally high salaries cannot be obtained). The result of this phase is four tables {male full-time, male part-time, female full-time, female part-time} containing the coefficients of the fitted order 3 polynomial, with an optional ceiling percentile when relevant.

A percentile is chosen randomly (uniformly) for each individual, and the salary is then deduced according to their full-time/part-time status, region, sex and SOC category. A basic hourly salary column is added to the unprocessed SPC data, as well as a corresponding annual salary based on their estimated hours worked per day, according to the Time Use Survey matching. In addition, we repeat this process for all individuals that are categorised as 'Self-employed' or 'Employee unspecified' by the Time Use Survey matching,, as if they were full time employees. These values are recorded in the columns `IncomeHAsIF` and `IncomeYAsIf`. We noticed that

a high number of employees were given no worked hours by the Time Use Survey. We have added to the `IncomeYAsIf` column an estimation of their annual salary based on Table 15.9a: Paid hours worked - Total 2020, and also depending on the same four variables as above (full-time/part-time status, region, sex and SOC category).

In addition, age data are made available by ONS. Part of the differences that can be observed between different age groups are already taken into account through the fact that the SOC category can evolve during a career. To take into account that dependence, we first run the above method without weighing by age. The results are shown in the age validation section below. The residual impact of age alone is then added to the model in the following way. When the percentile is drawn for a specific individual, it is morphed to fit within the usual percentage range accessible to that age category. The function that operates this morphing is inferred beforehand and takes into account the salary distribution per age computed by the previous non-age weighted iteration of the modelling (see figure - TBA - for a more detailed description of this function).

The R codes for this modelling are here.

The methods are validated in the next section. Since it is not possible to optimise every criterion at once, this next secton can also be used as a reference to re-adjust some values to match exactly the ONS estimated means for one particular criterion of interest.

7.2.2 Comparison to reference values from ONS

We compare the results of the modelling to the raw datasets from ONS.

- `Mod` for modelled
- `M` for male
- `F` for female
- `H` for hourly gross salary
- `Y` for annual gross salary
- `FT` for full-Time
- `PT` for part-Time
- Only individuals recorded as employees (i.e. not self-employed) are taken into account in this section.

Number of employees per sex and full-time/part-time classification

The numbers given by ONS vary from dataset to dataset and are reported by ONS as indicative only. For the modelled values, we give the total number of individuals with a non-zero salary in each category.

Variable	All	FT	PT	M	M FT	M PT	F	F FT	F PT
ONS tot	22-26k	16-19k	6-8k	11-13k	9-11k	1.5-2k	11-13k	6.5-7.5k	4.5-5.5k
Mod tot H	23.1k	18.5k	4.6k	11.8k	11k	0.8k	11.3k	7.5k	3.8k
Mod tot Y	17.6k	14.8k	2.8k	9.4k	8.9k	0.5k	8.2k	5.9k	2.3k

A significant number of individuals listed as working either full or part time have 0 effective worked hours per day according to the Time Use Survey matching. In those cases, an hourly salary is modelled depending on their SOC, region and sex, as for any other employee, but the annual salary will be displayed as 0. It is possible to estimate their likely true number of hours worked from the same ONS dataset (Table 15.9a: Paid hours worked - Total 2020), also depending on their sex, soc and region. This calculation has been added to the "As If" column.

Hourly gross salary per sex and full-time/part-time classification

Variable	All	FT	PT	M	M FT	M PT	F	F FT	F PT
ONS mean	17.63	18.32	13.93	18.81	19.12	14.69	16.19	17.08	13.68
ONS median	13.71	15.15	10.38	14.84	15.58	10.12	12.58	14.42	10.47
Mod mean	16.45	17.19	13.45	17.50	17.84	12.75	15.35	16.23	13.60
Mod median	13.55	14.46	10.23	14.27	14.72	9.16	12.79	14.12	10.51

The median values are quite close to the ONS values, but the mean values are always lower. This is expected, see the description of the modelling above.

Annual gross salary per sex and full-time/part-time classification

Only values > 0 are retained for these calculations.

Variable	All	FT	PT	M	M FT	M PT	F	F FT	F PT
ONS mean	31,646	38,552	13,819	38,421	42,072	14,796	24,871	33,253	13,512
ONS median	25,886	31,487	11,240	31,393	33,915	10,883	20,614	28,002	4,743
Mod mean	34,317	36,595	22,257	37,574	38,496	20,698	30,594	33,729	22,585
Mod median	28,713	30,942	17,928	31,404	32,382	17,382	25,875	29,028	18,137

The average salary for part-time employees is correct when values equal to 0 are taken into account. This suggests that the total number of hours worked for part-time employees is

correct, but the way they are distributed among individuals is not. It could be due to the TUS taking a snapshot of the situation during a particular week, rather than averaging their data over the year. It appears that the TUS matching also overestimates the average number of hours worked for female employees.

Regional differences (hourly gross salary)

Region	East	East Mid-lands	London	North East	North West	South East	South West	West Mid-lands	Yorkshire and The Humber
ONS mean	16.74	15.87	23.78	15.69	16.36	17.88	16.36	16.34	15.76
ONS median	13.28	12.65	18.30	12.40	12.90	14.33	12.74	12.92	12.46
Mod mean	16.67	15.29	19.39	15.05	15.22	17.34	15.92	15.47	14.41
Mod median	13.69	12.79	16.25	12.42	12.44	14.84	13.35	12.64	12.44

The pearson correlations for mean and median between the modelled and raw values are 0.92 and and 0.93.

Hourly gross salary per one-digit SOC

1d SOC	1	2	3	4	5	6	7	8	9
ONS mean	26.77	23.38	18.29	13.42	13.35	10.87	10.94	12.23	10.77
ONS median	20.96	21.34	15.66	11.54	12.04	10.08	9.52	10.93	9.22
Mod mean	21.52	22.14	16.00	12.76	12.55	10.49	10.50	12.05	9.87
Mod median	17.22	20.66	14.12	11.46	11.34	9.71	9.59	10.82	9.12

1. Managers, directors and senior officials
2. Professional occupations
3. Associate professional and technical occupations
4. Administrative and secretarial occupations
5. Skilled trades occupations
6. Caring, leisure and other service occupations
7. Sales and customer service occupations
8. Process, plant and machine operatives
9. Elementary occupations.

The pearson correlations for mean and median between the modelled and raw values are 0.98 and 0.98.

Hourly gross salary per age

The reference for this table is: Table 6.5a Hourly pay - Gross 2020

Table before weighting by age:

Age	16-17	18-21	22-29	30-39	40-49	50-59	60+
ONS mean	7.21	9.59	14.09	18.13	20.04	19.12	16.32
ONS median	6.36	9.00	12.26	15.08	15.89	14.39	12.17
Mod mean	12.77	14.96	16.33	16.93	16.83	16.66	16.29
Mod median	10.93	12.71	13.88	14.02	13.96	13.85	13.65

The pearson correlations for mean and median between the modelled and raw values are 0.92 and 0.92.

Table after weighting by age:

Age	16-17	18-21	22-29	30-39	40-49	50-59	60+
ONS mean	7.21	9.59	14.09	18.13	20.04	19.12	16.32
ONS median	6.36	9.00	12.26	15.08	15.89	14.39	12.17
Mod mean	9.05	11.15	14.87	17.35	17.96	17.47	15.41
Mod median	8.20	9.51	12.86	14.41	14.78	14.43	12.56

The pearson correlations for mean and median between the modelled and raw values are 0.99 and 0.99.

8 Data sources

The data is sorted around the 2011 Middle-layer Super Output Area (MSOA) geographical unit. These units where created for census collection and are designed to be relatively homogeneous, with an average population size of 8000. Any list of MSOAs in England can be run, with the exception of the MSOAs forming the City of London (i.e the London borough called the City, not London as a whole).

The data from Open Street Map (OSM) is downloaded directly from https://www.openstreetmap.org. Everything else is hosted as local copies on one Azure repository that interacts automatically with the model, and divided into utilities, county level data and national data.

8.1 Utility data

`lookUp.csv`

The look-up table links different geographies together. It is used internally by the model, but can also help the user define their own study area. `MSOA11CD`, `MSOA11NM`, `LAD20CD`, `LAD20NM`, `ITL321CD`, `ITL321NM`, `ITL221CD`, `ITL221NM`, `ITL121CD`, `ITL121NM` are all standard denominations fully compatible with ONS fields of the same name. They are based on ONS lookups. See ONS documentation for more details. `CTY20NM` and `CCTY20NM` are custom denominations for the counties of England (used to sort the county level population data) and the ceremonial counties of England respectively. Their spelling may vary in different data sources and the field `CTY20NM` is not compatible with the ONS field of the same name (which excludes all counties that are also unitary authorities). `GoogleMob` and `OSM` are different spellings for the counties of England used by Google and OSM for their data releases.

8.2 County level data

Contains 47 files, each representing the population in 2020 of one of the counties of England mentioned above, and named

`tus_hse_<county_name>.gz`

This data is based on the 2011 UK census, the Time Use Survey 2014-15 and the Health Survey for England 2017. The SPENSER (Synthetic Population Estimation and Scenario Projection) microsimulation model (reference) distributes a synthetic population based on the census at MSOA scale and projects it to 2020 according to estimates from the Office for National Statistics (ONS). This information was enriched with some of the content of the other two datasets through propensity score matching (PSM) by Prof. Karyn Morrissey (Technical University of Denmark). The rest of the datasets can be added *a posteriori* from the identifiers provided.

The fields currently contained are:

- `idp`: a unique global individual identifier across all counties
- `MSOA11CD`: MSOA code where the individual lives
- `hid`: household identifier, includes communal establishments
- `pid`: identifier linking to the 2011 Census
- `pid_tus`: identifier linking to the Time Use Survey 2015
- `pid_hse`: identifier linking to the Health Survey for England 2017
- `sex`: 0 female; 1 male
- `age`: in years
- `origin`: 1 White; 2 Black; 3 Asian; 4 Mixed; 5 Other
- `nssec5`: National Statistics Socio-economic classification:
  - 1: Higher managerial, administrative and professional occupations
  - 2: Intermediate occupations
  - 3: Small employers and own account workers
  - 4: Lower supervisory and technical occupations
  - 5: Semi-routine and routine occupations
  - 0: Never worked and long-term unemployed
- `soc2010`: Previous version of the Standard Occupational Classification
- `sic1d07`: Standard Industrial Classification of Economic Activities 2007, 1st layer (number corresponding to the letter in alphabetical order)
- `sic2d07`: Standard Industrial Classification of Economic Activities 2007, 2nd layer
- `pwkstat`: Employment status according to the TUS
- Proportion of 24h spent doing different daily activities:
  - `punknown` + `pwork` + `pschool` + `pshop` + `pservices` + `pleisure` + `pescort` + `ptransport` = `pnothome`
  - `phome` + `pworkhome` = `phometot`
  - `pnothome` + `phometot` = 1
- `IncomeX`: hourly (X = "H") and annual (X = "Y") income for employees, see modelling methods for more details
- `cvd`: has a cardio-vascular disease (0 or 1)
- `diabetes`: has diabetes (0 or 1)

- `bloodpressure`: has high blood pressure (0 or 1)
- `BMIvg6`: Body Mass Index:
  - Not applicable
  - Underweight: less than 18.5
  - Normal: 18.5 to less than 25
  - Overweight: 25 to less than 30
  - Obese I: 30 to less than 35
  - Obese II: 35 to less than 40
  - Obese III: 40 or more
- `lng`: longitude of the MSOA11CD centroid
- `lat`: latitude of the MSOA11CD centroid

Some other fields were kept from and for other specific projects but are not from official sources and should generally not be used.

8.3 National data

`businessRegistry.csv`

Contains a breakdown of all business units (i.e. a single workplace) in England at LSOA scale (smaller than MSOA), estimated by the project contributors from two nomis datasets: UK Business Counts - local units by industry and employment size band 2020 and Business Register and Employment Survey 2015. Each item contains the `size` of the unit and its main `sic1d07` code in reference to standard Industrial Classification of Economic Activities 2007 (number corresponding to the letter in alphabetical order). It is used to compute commuting flows.

The R codes to compute this file are here.

`MSOAS_shp.tar.gz`

Is a simple shapefile taken from ONS boundaries.

`QUANT_RAMP.tar.gz`

See: Milton R, Batty M, Dennett A, dedicated RAMP Spatial Interaction Model GitHub repository. It is used to compute the flows towards schools and retail.

`timeAtHomeIncreaseCTY.csv`

This file is a subset from Google COVID-19 Community Mobility Reports, cropped to England. It describes the daily reduction in mobility, averaged at county level, due to lockdown and other COVID-19 restrictions between the 15th of February 2020 and 15th of April 2022. Missing values have been replaced by the national average. These values can be used directly to reduce `pnothome` and increase `phometot` (and their sub-categories) to simulate more accurately the period.

The R codes to process these data are here.

Part III

Advanced

9 Developer guide

9.1 Updating the docs

The site is built with Quarto. You can iterate on it locally: `cd docs; quarto preview`

9.2 Code hygiene

We use automated tools to format the code.

```
cargo fmt

# Format Markdown docs
prettier --write *.md
prettier --write docs/*.qmd --parser markdown
```

Install prettier for Markdown.

9.3 Some tips for working with Rust

There are two equivalent ways to rebuild and then run the code. First:

```
cargo run --release -- devon
```

The `--` separates arguments to `cargo`, the Rust build tool, and arguments to the program itself. The second way:

```
cargo build --release
./target/release/aspics devon
```

You can build the code in two ways – **debug** and **release**. There's a simple tradeoff – debug mode is fast to build, but slow to run. Release mode is slow to build, but fast to run. For the ASPICS codebase, since the input data is so large and the codebase so small, I'd recommend always using `--release`. If you want to use debug mode, just omit the flag.

If you're working on the Rust code outside of an IDE like VSCode, then you can check if the code compiles much faster by doing `cargo check`.

9.4 Docker

We provide a Dockerfile in case it's helpful for running, but don't recommend using it. If you want to, then assuming you have Docker setup:

```
docker build -t spc .
docker run --mount type=bind,source="$(pwd)"/data,target=/spc/data -t spc /spc/target/releas
```

This will make the `data` directory in your directory available to the Docker image, where it'll download the large input files and produce the final output.

10 Code walkthrough

SPC is implemented in Rust, and its code can be found here. This is an unusual implementation choice in the data science world, so this page has some notes about it.

10.1 Generally useful techniques

The code-base makes use of some techniques that may be generally applicable to other projects, independent of the language chosen.

10.1.1 Split code into two stages

Agent-based models and spatial interaction models require some kind of input. Often the effort to transform external data into this input can exceed that of the simulation component. Cleanly separating the two problems has some advantages:

- iterate on the simulation faster, without processing raw data every run
- reuse the prepared input for future projects
- force thinking about the data model needed by the simulation, and transform the external data into that form

SPC is exactly this first stage, originally split from ASPICS when further uses of the same population data were identified.

10.1.2 Explicit data schema

Dynamically typed languages like Python don't force you to explicitly list the shape of input data. It's common to read CSV files with `pandas`, filter and transform the data, and use that throughout the program. This can be quick to start prototyping, but is hard to maintain longer-term. Investing in the process of writing down types:

- makes it easier for somebody new to understand your system – they can first focus on **what** you're modeling, instead of how that's built up from raw data sources
- clarifies what data actually matters to your system; you don't carry forward unnecessary input

- makes it impossible to express invalid states
  - One example is here – per person and activity, there's a list of venues the person may visit, along with a probability of going there. If the list of venues and list of probabilities are stored as separate lists or columns, then their length may not match.
- reuse the prepared input for future projects

There's a variety of techniques for expressing strongly typed data:

- protocol buffers or flatbuffers
- JSON schemas
- Python data classes and optional type hints
- statically typed languages like Rust

10.1.3 Type-safe IDs

Say your data model has many different objects, each with their own ID – people, households, venues, etc. You might store these in a list and use the index as an ID. This is fine, but nothing stops you from confusing IDs and accidentally passing in venue 5 to a function instead of household 5. In Rust, it's easy to create "wrapper types" like this and let the compiler prevent these mistakes.

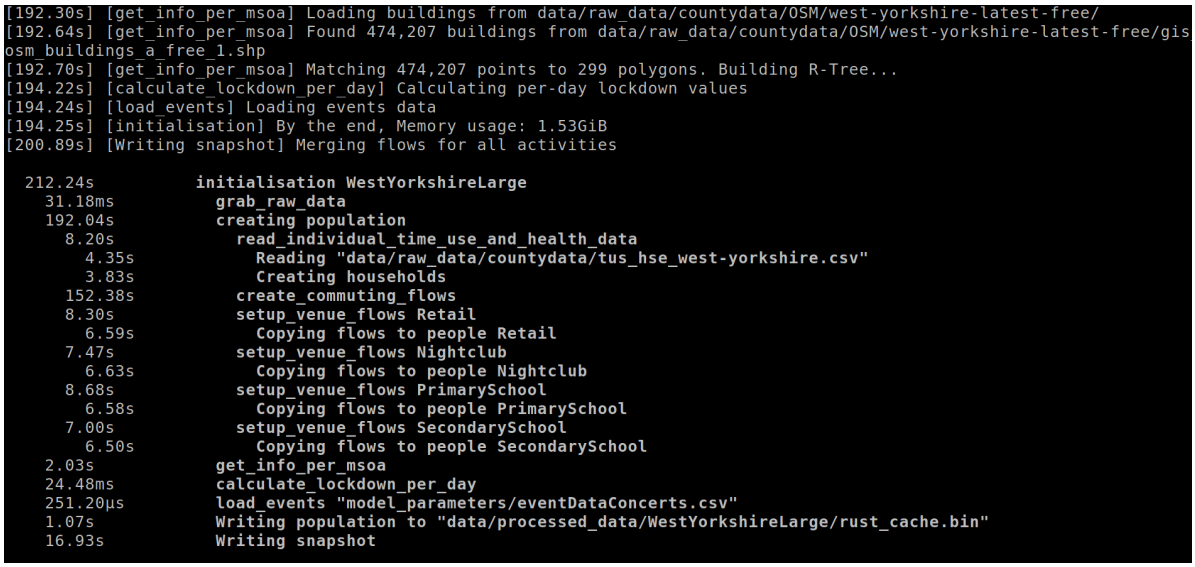
This technique is also useful when preparing external data. GTFS data describing public transit routes and timetables contains many string IDs – shapes, trips, stops, routes. As soon as you read the raw input, you can store the strings in more precise types that prevent mixing up a stop ID and route ID.

10.1.4 Idempotent data preparation

If you're iterating on your initialisation pipeline's code, you probably don't want to download a 2GB external file every single run. A common approach is to first test if a file exists and don't download it again if so. In practice, you may also need to handle unzipping files, showing a progress bar while downloading, and printing clear error messages. This codebase has some common code for doing this in Rust. We intend to publish a separate library to more easily call in your own code.

10.1.5 Logging with structure

It's typical to print information as a complex pipeline runs, for the user to track progress and debug problems. But without any sort of organization, it's hard to follow what steps take a long time or encounter problems. What if your logs could show the logical structure of your pipeline and help you understand where time is spent?



The screenshot above shows a summary printed at the end of a long pipeline run. It's immediately obvious that the slowest step is creating commuting flows.

This codebase uses the tracing framework for logging, with a custom piece to draw the tree. (We'll publish this as a separate library once it's more polished.) The tracing framework is hard to understand, but the main conceptual leap over regular logging framworks is the concept of a **span**. When your code starts one logical step, you call a method to create a new span, and when it finishes, you close that span. Spans can be nested in any way – `create_commuting_flows` happens within the larger step of `creating population`.

10.1.6 Determinism

Given the same inputs, your code should always produce identical output, no matter where it's run or how many times. Otherwise, debugging problems becomes very tedious, and it's more difficult to make conclusions from results. Of course, many projects have a stochastic element – but this should be controlled by a random number generator (RNG) seed, which is part of the input. You vary the seed and repeat the program, then reason about the distribution of results.

Aside from organizing your code to let a single RNG seed influence everything, another possible source of non-determinism is iteration order. In Rust, a `HashMap` could have different order every time it's used, so we use a `BTreeMap` instead when this matters. In Python, dictionaries are ordered. Be sure to check for your language.

10.2 Protocol buffers

SPC uses protocol buffers for output. This has some advantages explained the "explicit data schema" section above, but the particular choice of protocol buffer has some limitations.

First, proto3 doesn't support required fields. This is done to allow schemas to evolve better over time, but this isn't a feature SPC makes use of. There's no need to have new code work with old data, or vice versa – if the schema is updated, downstream code should adapt accordingly and use the updated input files. The lack of required fields leads to imprecise code – a person's health structure is always filled out, but in Rust, we wind up with `Option<Health>`. Differentiating 0 from missing data also becomes impossible – `urn` is optional, but in protobuf, we're forced to map the missing case to 0 and document this.

Second, protocol buffers don't easily support type-safe wrappers around numeric IDs, so downstream code has to be careful not to mix up household, venue, and person IDs.

Third, protocol buffers support limited key types for maps. Enumerations can't be used, so we use the numeric value for the activity enum.

We'll evaluate flatbuffers and other alternative encodings.

Note that in any case, SPC internally doesn't use the auto-generated code until the very end of the pipeline. It's always possible to be more precise with native Rust types, and convert to the less strict types last.

10.3 An example of the power of static type checking

Imagine we want to add a new activity type to represent people going to university and higher education. SPC already has activities for primary and secondary school, so we'll probably want to follow those as a guide. In any language, we could search the codebase for relevant terms to get a sense of what to update. In languages like Python without an up-front compilation step, if we fail to update something or write blatantly incorrect code (such as making a typo in variable names or passing a list where a string was expected), we only find out when that code happens to run. In pipelines with many steps and large input files, it could be a while before we reach the problematic code.

Let's walk through the same exercise for SPC's Rust code. We start by adding a new `University` case to the Activity enum. If we try to compile the code here (with `cargo check` or an IDE), we immediately get 4 errors.

```
error[E0004]: non-exhaustive patterns: `University` not covered
   --> src/init/quant.rs:38:44
    |
38  |       let (population_csv, prob_sij) = match activity {
    |                                         ^^^^^^^^ pattern `University` not covered
    |
   ::: src/lib.rs:129:1
    |
129 | / pub enum Activity {
130 | |     Retail,
131 | |     PrimarySchool,
132 | |     SecondarySchool,
...   |
135 | |     University,
    | |     ---------- not covered
136 | | }
    | |_- `Activity` defined here
    |
    = help: ensure that all possible cases are being handled, possibly by adding wildcards or more
match arms
    = note: the matched value is of type `Activity`
```

Three of the errors are in the QUANT module. The first is here. It's immediately clear that for retail and primary/secondary school, we read in two files from QUANT representing venues where these activities take place and the probability of going to each venue. Even if we were unfamiliar with this codebase, the compiler has told us one thing we'll need to figure out, and where to wire it up.

```
error[E0004]: non-exhaustive patterns: `University` not covered
   --> src/protobuf.rs:135:11
    |
135 |       match activity {
    |             ^^^^^^^^ pattern `University` not covered
    |
   ::: src/lib.rs:129:1
    |
129 | / pub enum Activity {
130 | |     Retail,
131 | |     PrimarySchool,
132 | |     SecondarySchool,
...   |
135 | |     University,
    | |     ---------- not covered
136 | | }
    | |_- `Activity` defined here
    |
    = help: ensure that all possible cases are being handled, possibly by adding wildcards or more
match arms
    = note: the matched value is of type `Activity`
```

The other error is in the code that writes the protobuf output. Similarly, we need a way to represent university activities in the protobuf scheme.

Extending an unfamiliar code-base backed by compiler errors is a very guided experience. If you wanted to add more demographic attributes to people or energy use information to households,

you don't need to guess all of the places in the code you'll need to update. You can just add the field, then let the compiler tell you all places where those objects get created.

11 Performance

The following tables summarizes the resources SPC needs to run in different areas.

study_area	num_msoas	num_households	num_people	pbf_file_size	runtime	commuting_runtime	memory_usage
bedfordshire	74	271,487	650,950	111.88MiB	9 seconds	5 seconds	343.79MiB
berkshire	107	363,653	878,045	150.47MiB	11 seconds	6 seconds	353.77MiB
bristol	55	196,230	456,532	80.66MiB	5 seconds	1 second	178.53MiB
buckinghamshire	99	324,843	759,879	129.01MiB	9 seconds	4 seconds	349.16MiB
cambridgeshire	98	346,532	834,141	142.28MiB	11 seconds	5 seconds	353.06MiB
cheshire	139	463,106	1,040,634	178.77MiB	12 seconds	5 seconds	361.23MiB
cornwall	74	246,873	564,604	98.35MiB	6 seconds	2 seconds	326.03MiB
cumbria	64	224,779	485,035	83.46MiB	6 seconds	2 seconds	179.53MiB
derbyshire	131	457,791	1,024,952	175.94MiB	14 seconds	7 seconds	360.48MiB
devon	156	521,790	1,178,315	202.97MiB	16 seconds	7 seconds	654.43MiB
dorset	95	344,246	751,334	129.13MiB	9 seconds	3 seconds	350.63MiB
durham	117	406,164	904,785	154.56MiB	8 seconds	3 seconds	353.70MiB
east_sussex	102	380,180	830,761	142.07MiB	9 seconds	4 seconds	352.62MiB
east_yorkshire_with_hull	75	261,267	579,746	100.62MiB	6 seconds	2 seconds	327.72MiB
essex	211	771,734	1,798,893	308.22MiB	28 seconds	17 seconds	708.37MiB

study_area	num_msoas	num_households	num_people	pbf_file_size	runtime	commuting_runtime	memory_usage
gloucestershire	107	392,120	901,395	153.74MiB	11 seconds	4 seconds	356.12MiB
greater_london	983	3,135,814	8,672,103	1.46GiB	8 minutes	7 minutes	5.07GiB
greater_manchester	346	871,651	2,746,858	462.05MiB	75 seconds	60 seconds	1.28GiB
hampshire	225	775,203	1,803,991	310.31MiB	32 seconds	21 seconds	709.56MiB
herefordshire	23	83,115	191,282	32.85MiB	3 seconds	1 second	87.82MiB
hertfordshire	153	492,783	1,144,974	195.70MiB	15 seconds	8 seconds	653.54MiB
isle_of_wight	18	64,602	135,125	24.05MiB	2 seconds	1 second	82.18MiB
kent	220	692,896	1,808,206	307.15MiB	24 seconds	13 seconds	707.24MiB
lancashire	191	597,317	1,472,550	249.28MiB	18 seconds	9 seconds	694.72MiB
leicestershire	120	417,621	1,043,283	175.53MiB	14 seconds	8 seconds	358.30MiB
lincolnshire	134	434,060	1,064,174	180.30MiB	11 seconds	5 seconds	647.65MiB
merseyside	184	546,791	1,401,012	236.74MiB	20 seconds	12 seconds	692.18MiB
norfolk	110	379,188	891,006	153.48MiB	9 seconds	3 seconds	356.10MiB
north_yorkshire	138	434,489	1,069,514	182.58MiB	12 seconds	5 seconds	648.94MiB
northamptonshire	91	293,580	733,190	125.64MiB	8 seconds	3 seconds	348.52MiB
northumberland	40	113,436	316,618	53.46MiB	4 seconds	1 second	164.67MiB
northwest_transpennine	829	1,919,598	6,419,933	1.06GiB	5 minutes	4 minutes	2.60GiB
nottinghamshire	138	413,097	1,139,096	193.89MiB	15 seconds	8 seconds	650.85MiB
oxford_cambridge_arc	353	1,152,245	2,823,838	484.18MiB	77 seconds	58 seconds	1.36GiB
oxfordshire	86	254,974	669,237	114.13MiB	8 seconds	3 seconds	329.91MiB

study_area	num_msoas	num_households	num_people	pb_file_size	runtime	commuting_runtime	memory_usage
rutland	5	16,688	39,475	6.57MiB	2 seconds	1 second	21.78MiB
shropshire	62	153,284	497,064	83.62MiB	5 seconds	2 seconds	177.97MiB
somerset	124	384,165	944,394	162.97MiB	12 seconds	5 seconds	357.18MiB
south_yorkshire	172	358,717	1,373,401	227.91MiB	20 seconds	13 seconds	655.11MiB
staffordshire	143	439,176	1,104,925	186.74MiB	12 seconds	5 seconds	649.64MiB
suffolk	90	326,760	739,296	126.15MiB	8 seconds	3 seconds	348.31MiB
surrey	151	461,466	1,136,090	194.14MiB	17 seconds	10 seconds	653.18MiB
tyne_and_wear	145	414,128	1,111,239	186.80MiB	11 seconds	5 seconds	648.22MiB
warwickshire	108	319,511	933,391	157.07MiB	13 seconds	7 seconds	354.84MiB
west_midlands	314	523,264	2,475,918	413.02MiB	31 seconds	19 seconds	1.24GiB
west_sussex	100	373,326	838,440	143.98MiB	9 seconds	3 seconds	353.52MiB
west_yorkshire	299	501,156	2,272,063	380.46MiB	32 seconds	21 seconds	1.23GiB
wiltshire	89	305,679	686,963	118.65MiB	7 seconds	2 seconds	346.00MiB
worcestershire	85	254,383	572,751	98.94MiB	8 seconds	4 seconds	326.61MiB

Notes:

- `pb_file_size` refers to the size of the uncompressed protobuf file in `data/output/`
- The total `runtime` is usually dominated by matching workers to businesses, so `commuting_runtime` gives a breakdown
- Measuring memory usage of Linux processes isn't straightforward, so `memory_usage` should just be a guide
- These measurements were all taken on one developer's laptop, and they don't represent multiple runs. This table just aims to give a general sense of how long running takes.
  - That machine has 16 cores, which matters for the parallelized commuting calculation.

- `scripts/collect_stats.py` produces the table above