Geocoding Addresses

In this section, we'll parse the addresses from the survey using the postmastr package for geocoding.

We first create a row.id column to merge our postmastr results back with the survey dataset.

```
survey_recode_df <- survey_recode_df %>%
  dplyr::mutate(row.id = dplyr::row_number())
```

Process Adress Columns using Postmastr

Our survey data set has 3 columns with address information: 1. Addr_ZIP: Postal Codes 2. Addr_CityState : City and State 3. Addr_StreetLocation: Street Names

Postal Codes

Process Postal Code Column

We first explore the postal codes in Addr_ZIP.

```
survey_addr_zipcodes <- survey_recode_df %>%
  dplyr::select(row.id, Addr_ZIP)

postmastr_zip_id <- postmastr::pm_identify(survey_addr_zipcodes, var = "Addr_ZIP")
postmastr_zip_prep <- postmastr::pm_prep(postmastr_zip_id, var = "Addr_ZIP", type = "zip")
```

Our postmastr object returns FALSE when asked if all ZIP codes in the survey data set are valid. This requires additional exploration.

```
zip_detect <- postmastr::pm_postal_detect(postmastr_zip_prep) %>%
  dplyr::filter(pm.hasZip == FALSE)
```

pm_postal_detect() reveals that the survey responses without a valid ZIP code have the following values: -99, which indicate missingness. We can thus remove them from our postal code data set.

```
survey_addr_zipcodes <- survey_recode_df %>%
  dplyr::select(row.id, Addr_ZIP) %>%
  dplyr::filter(! Addr_ZIP %in% c(NA, "-99"))

postmastr_zip_id <- postmastr::pm_identify(survey_addr_zipcodes, var = "Addr_ZIP")
```

```
postmastr_zip_prep <- postmastr::pm_prep(postmastr_zip_id, var = "Addr_ZIP", type = "zip")
```

Our postmastr object now returns TRUE when asked if all ZIP codes in the survey data set are valid.

Parsing Zip Codes

We can now parse and process all postal codes. After parsing, we left join our results with the original dataset to keep row.ids.

```
postmastr_zip_parsed <- postmastr::pm_postal_parse(postmastr_zip_prep)
```

```
Warning: There was 1 warning in `dplyr::mutate()`.
i In argument: `pm.address = ifelse(...)`.
Caused by warning in `stri_sub()`:
! argument is not an atomic vector; coercing
```

```
zip_parsed <- postmastr_zip_id %>%
  tidylog::left_join(postmastr_zip_parsed) %>%
  dplyr::select(row.id, pm.zip, pm.zip4)
```

```
Joining with `by = join_by(pm.uid)`
left_join: added 3 columns (pm.address, pm.zip, pm.zip4)
> rows only in x 0
> rows only in postmastr_zip_parsed ( 0)
> matched rows 760
> =====
> rows total 760
```

City and State

We can now repeat the process for Addr_CityState.

Processing Data

```
statedict <- postmastr::pm_dictionary(type = "state")

survey_addr_citystate <- survey_recode_df %>%
  dplyr::select(row.id, Addr_CityState) %>%
  dplyr::filter(!Addr_CityState %in% c(NA, "-99")) %>%
  dplyr::mutate(
    Addr_CityState = gsub("\\.", "", Addr_CityState),
    Addr_CityState = gsub("Falls ChurchVA 22042", "Falls Church, VA", Addr_CityState),
    Addr_CityState = gsub("Sonoma cA", "Sonoma, CA", Addr_CityState)
  )

postmastr_citystate_id <- postmastr::pm_identify(survey_addr_citystate, var = "Addr_CitySt
postmastr_citystate_prep <- postmastr::pm_prep(postmastr_citystate_id, var = "Addr_CitySta
```

Our postmastr object now returns FALSE when asked if all states in the Addr_CityState column are valid.

We next examine the rows without a valid state.

```
pm_state_none(postmastr_citystate_prep)
```

```
# A tibble: 291 x 2
   pm.uid pm.address
    <int> <chr>
 1      1 Indianapolis
 2      2 Willowbrook
 3      6 Lincoln
 4      8 Lodi
 5     10 Largo Fl
 6     11 Jeffersonville
 7     12 Atlanta
 8     14 Norfolk
 9     15 OSSINING
10     20 Miami
# i 281 more rows
```

This data set contains some title case state abbreviations that are undetected. We can append them to our state dictionary. We can then check the unmatched variables to double-check if any states were missed.

```r
states_append <- postmastr::pm_append(
  type = "state",
  input = c(
    "Fl",
    "North Caroling",
    "Tx",
    "Ca",
    "Falls ChurchVA 22042",
    "GA 30014",
    "NV.",
    "Ky",
    "Ma",
    "CO.",
    "OaklandCA",
    "Mi",
    "cA",
    "Ny"
  ),
  output = c(
    "FL",
    "NC",
    "TX",
    "CA",
    "VA",
    "GA",
    "NV",
    "KY",
    "MA",
    "CO",
    "CA",
    "MI",
    "CA",
    "NY"
  ),
  locale = "us"
)

statedict <- postmastr::pm_dictionary(type = "state", append = states_append)

pm_state_none(postmastr_citystate_prep, dictionary = statedict)
```

```
# A tibble: 278 x 2
```

```
   pm.uid pm.address
    <int> <chr>
 1      1 Indianapolis
 2      2 Willowbrook
 3      6 Lincoln
 4      8 Lodi
 5     11 Jeffersonville
 6     12 Atlanta
 7     14 Norfolk
 8     15 OSSINING
 9     20 Miami
10     22 Portland
# i 268 more rows
```

Parsing State Data

Once processing is complete, we can parse States from the Addr_CityState column.

```
postmastr_state_parsed <- postmastr::pm_state_parse(postmastr_citystate_prep,
                                                    dictionary = statedict)
```

```
Warning: There was 1 warning in `dplyr::mutate()`.
i In argument: `pm.address = ifelse(...)`.
Caused by warning in `stri_sub()`:
! argument is not an atomic vector; coercing
```

```
postmastr_state_parsed
```

```
# A tibble: 642 x 3
   pm.uid pm.address   pm.state
    <int> <chr>        <chr>
 1      1 Indianapolis <NA>
 2      2 Willowbrook  <NA>
 3      3 Springfield  MA
 4      4 Chicago      IL
 5      5 Selinsgrove  PA
 6      6 Lincoln      <NA>
 7      7 Dayton       OH
 8      8 Lodi         <NA>
 9      9 Harrison     ME
```

```
10     10 Largo        FL
# i 632 more rows
```

Processing City Data

We repeat the same process for city data. We include a filter for the states identified after parsing with Postmastr.

```
states <- unique(postmastr_state_parsed$pm.state)[!is.na(unique(postmastr_state_parsed$pm.

citydict <- postmastr::pm_dictionary(type = "city", filter = states)

pm_city_none(postmastr_state_parsed, dictionary = citydict)
```

```
# A tibble: 50 x 3
   pm.uid pm.address     pm.state
    <int> <chr>          <chr>
 1      1 Indianapolis   <NA>
 2      5 Selinsgrove    PA
 3     25 St Louis       MO
 4     32 Westbrookville <NA>
 5     55 Honolulu       <NA>
 6     75 Dillsburg      <NA>
 7     89 AHMEDABAD      <NA>
 8     96 Carrolltown    <NA>
 9    120 Chestnut Hill  MA
10    128 Newfoundland   PA
# i 40 more rows
```

After processing state names, our postmastr object contains sum unmatched cities. We create an additional city-level dictionary to append to our current city-level dictionary. We coerce valid city names to NA values to let postmastr know that these inputs are valid.

```
city_append <- postmastr::pm_append(
  type = "city",
  input = c(
    "Indianapolis",
    "Selinsgrove",
    "Birmingham.",
    "Westbrookville",
    "Honolulu",
```

```
"Dillsburg",
"Carrolltown",
"Chestnut Hill",
"Newfoundland",
"Mercersburg",
"Moultonborough",
"Hanibal",
"OKC",
"Saint Clair Shores",
"Mt Pleasant",
"Boise",
"idyllwild",
"Weehawken",
"Appleon",
"Pacoima",
"SEattle",
"West Palm beach",
"Chelmsford",
"Metuchen",
"Marietta.",
"Sewickley",
"Falls ChurchVA",
"Trout Run",
"JUD",
"OLEMA",
"Danielson",
"North Las Vegas",
"Braintree",
"Lititz",
"benesnville",
"OaklandCA",
"Phoenixville",
"Denver",
"Van Nuys",
"Jacvksonville",
"Bronx",
"Pratts",
"Baltimore",
"Sonoma",
"Williston Pk",
"Wernersville",
```

```
  "St Louis",
  "La Canada Flintridge",
  "Falls Church"
),
output = c(
  NA,
  NA,
  "Birmingham",
  NA,
  NA,
  NA,
  NA,
  NA,
  NA,
  NA,
  NA,
  NA,
  "Oklahoma City",
  NA,
  "Mount Pleasant",
  NA,
  "Idyllwild",
  NA,
  "Appleton",
  NA,
  "Seattle",
  "West Palm Beach",
  NA,
  NA,
  "Marietta",
  "Sewickley",
  "Falls Church",
  NA,
  "Jud",
  "Olema",
  NA,
  "North Las Vegas",
  NA,
  NA,
  "Bensenville",
  "Oakland",
```

```
    NA,
    NA,
    NA,
    NA,
    "The Bronx",
    "Pratt",
    "Baltimore",
    NA,
    "Williston Park",
    NA,
    "St. Louis",
    NA,
    "Falls Church"
  )
)

citydict <-
  postmastr::pm_dictionary(type = "city",
                           filter = states,
                           append = city_append)

pm_city_none(postmastr_state_parsed, dictionary = citydict)
```

```
# A tibble: 5 x 3
  pm.uid pm.address                pm.state
   <int> <chr>                     <chr>
1     89 AHMEDABAD                 <NA>
2    382 Falls ChurchVA 22042      <NA>
3    486 Tay Ho Ha Noi             <NA>
4    491 Bangalore Karnataka India <NA>
5    595 Sonoma cA                 <NA>
```

The unmatched cities in our data set are not from the US and will be ignored.

```
postmastr_citystate_parsed <- postmastr::pm_city_parse(postmastr_state_parsed, dictionary
```

```
Warning: There was 1 warning in `dplyr::mutate()`.
i In argument: `pm.address = ifelse(...)`.
Caused by warning in `stri_sub()`:
! argument is not an atomic vector; coercing
```

```
  citystate_parsed <- postmastr_citystate_parsed %>%
    tidylog::left_join(postmastr_citystate_id) %>%
    dplyr::select(row.id, pm.city, pm.state)
```

```
Joining with `by = join_by(pm.uid)`
left_join: added 4 columns (pm.id, pm.type, row.id, Addr_CityState)
> rows only in x 0
> rows only in postmastr_citystate_id ( 0)
> matched rows 765 (includes duplicates)
> =====
> rows total 765
```

Street Names

Prepare Data

The final column to process is Addr_StreetLocation which contains street names.

```
  survey_addr_street <- survey_recode_df %>%
    dplyr::select(row.id, Addr_StreetLocation) %>%
    dplyr::filter(! Addr_StreetLocation %in% c("-99", NA))

  postmastr_street_id <- postmastr::pm_identify(survey_addr_street, var = "Addr_StreetLocati
  postmastr_street_prep <- postmastr::pm_prep(postmastr_street_id, var = "Addr_StreetLocatio

  head(postmastr_street_prep, 10)
```

```
# A tibble: 10 x 2
   pm.uid pm.address
    <int> <chr>
 1      1 6130 N Michigan Rd
 2      2 6141 Bentley Ave
 3      3 38 Oxford Street
 4      4 2545 W Diversey Ave Suite 225
 5      5 429 Eighth Street
 6      6 912 N. 70th Street
 7      7 409 Troy St
 8      8 275 Poplar Street
 9      9 156 Deertrees Rd
10     10 12552 Belcher Rd S
```

Parse House Numbers

Next, we extract house/unit numbers from the street addresses.

```
postmastr_house_parsed <- postmastr::pm_house_parse(postmastr_street_prep)
head(postmastr_house_parsed, 10)
```

```
# A tibble: 10 x 3
   pm.uid pm.address                pm.house
    <int> <chr>                     <chr>
 1      1 N Michigan Rd             6130
 2      2 Bentley Ave               6141
 3      3 Oxford Street             38
 4      4 W Diversey Ave Suite 225 2545
 5      5 Eighth Street             429
 6      6 N. 70th Street            912
 7      7 Troy St                   409
 8      8 Poplar Street             275
 9      9 Deertrees Rd              156
10     10 Belcher Rd S              12552
```

Parse Street Prefix and Suffix

To parse street prefixes and suffixes requires 2 dictionaries. 1 with directions and 1 with conversions for suffixes like "street" or "boulevard".

```
dirs <- pm_dictionary(type = "directional", locale = "us")
postmastr_streetdir_parsed <- postmastr::pm_streetDir_parse(postmastr_house_parsed, dictio
postmastr_streetSuf_parsed <- postmastr::pm_streetSuf_parse(postmastr_streetdir_parsed)
```

```
Warning: There was 1 warning in `dplyr::mutate()`.
i In argument: `pm.address = ifelse(...)`.
Caused by warning in `stri_sub()`:
! argument is not an atomic vector; coercing
```

Parse Street Names

The final element for parsing is the street names themselves.

```r
postmaster_street_parsed <- postmastr::pm_street_parse(postmastr_streetSuf_parsed,
                                                       ordinal = TRUE,
                                                       drop = TRUE)
postmaster_street_parsed
```

```
# A tibble: 756 x 6
   pm.uid pm.house pm.preDir pm.street             pm.streetSuf pm.sufDir
    <int> <chr>    <chr>     <chr>                 <chr>        <chr>
 1      1 6130     N         Michigan              Rd           <NA>
 2      2 6141     <NA>      Bentley               Ave          <NA>
 3      3 38       <NA>      Oxford                St           <NA>
 4      4 2545     W         Diversey Ave Suite 225 <NA>         <NA>
 5      5 429      <NA>      8th                   St           <NA>
 6      6 912      N         70th                  St           <NA>
 7      7 409      <NA>      Troy                  St           <NA>
 8      8 275      <NA>      Poplar                St           <NA>
 9      9 156      <NA>      Deertrees             Rd           <NA>
10     10 12552    <NA>      Belcher               Rd           S
# i 746 more rows
```

Combine Street Components Into Single Column

After parsing each component, we concatenate them into a single street address.

```r
postmastr_street_full_parsed <- postmastr::pm_replace(postmaster_street_parsed,
                                                      source = postmastr_street_id)
endQ <- rlang::quo(!! rlang::sym("pm.sufDir"))

postmastr_street_full_parsed <- postmastr::pm_rebuild(postmastr_street_full_parsed,
                                                      output = "short",
                                                      keep_ids = TRUE)

street_parsed <- postmastr_street_full_parsed %>%
  tidylog::left_join(postmastr_street_id) %>%
  dplyr::select(row.id, pm.address)
```

```
Joining with `by = join_by(pm.id, pm.uid, pm.type, row.id,
Addr_StreetLocation)`
left_join: added no columns
> rows only in x 0
```

```
> rows only in postmastr_street_id ( 0)
> matched rows 766
> =====
> rows total 766
```

Combine All Parsed Addresses Components Together

And finally, we concatenate our postal code, city, state and street address into a single f_address column for geocoding.

```
postmastr_faddress_parsed <- street_parsed %>%
  tidylog::left_join(citystate_parsed) %>%
  tidylog::left_join(zip_parsed) %>%
  tidyr::unite(f_address, pm.address, pm.city, pm.state, pm.zip, pm.zip4, na.rm = TRUE, se
```

```
Joining with `by = join_by(row.id)`
left_join: added 2 columns (pm.city, pm.state)
> rows only in x 3
> rows only in citystate_parsed ( 2)
> matched rows 763
> =====
> rows total 766
Joining with `by = join_by(row.id)`
left_join: added 2 columns (pm.zip, pm.zip4)
> rows only in x 8
> rows only in zip_parsed ( 2)
> matched rows 758
> =====
> rows total 766
```

```
head(postmastr_faddress_parsed, 20)
```

```
# A tibble: 20 x 2
   row.id f_address
    <int> <chr>
 1      2 6130 N Michigan Rd,Indianapolis,46228
 2      3 6141 Bentley Ave,Willowbrook,60527
 3      4 38 Oxford St,Springfield,MA,01108
 4      5 2545 W Diversey Ave Suite 225,Chicago,IL,60647
 5      6 429 8th St,Selinsgrove,PA,17870
```

```
 6      7 912 N 70th St,Lincoln,68505
 7      8 409 Troy St,Dayton,OH,45404
 8      9 275 Poplar St,Lodi,95240
 9     10 156 Deertrees Rd,Harrison,ME,04040
10     11 12552 Belcher Rd S,Largo,FL,33773
11     12 118 E Chestnut Street Po Box 886,Jeffersonville,47130
12     13 1297 Jonesboro Rd SE,Atlanta,30315
13     14 301 Front St,Key West,FL,33040
14     15 150 St Paul's Blvd. 5th Floor,Norfolk,23510
15     16 138 Spring St,OSSINING,10562
16     17 15 Saunders Way #500d,Westbrook,ME,04092
17     18 3100 Independence Dr,Birmingham,AL,35209
18     19 16173 Baseline Rd,Genoa,IL,60135
19     20 2100 Sherman Ave Suite 300,Cincinnati,OH,45212
20     21 2800 Biscayne Blvd Suite 300,Miami,33137
```

The f_address column now has parsed addresses that can be formatted using a geocoder of choice. We can now rejoin the parsed address data via row.id from our original data set.

```
survey_geoproc_df <- survey_recode_df %>%
  tidylog::left_join(postmastr_faddress_parsed, by = "row.id")
```

```
left_join: added one column (f_address)
           > rows only in x                          56
           > rows only in postmastr_faddress_parsed (  0)
           > matched rows                           766
           >                                       =====
           > rows total                             822
```

Geocoding

Passing Parsed Addresses to Geocoder

This parsed data set can be submitted to the geocoder of choice. In this case, the survey was processed using the Urban Institue's Internal Geocoder.

Reading Geocoder Outputs

The outputs from the geocoding process are presented below.

```
# A tibble: 766 x 61
   row.id f_address  Match_addr Longitude Latitude Addr_type Score Status Region
    <dbl> <chr>      <chr>          <dbl>    <dbl> <chr>     <dbl> <chr>  <chr>
 1      2 6130 N Mi~ 6130 N Mi~     -86.2     39.9 StreetAd~   100 M      India~
 2      3 6141 Bent~ 6141 Bent~     -88.0     41.8 PointAdd~   100 M      Illin~
 3      4 38 Oxford~ 38 Oxford~     -72.6     42.1 PointAdd~   100 M      Massa~
 4      5 2545 W Di~ 2545 W Di~     -87.7     41.9 Subaddre~   100 M      Illin~
 5      6 429 8th S~ 429 8th S~     -76.9     40.8 PointAdd~   100 M      Penns~
 6      7 912 N 70t~ 912 N 70t~     -96.6     40.8 StreetAd~   100 M      Nebra~
 7      8 409 Troy ~ 409 Troy ~     -84.2     39.8 PointAdd~   100 M      Ohio
 8      9 275 Popla~ 275 Popla~    -121.      38.1 StreetAd~   100 M      Calif~
 9     10 156 Deert~ 156 Deert~     -70.7     44.1 PointAdd~   100 M      Maine
10     11 12552 Bel~ 12552 Bel~     -82.7     27.9 PointAdd~   100 M      Flori~
# i 756 more rows
# i 52 more variables: RegionAbbr <chr>, Subregion <chr>, MetroArea <chr>,
#   City <chr>, Nbrhd <chr>, geometry <chr>, Match_type <chr>, LongLabel <chr>,
#   ShortLabel <chr>, Type <chr>, PlaceName <chr>, Place_addr <chr>,
#   Phone <chr>, URL <lgl>, Rank <dbl>, AddBldg <lgl>, AddNum <chr>,
#   AddNumFrom <dbl>, AddNumTo <dbl>, AddRange <chr>, Side <chr>,
#   StPreDir <chr>, StPreType <chr>, StName <chr>, StType <chr>, ...
```

We do not need all geographic information returned from the geocoder. The variables of interest are listed in the table below.

Variable	Description
row.id	Row number from original data set for merging
Region	State name
RegionAbbr	State abbreviation
Subregion	County name
City	City name
Latitude	10-digit Latitude
Longitude	10-digit Longitude

We can also pre-emptively drop geographic variables with personally identifiable information such as the postalcode and street address.

```r
# Define geocoded variables of interest
geocode_cols <- c(
  "row.id",
  "Region",
  "RegionAbbr",
```

```
  "Subregion",
  "City",
  "Latitude",
  "Longitude"
)

# Subset the geocoded data set
postmastr_geocoded_subset <- postmastr_geocoded %>%
  dplyr::select(
    tidyselect::all_of(geocode_cols)
  )

# Merge those results via row.id with the processed survey data
survey_geocoded_df <- survey_recode_df %>%
  tidylog::left_join(postmastr_geocoded_subset,
                     by = "row.id")
```

```
left_join: added 6 columns (Region, RegionAbbr, Subregion, City, Latitude, …)
           > rows only in x                              56
           > rows only in postmastr_geocoded_subset (  0)
           > matched rows                               766
           >                                           =====
           > rows total                                 822
```

```
# Define geographic variables to exclude from processed data set
geo_cols_to_exclude <- c(
  "address",
  "city",
  "state",
  "zip5",
  "zipcode",
  "Addr_StreetLocation",
  "Addr_CityState",
  "Addr_ZIP",
  "row.id"
)

# Exclude Columns and Rename Existing Geographic Columns
survey_geocoded_df <- survey_geocoded_df %>%
  dplyr::select(
    ! tidyselect::all_of(geo_cols_to_exclude)
```

```
  ) %>%
  dplyr::rename(
    "State" = "Region",
    "StateAbbr" = "RegionAbbr",
    "County" = "Subregion"
  )
```

Saving Geocoded Outputs

We can now write these outputs to disk and move on to the next chapter, where we add census level metadata to our survey responses.

```
setwd("Y:/CNP/Generosity Commission/")
readr::write_csv(
  survey_geocoded_df,
  "DATA-PREP/02-data-intermediate/02-wave-two/wave-02-data-intermediate-geocoded.csv"
)
```