Appendix 1.B: Analysing global trends in within-country inequality

```
import pandas as pd
import numpy as np
from plotnine import *
```

Load in the data

```
# df_wid = pd.read_csv("data/clean/wid.csv")
# df_pip = pd.read_csv("data/clean/pip.csv")
```

```
df_wid = pd.read_csv("PhD_pages/data_appendices/data/clean/wid.csv")
df_pip = pd.read_csv("PhD_pages/data_appendices/data/clean/pip.csv")
```

A key difficulty of comparing trends is that the data is incomplete coverage. The paper uses two methods to calculate and compare aggregate within-country trends across incomplete data: using reference years and using year fixed effects in a regression.

Method 1: Reference years

I need to allow for the income/consumption issue (currently the data includes both – i.e. multi observations per country year).

Mapping data to a reference year

This function grabs the observation, by a grouping variable (e.g. by country), for which a reference variable (e.g. year) is closest to a reference value (e.g. 2002). You can specify a maximum distance from the reference value, beyond which no match will be returned. Because there may be tie-breaks – matches equally distant above or below the reference value – there is an argmuent to specify how these tie-breaks should resolved.

```
#  Function get matching for ref years
def closest_to_reference(df, reference_val, max_dist_from_ref, reference_col, group_by_col

    df = df.loc[:, [reference_col, group_by_col, value_col]]

    # Drop NAs
    df = df.dropna()

    # Calculate absolute distance from reference value
    df['ref_diff'] = abs(df[reference_col] - reference_val)

    # Drop any rows with a distance beyond threshold
    if not pd.isna(max_dist_from_ref):
      df = df.loc[df['ref_diff'] <= max_dist_from_ref]

    # Keep closest observation to reference value - including tie-breaks (where there is a
    df = df[df.groupby(group_by_col)['ref_diff'].transform('min') == df['ref_diff']].reset

    # Settle tie-breaks
    if tie_break == 'below':
      df = df[df.groupby(group_by_col)[reference_col].transform('min') == df[reference_col

    elif tie_break == 'above':
      df = df[df.groupby(group_by_col)[reference_col].transform('max') == df[reference_col

    df = df.drop('ref_diff', axis=1)

    return df
```

I test this function here:

```
test = closest_to_reference(
  df = df_wid,
  reference_val = 2002,
  max_dist_from_ref = 5,
  reference_col = 'Year',
  group_by_col = 'Entity',
  value_col = 'Gini',
  tie_break = 'below'
  )

test.head()
```

	Year	Entity	Gini
0	1998	United Arab Emirates	0.656802
1	2002	Albania	0.481698
2	2002	Armenia	0.535265
3	2000	Angola	0.682466
4	2002	Argentina	0.666745

Obtaining a pair of observations

This function runs the `closest_to_reference` function twice, over a pair of reference values. Tie-breaks are settled so as to maximise the gap between the two reference values. You can also specify a minimum distance between the two observations (e.g. pairs of matches that fall less than X years apart will be dropped).

```
# Merge matches for different reference points
def merge_two_ref_matches(df, reference_vals, max_dist_from_refs, min_dist_between, refere

  # Make sure the pair of reference values are in ascending order
  reference_vals.sort()

  # Maximise distance between two refs by settling tie-breaks below the lowest ref and abo

  # Find matches for lower reference value
  lower_ref_matches = closest_to_reference(df, reference_vals[0], max_dist_from_refs[0], r

  # Find matches for higher reference value
  higher_ref_matches = closest_to_reference(df, reference_vals[1], max_dist_from_refs[1],

  # Merge the two sets of matches
  merged_df = pd.merge(lower_ref_matches, higher_ref_matches, on=group_by_col, suffixes=(r

  # Drop obs that do not have data for both ref values
  merged_df = merged_df.dropna()

  # Drop obs where the matched data does not meet the min distance requirement
  if not pd.isna(min_dist_between):

    # Store the names of the reference column returned from the two matches
    ref_var_high = f'{reference_col}{reference_vals[1]}'
    ref_var_low = f'{reference_col}{reference_vals[0]}'
```

```
    # Keep only rows >= to the min distance
    merged_df = merged_df.loc[(merged_df[ref_var_high] - merged_df[ref_var_low]) >= min_di


  return merged_df
```

I test this function here:

```
test = merge_two_ref_matches(
  df = df_wid,
  reference_vals = [2000, 2010],
  max_dist_from_refs = [5, 4],
  min_dist_between = 9,
  reference_col = 'Year',
  group_by_col = 'Entity',
  value_col = 'Gini'
  )

test.head()
```

	Year2000	Entity	Gini2000	Year2010	Gini2010
0	1998	United Arab Emirates	0.656802	2009	0.676088
1	2002	Albania	0.481698	2012	0.467515
2	1999	Armenia	0.552912	2010	0.498014
4	2001	Argentina	0.660888	2010	0.570943
6	2000	Australia	0.474796	2010	0.475651

Allowing for the different welfare concepts in the PIP data

As noted in …. (add backlink) … the PIP data contains both income and consumption observations. This function produces matched pairs of observations from that data in which only one pair is selected for each value of the grouping variable (i.e. each country). Priority is given to pairs for which both observations relate to income. Second priority is given to pairs where both observations relate to consumption. Only if neither pair is available then it will return a mixed pair of observations if that is available.

```
# For PIP run this three times - first filtering data for just consumpion only, then with
def pip_welfare_routine(df, reference_vals, max_dist_from_refs, min_dist_between, referenc

  # Specify the name of the column in which the income/consumption welfare definition is s
  welfare_colname = 'welfare_type'
```

```
    # Creat dataframes for thee scenarios:
    # Scenario 1: only allow income data
    df_inc_filter = df.loc[df[welfare_colname] == "income", :]
    df_inc_filter.name = "Income"

    # Scenario 2: only allow consumption data
    df_cons_filter = df.loc[df[welfare_colname] == "consumption", :]
    df_cons_filter.name = "Consumption"
    # Scenario 3: allow a mix - dropping consumption data where income data is available in
    df_mixed = df.copy()

    df_mixed['welfare_count'] = df_mixed.groupby([reference_col, group_by_col])[welfare_coln

    df_mixed = df_mixed.loc[(df_mixed['welfare_count'] == 1) | (df_mixed[welfare_colname] ==

    df_mixed.name = "Mixed"
    #  Store the scneario dataframes in a list
    df_scenarios = [df_inc_filter, df_cons_filter, df_mixed]

    # Run the matching function on each scenario
    scenario_matches = [merge_two_ref_matches(
      df_scenario,
      reference_vals,
      max_dist_from_refs,
      min_dist_between,
      reference_col,
      group_by_col,
      value_col) for df_scenario in df_scenarios]

    # Combine the scenarios, keeping only one match where there matches in more than one sce
    df_combined_matches = pd.concat([scenario_matches[0], scenario_matches[1]], keys=[df_sce

    # Tidy up indexes
    df_combined_matches = df_combined_matches.reset_index()

    df_combined_matches = df_combined_matches.drop('level_1', axis=1)

    df_combined_matches = df_combined_matches\
      .rename(columns={"level_0": "pip_welfare"})
    return df_combined_matches
```

I test this function here:

```
test = pip_welfare_routine(

  df = df_pip,
  reference_vals = [1986, 2016],
  max_dist_from_refs = [5, 5],
  min_dist_between = 30,
  reference_col = 'Year',
  group_by_col = 'Entity',
  value_col = 'Gini'

  )

test.head()
```

	pip_welfare	Year1986	Entity	Gini1986	Year2016	Gini2016
0	Income	1986	Argentina	0.428089	2016	0.420325
1	Income	1985	Australia	0.324977	2016	0.336858
2	Income	1985	Belgium	0.252039	2016	0.275810
3	Income	1986	Brazil	0.584646	2016	0.533428
4	Income	1987	Chile	0.562102	2017	0.444410

Merging reference year aligned data from WID and PIP datasets

```
def prep_wid_pip_data(reference_vals, max_dist_from_refs, min_dist_between, reference_col,
  pip_matches = pip_welfare_routine(
    df = df_pip,
    reference_vals = reference_vals,
    max_dist_from_refs = max_dist_from_refs,
    min_dist_between = min_dist_between,
    reference_col = reference_col,
    group_by_col = group_by_col,
    value_col = value_col
  )

  wid_matches = merge_two_ref_matches(
    df = df_wid,
    reference_vals = reference_vals,
    max_dist_from_refs = max_dist_from_refs,
    min_dist_between = min_dist_between,
    reference_col = reference_col,
```

```
    group_by_col = group_by_col,
    value_col = value_col
  )

  ref_pairs = pd.concat([pip_matches, wid_matches,], keys=['pip', 'wid'])

  # Tidy up indexes
  ref_pairs = ref_pairs.reset_index()

  ref_pairs = ref_pairs.drop('level_1', axis=1)

  ref_pairs = ref_pairs\
    .rename(columns={"level_0": "source"})



  # Add a count by country - showing whether data is available from both sources or not
  ref_pairs['source_count'] = ref_pairs.groupby('Entity')['source'].transform('count')

  return ref_pairs
```

Plot

```
# Specifications
reference_vals = [1990, 2016]
max_dist_from_refs = [5, 5]
min_dist_between = 25
value_col = 'Gini'

# Set the value which constitutes a significant change in the value being measured
  # E.g. for Gini tolerance = 0.02 (2 points)
tolerance = 0.02

# Data from both sources is needed?
  # If this = 1 then a country will be incldued even if it is only availalbe from one sour
  # If = 2, then only countries with data for both sources will be included
source_count_requirement = 1

# Unlikely to change:
reference_col = 'Year'
```

```
group_by_col = 'Entity'


# Prep data

ref_pairs = prep_wid_pip_data(
  reference_vals = reference_vals,
  max_dist_from_refs = max_dist_from_refs,
  min_dist_between = min_dist_between,
  reference_col = reference_col,
  group_by_col = group_by_col,
  value_col = value_col,
)

plot_data = ref_pairs.loc[ref_pairs['source_count'] >= source_count_requirement]


# ----- Prep plot ------

# Store the names of the columns to be used onthe X and Y axis
x_axis = f'{value_col}{reference_vals[0]}'
y_axis = f'{value_col}{reference_vals[1]}'

# I grab what I am guessing to be the max and min tick marks on the x axist, in order to d
x_min = plot_data[x_axis].min()
x_min_floor = np.floor(x_min * 10) / 10

x_max = plot_data[x_axis].max()
x_max_ceiling = np.ceil(x_max * 10) / 10

# Set the coordinates of the shaded ribbon
shaded_coord = pd.DataFrame({'x': [x_min_floor, x_max_ceiling, x_max_ceiling, x_min_floor]
  'y': [x_min_floor-tolerance, x_max_ceiling-tolerance, x_max_ceiling+tolerance , x_min_fl


plot = (ggplot(plot_data
, aes(x_axis, y_axis))
 + geom_point()
 + facet_wrap('~ source')
 + geom_polygon(aes(x='x', y='y'), data=shaded_coord, fill="#FF000066"))
```

```
plot

# ---- Prep summary table ----

# Count by rise/stable/fall bins

# Calculate the change between the two ref periods
plot_data['change'] = (plot_data[y_axis] - plot_data[x_axis])


# Define the thresholds using the tolerance specified
thresholds = [-float("inf"), -tolerance, tolerance, float("inf")]

# Count observations in bins defined by the thresholds
threshold_counts = plot_data\
  .groupby(['source', pd.cut(plot_data['change'], thresholds)])\
  .size()\
  .unstack()\
  .T\

# Label the bins and set as the index
threshold_counts['summary'] = ['fall', 'stable', 'rise']
threshold_counts = threshold_counts.set_index('summary')

# Calculate average change and n
mean_and_count = pd.DataFrame(plot_data.groupby('source')['change']\
  .agg(['mean', 'count']))\
  .T

mean_and_count['summary'] = ['avg_change', 'n']

mean_and_count = mean_and_count.set_index('summary')


# Concat together and present in a table

df_summary = pd.concat([threshold_counts, mean_and_count])

df_summary
```

source summary	pip	wid
fall	23.000000	13.000000
stable	12.000000	12.000000
rise	23.000000	39.000000
avg_change	-0.011373	0.037544
n	58.000000	64.000000

Method 2: regression analysis

Explore the date

I plan to build a Shiny app to help compare trends across datasets (using Shinylive – built on Shiny for Python -).

Here is a test app just so I can test the wiring of how such an app works.

```
#| standalone: true
#| viewerHeight: 420

from shiny import *
from plotnine import *
from pyodide.http import open_url
import pandas as pd



app_ui = ui.page_fluid(
    ui.output_plot("example_plot"),
)


def server(input, output, session):
    @output
    @render.plot
    def example_plot():

        url = 'https://raw.githubusercontent.com/owid/notebooks/main/BetterDataDocs/JoeHas

        df = pd.read_csv(open_url(url))
```

```
        plot = (ggplot(df, aes('Year', 'headcount_ratio_365', color = 'Entity'))
        + geom_line())

        # d = {'col1': [1, 2], 'col2': [3, 4]}
        # df = pd.DataFrame(data=d)
        # plot = (ggplot(df, aes('col1', 'col2'))
        # + geom_point())


        return plot


app = App(app_ui, server)
```