Figures on the U.S. Income Distribution
Code Implementation

Conrad Kosowsky

This file documents `income_figs.py`, which produces figures on the U.S. income distribution using a data product from the Census Bureau. The data product is stored in a single `xlsx` file that contains 14 sheets. Those sheets are:

- `Sheet1_bin_edges_1967`—endpoints corresponding to different bins from the 1967 data
- `Sheet2_bin_freq_1967`—frequencies (proportion of total survey weight) appearing in each bin from the 1967 data
- `Sheet3_bin_edges_1995`—endpoints for 1995
- `Sheet4_bin_freq_1995`—frequencies for 1995
- `Sheet5_bin_edges_2023`—endpoints for 2023
- `Sheet6_bin_freq_2023`—frequencies for 2023
- `Sheet7_CSS_InvG_lin_constants`—constants for the constant-shift-scale inverse-gamma distribution, assuming a linear relationship between parameters
- `Sheet8_CSS_InvG_lin_parameters`—parameter estimates for the constant-shift-scale inverse-gamma distribution, assuming a linear relationship between parameters
- `Sheet9_CSS_InvG_se`—standard errors for the constant-shift-scale inverse-gamma distribution
- `Sheet10_estimates_2023`—parameter estimates for a variety of distributions using 2023 data
- `Sheet11_Fisk_parameters`—parameter estimates for the Fisk distribution for all years of data
- `Sheet12_InvG_parameters`—parameter estimates for the inverse-gamma distribution for all years of data
- `Sheet13_CSS_InvG_prop_constants`—constants for the constant-shift-scale inverse-gamma distribution, assuming a proportional relationship between parameters
- `Sheet14_CSS_InvG_prop_parameter`—parameter estimates for the constant-shift-scale inverse-gamma distribution, assuming a proportional relationship between parameters

This code for figures is the same as in `make_figures.py` and `main.py`. See also `estimate_parameters.pdf`. What's different is getting the data out of the file. The data file is called `output_request.xlsx`, and we are assuming it is in the same directory as `figs.py`.

The software documented in this file is designed to use a data product that was made using internal U.S. Census Bureau data. Any views expressed are those of the author and not those of the U.S. Census Bureau. The Census Bureau has reviewed this data product to ensure appropriate access, use, and disclosure avoidance protection of the confidential source data used to produce this product. This research was performed at a Federal Statistical Research Data Center under FSRDC Project Number 2679. (CBDRB-FY24-P2679-R11429)

1 Reading in the data

We begin by reading in the data. For the sample densities, we create one DataFrame for each of the three years with one column for midpoint of the bin and one column for size of the sample density. We begin by importing modules.

```
import pandas as pd
```

Then read in the data for each year. We will store these DataFrames in `sample_dens`.

```
sample_dens = {}
for k,v in enumerate([1967, 1995, 2023]):
  temp_edges = pd.read_excel("output_request.xlsx",
    sheet_name="Sheet{0}_bin_edges_{1}".format(2*k + 1, v),
    skiprows=2, header=0)
  temp_freq = pd.read_excel("output_request.xlsx",
    sheet_name="Sheet{0}_bin_freq_{1}".format(2*k + 2, v),
    skiprows=2, header=0)
```

Now convert the frequency to a density and store it in `sample_dens`.

```
  temp_freq["freq"] = temp_freq["freq"] / (temp_edges["right"] -
    temp_edges["left"])
  sample_dens[v] = pd.DataFrame(
    {"mid": (temp_edges["right"] + temp_edges["left"]) / 2,
     "dens": temp_freq["freq"]})
```

Now read in the constants and parameter estimates for the constant-shift-scale distribution. We store them in `CSS_InvG_lin_constants` and `CSS_InvG_lin_parameters`.

```
CSS_InvG_lin_constants = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet7_CSS_InvG_lin_constants",
  skiprows=2, header=None, names=["val"], index_col=0)
```

Constants.

```
phi_lin = CSS_InvG_lin_constants.at["phi", "val"]
psi0_lin = CSS_InvG_lin_constants.at["psi0", "val"]
psi1_lin = CSS_InvG_lin_constants.at["psi1", "val"]
psi2_lin = CSS_InvG_lin_constants.at["psi2", "val"]
```

Parameter estimates.

```
CSS_InvG_lin_parameters = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet8_CSS_InvG_lin_parameters",
  skiprows=2, header=0, index_col=0)
```

The sheet containing 2023 parameter estimates needs additional parsing because the format is irregular. We store the parameter estimates in a dictionary called `parameters`. The keys are strings corresponding to the different distributions.

```
parameters = {}
temp = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet10_estimates_2023",
  skiprows=2, header=None)
```

The function `pull_from_params` takes two arguments. One is a row number, and one is a number of values. It pulls out a number of entries from `temp` that are all in the specified row by starting at column 1 and moving right.

```
def pull_from_params(row, num):
  return [temp.at[row,1 + i] for i in range(num)]
parameters["Dagum"] = pull_from_params(2,4)
parameters["Burr"] = pull_from_params(5,4)
parameters["Fisk"] = pull_from_params(8, 3)
parameters["InvG"] = pull_from_params(11, 3)
parameters["Davis"] = pull_from_params(14, 3)
parameters["LogN_P_cut"] = pull_from_params(17, 6)
parameters["GB2"] = pull_from_params(20, 5)
parameters["LogN_P_mix"] = pull_from_params(23, 6)
```

We store the Fisk parameter estimates in `Fisk_parameters`.

```
Fisk_parameters = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet11_Fisk_parameters",
  skiprows=2, header=0, index_col=0)
```

Same with inverse-gamma parameter estimates.

```
InvG_parameters = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet12_InvG_parameters",
  skiprows=2, header=0, index_col=0)
```

For the constant-shift-scale inverse-gamma model with proportional relationship between parameters, we load the data the same way as previously with the linear version.

```
CSS_InvG_prop_constants = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet13_CSS_InvG_prop_constants",
  skiprows=2, header=None, names=["val"], index_col=0)
phi_prop = CSS_InvG_prop_constants.at["phi", "val"]
psi0_prop = CSS_InvG_prop_constants.at["psi0", "val"]
psi1_prop = CSS_InvG_prop_constants.at["psi1", "val"]
CSS_InvG_prop_parameters = pd.read_excel("output_request.xlsx",
  sheet_name = "Sheet14_CSS_InvG_prop_parameter",
  skiprows=2, header=0, index_col=0)
```

2 Density Functions

This section contains code for different density functions of the various models. It is taken from `estimate_parameters.py`. First we import modules.

```
import numpy as np
import scipy.special as sp
import scipy.optimize as opt
```

Some constants

```
pi    = np.pi
e     = np.e
exp   = np.exp
floor = np.floor
log   = np.log
sqrt  = np.sqrt
```

Special functions

```
B     =  sp.beta
erf   =  sp.erf
G     =  sp.gamma
I     =  sp.betainc
Phi   =  sp.ndtr
Phinv =  sp.ndtri
psi   =  sp.digamma
Q     =  sp.gammaincc
root = opt.root_scalar
def psi1(x):
  return  sp.polygamma(1,x)
def Phi_prime(x):
  return exp(-x**2 / 2) / sqrt(2 * pi)
def zeta(x):
  return 1 + sp.zetac(x)
def zeta(x):
  return 1 +  sp.zetac(x)
```

We use the the approximation for $\log\Gamma(z)$ from Chen (2016) when z is large to avoid any numerical problems.[1] The approximation is

$$\log\Gamma(x+1) \approx \log\sqrt{2\pi x} + x(\log x - 1) + \left(x^2 + \frac{53}{210}\right)\log\left(1 + \frac{1}{12x^3 + \frac{24}{7}x - \frac{1}{2}}\right)$$

When $x \geq 10$, this approximation is accurate to one part in a billion. It is actually probably better to use Scipy's built-in gamma function, which is based on a series approximation, but I didn't realize that when I originally wrote this code. Such is life.

```
Gamma_frac1 = 53/210
Gamma_frac2 = 24/7
```

[1] Chen, Chao-Ping. 2016. "A More Accurate Approximation for the Gamma Function." *Journal of Number Theory* 164: 417–428.

```
def log_G(z):
  return log(G(z))
def log_G_approx(z):
  x = z - 1
  term1 = 0.5 * log(2 * pi * x)
  term2 = x * (log(x) - 1)
  term3 = (x * x + Gamma_frac1)
  term4 = log(1 + 1 / (12 * (x * x * x) + Gamma_frac2 * (x) - 0.5))
  return term1 + term2 + term3 * term4
```

Empty dictionary where we will store densities later.

```
density  = {}
```

GB2: The density is

$$y = \frac{\alpha\,(x-c)^{\alpha p-1}}{\beta^{\alpha p} B(p,q)\left(1+\left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{p+q}}$$

where B is the beta function. We rewrite the numerator and denominator in terms of a logarithm in case any factors are particularly small or particularly large. The beta function can be difficult for a computer to handle numerically depending on how it is implemented. We use the previous approximation for $\log\Gamma(z)$ to calculate $\log B(p,q)$ (specifically $\log\Gamma(p)$, $\log\Gamma(q)$, and $\log\Gamma(p+q)$) without any risk of numerical complications. Code:

```
def density_GB2(x, params):
  a, b, p, q, c = params
```

We manually check if p, q, and $p+q$ are each less than 10. If yes, we can use the built-in gamma function, and otherwise, we approxmate.

```
  if p < 10:
    log_G_p = log_G(p)
  else:
    log_G_p = log_G_approx(p)
  if q < 10:
    log_G_q = log_G(q)
  else:
    log_G_q = log_G_approx(q)
  if p + q < 10:
    log_G_pq = log_G(p + q)
  else:
    log_G_pq = log_G_approx(p + q)
```

And now calculate the density.

```
  if x <= c:
    return 0
  else:
    num   = log(a) + (a*p-1) * log(x-c)           # numerator
    denom = (     (a*p) * log(b) + log_G_p
                + log_G_q - log_G_pq
                + (p+q) * log(1+((x-c)/b)**a)    )    # denominator
    return exp(num - denom)
```

Dagum: The density is

$$y = \frac{\alpha p (x-c)^{\alpha p - 1}}{\beta^{\alpha p} \left(1 + \left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{p+1}}$$

As with GB2, we calculate the numerator an denominator in logs to avoid any potential numerical issues.

```
def density_Dagum(x, params):
  a, b, p, c = params
  if x <= c:
    return 0
  else:
    num = log(a) + log(p) + (a*p-1) * log(x-c)
    denom = (a*p) * log(b) + (p+1) * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

Burr: Density is

$$y = \frac{\alpha q (x-c)^{\alpha - 1}}{\beta^{\alpha} \left(1 + \left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{q+1}}$$

Code:

```
def density_Burr(x, params):
  a, b, q, c = params
  if x <= c:
    return 0
  else:
    num = log(a) + log(q) + (a-1) * log(x-c)
    denom = a * log(b) + (1+q) * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

Fisk: The density is

$$y = \frac{\alpha (x-c)^{\alpha - 1}}{\beta^{\alpha} \left(1 + \left(\frac{x-c}{\beta}\right)^{\alpha}\right)^{2}}$$

Code:

```
def density_Fisk(x, params):
  a, b, c = params
  if x <= c:
    return 0
  else:
    num   = log(a) + (a-1) * log(x-c)
    denom = a * log(b) + 2 * log(1 + ((x-c)/b)**a)
    return exp(num - denom)
```

InvG: The density is

$$y = \frac{\beta^\alpha}{\Gamma(\alpha)} \frac{e^{-\frac{\beta}{x-c}}}{(x-c)^{1+\alpha}}$$

Code:

```
def density_InvG(x, params):
  a, b, c = params
```

Similar to GB2, we manually check if we need to approximate $\log \Gamma(\alpha)$.

```
  if a < 10:
    log_G_a = log_G(a)
  else:
    log_G_a = log_G_approx(a)
  if x <= c:
    return 0
  else:
    num = a * log(b) - b / (x-c)
    denom = log_G_a + (1+a) * log(x-c)
    return exp(num - denom)
```

Davis: The probability density is

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)\zeta(\alpha)} \left[\frac{1}{\left(e^{\frac{\beta}{x-c}} - 1\right)(x-c)^{1+\alpha}} \right]$$

Code for density. We use $c+1000$ instead of c because for x close to c, the code was producing an overflow error.

```
def density_Davis(x, params):
  a, b, c = params
  if x <= c + 1000:
    return 0
  else:
    num = a * log(b)
    denom = log(G(a)) + log(zeta(a)) + \
            log(exp(exp(log(b) - log(x - c))) - 1) + \
            (1 + a) * log(x - c)
    return exp(num - denom)
```

CSS InvG: The density is

$$y = \frac{(\psi_0 + \psi_1 t + \psi_2 \alpha)^\alpha}{\Gamma(\alpha)} \frac{e^{-\frac{\psi_0 + \psi_1 t + \psi_2 \alpha}{x - \phi(\psi_0 + \psi_1 t + \psi_2 \alpha)}}}{(x - \phi(\psi_0 + \psi_1 t + \psi_2 \alpha))^{1+\alpha}}$$

```
def density_CSS_InvG(x, t, phi, psi, a):
  psi0, psi1, psi2 = psi
  beta = (psi0 + psi1 * t + psi2 * a) / phi
```

```
  c = psi0 + psi1 * t + psi2 * a
  if x <= c:
    return 0
  else:
    return density_InvG(x, [a, beta, c])
```

CSS_InvG_Prop: The density is

$$y = \frac{\alpha^\alpha(\psi_0 + \psi_1 t)^\alpha}{\phi^\alpha \Gamma(\alpha)} \frac{e^{-\frac{1}{\phi}\frac{\alpha(\psi_0+\psi_1 t)}{x-\alpha(\psi_0+\psi_1 t)}}}{(x-\alpha(\psi_0+\psi_1 t))^{1+\alpha}}$$

```
def density_CSS_InvG_prop(x, t, phi, psi, a):
  psi0, psi1 = psi
  beta = a * (psi0 + psi1 * t) / phi
  c = a * (psi0 + psi1 * t)
  if x <= c:
    return 0
  else:
    return density_InvG(x, [a, beta, c])
```

LogN_P_cut: Density:

$$y = \begin{cases} \frac{1}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} & \text{if } x < k \\ \frac{\alpha x_m^\alpha}{(x-c)^{1+\alpha}} & \text{if } x \geq k \end{cases}$$

Code:

```
def density_LogN_P_cut(x, params):
  mu, sigma_sq, k, x_m, a, c = params
  if k < c:
    print("k was less than c; setting k=c")
    k = c
  if x <= c:
    return 0
  elif x > c and x < k:
    frac = 1/((x-c) * sqrt(2 * pi * sigma_sq))
    exponent = -(log(x-c) - mu)**2 / (2 * sigma_sq)
    return frac * exp(exponent)
  elif x >= k:
    if x <= x_m + c:
      return 0
    else:
      return exp(log(a) + a * log(x_m) - (1+a) * log(x-c))
  else:
    raise RuntimeError("Something weird happened")
```

LogN_P_mix: Density is

$$y = \frac{\gamma}{(x-c)\sigma\sqrt{2\pi}} e^{-(\log(x-c)-\mu)^2/2\sigma^2} + (1-\gamma)\,\chi_{x \geq x_m + c} \frac{\alpha x_m^\alpha}{(x-c)^{1+\alpha}},$$

where $\chi_{x \geq x_m + c}$ is an indicator function. Code:

```
def density_LogN_P_mix(x, params):
  mu, sigma_sq, gamma, x_m, alpha, c = params
  if gamma < 0 or gamma > 1:
    raise ValueError("gamma is outside unit interval")
  if x <= c:
    return 0
  else:
    log_n_term = exp(log(gamma) - log(x - c) -
                     0.5 * log(2 * pi * sigma_sq) -
                     (log(x - c) - mu) ** 2 / (2 * sigma_sq))
    if x <= x_m + c:
      pareto_term = 0
    else:
      pareto_term = exp(log(1 - gamma) + log(alpha) + alpha * log(x_m) -
                        (1 + alpha) * log(x - c))
    return log_n_term + pareto_term
```

Then we add the density functions to `density`.

```
density["GB2"] = density_GB2
density["Dagum"] = density_Dagum
density["Burr"] = density_Burr
density["Fisk"] = density_Fisk
density["InvG"] = density_InvG
density["Davis"] = density_Davis
density["CSS_InvG"] = density_CSS_InvG
density["CSS_InvG_prop"] = density_CSS_InvG_prop
density["LogN_P_cut"] = density_LogN_P_cut
density["LogN_P_mix"] = density_LogN_P_mix
```

3 Code from make_figures.py

The following code is taken from make_figures.py. We change it slightly. For example, the microdata data is already binned, so we do not need to load bin. We define several graphics-producing functions:

- single_graph—makes a single graph
- lin_loglog_graphs—makes three rows of two graphs each, where the first column is linear and the second column is loglog scaling
- lin_graphs—makes six linearly scaled graphs
- loglog_graphs—makes six loglog scaled graphs
- four_graphs—makes four graphs in 2x2 layout; hook for extra code
- single_graph_ext—makes a single graph but with more control over the contents; similar to four_graphs in its implementation/interface

Pyplot commands happen inside these functions. These functions use the plt.savefig() function to save the figure directly as a pdf at the correct size. The first thing to do is import modules.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import matplotlib as mpl
```

Set some rcParams to make the figures look nice. You will need a working TEX distribution on your machine to produce figures, or you may have to rewrite some figure titles and labels.

```
plt.rcParams["legend.fontsize"] = "small"
plt.rcParams["font.family"] = "serif"
plt.rcParams["text.usetex"] = True      # tells Pyplot to use TeX
plt.rcParams["figure.constrained_layout.use"] = True
plt.rcParams["savefig.dpi"] = 300
plt.rcParams["savefig.format"] = "pdf"
```

Bookkeeping functions.

```
def check_var(data, var):
  if var not in data:
    raise KeyError("{0} is not a column in the data".format(var))
def check_list(x):
  is_list_like = hasattr(x, "__len__") and \
                 hasattr(x, "__getitem__") and \
                 hasattr(x, "__iter__")
  if not is_list_like or isinstance(x, str):
    raise TypeError("Please use list instead of {0} for {1}".format(type(x),x))
```

We begin by creating a function to show a single line graph. We will use this to create the figure of constant-shift-scale inverse-gamma parameters.

```
def single_graph(data, parameter, *, filename, title=None):
  plt.close(plt.gcf())
  if isinstance(title, type(None)):
    called_with_title = False
```

```
  else:
    called_with_title = True
  check_var(data, parameter)
  plt.plot(data.index, data[parameter], c="black", lw=0.5)
  if called_with_title:
    plt.title(title)
  #plt.show()
  plt.gcf().set_size_inches(3.25, 2.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The `lin_loglog_graphs` function is more complicated. It makes a 3x2 plot of three rows of sample and model densities in linear and log-log scales. Arguments of the function are

- Three DataFrames (containing sample density data)
- Several lists of things, where we plot everything from each list on one row of the subplot array
  - `list_F1`, `list_F2`, and `list_F3`—density functions, which take a single argument and return a (nonnegative) real number
  - `list_c1`, `list_c2`, and `list_c3`—minimum values of the support
  - `list_opt1`, `list_opt2`, and `list_opt3`—list of dictionaries containing keyword-options for plotting each density function
- Bounds on the graphs
- A filename for saving the figure
- An optional list of titles and boolean to determine if using a legend

For the lists of functions, constants, and options, we will plot each list on one row of the subplot array. The left column is a linear scaling, and the right column is a loglog scaling. We need to use lists of functions and constants so that we can make a figure where each row plots multiple functions. We begin with error checking and setting the `called_with_titles` boolean.

```
def lin_loglog_graphs(data1, data2, data3,
    list_F1, list_c1, list_opt1,
    list_F2, list_c2, list_opt2,
    list_F3, list_c3, list_opt3,
    x_bounds_lin, y_bounds_lin, x_bounds_log, y_bounds_log, *,
    filename, wgt=None, titles=[], with_legend=False):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

We loop through the rows of the subplot. For each row, we store the lists of density functions, constants, and option dictionaries in new variables. Then we check that all of them are in fact a list.

```
for i in range(1, 4):
  list_F = eval("list_F" + str(i))
  list_c = eval("list_c" + str(i))
  list_opt = eval("list_opt" + str(i))
  check_list(list_F)
  check_list(list_c)
  check_list(list_opt)
  binned_data = eval("data" + str(i))
```

Start with the linear plot. We add a title, a curve for the density, and sample density points. The `x_vals` list starts with `c_i` so that it is clear on the figure where the support of the model ends. We loop through the lists of functions, constants, and options concurrently and plot each one separately on the current plot.

```
  plt.subplot(3, 2, 2 * i - 1)
  if called_with_titles:
    plt.title(titles[2 * i - 2])
  for F, c, opt in zip(list_F, list_c, list_opt):
    x_vals = np.linspace(c, x_bounds_lin[1], 200)
    y_vals = list(map(F, x_vals))
    plt.plot(x_vals, y_vals, **opt)
  plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
  plt.xlim(x_bounds_lin)
  plt.ylim(y_bounds_lin)
  if with_legend:
    plt.legend(loc="upper right")
```

Now do the loglog plot. The code is similar except that this time, the `x_vals` list runs for the entire length of the horizontal axis.

```
  plt.subplot(3, 2, 2 * i)
  if called_with_titles:
    plt.title(titles[2 * i - 1])
  x_vals = np.geomspace(*x_bounds_log, 200)
  for F, c, opt in zip(list_F, list_c, list_opt):
    y_vals = list(map(F, x_vals))
    plt.plot(x_vals, y_vals, **opt)
  plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
  plt.xlim(x_bounds_log)
  plt.ylim(y_bounds_log)
  plt.loglog()
  if with_legend:
    plt.legend(loc="lower left")
```

After the for-loop, plot the figure.

```
#plt.show()
plt.gcf().set_size_inches(6.5, 7.5)
plt.savefig(filename)
plt.close(plt.gcf())
```

The lin_graphs and loglog_graphs functions will be the same as lin_loglog_graphs except that they take more data and scale their plots all the same way.

```
def lin_graphs(data1, data2, data3,
    data4, data5, data6,
    list_F1, list_c1, list_opt1,
    list_F2, list_c2, list_opt2,
    list_F3, list_c3, list_opt3,
    list_F4, list_c4, list_opt4,
    list_F5, list_c5, list_opt5,
    list_F6, list_c6, list_opt6,
    x_bounds, y_bounds, *,
    filename, wgt=None, titles=[], with_legend=True):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

Now loop through the panels, create pointers, and check list properties.

```
  for i in range(1, 7):
    list_F = eval("list_F" + str(i))
    list_c = eval("list_c" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_F)
    check_list(list_c)
    check_list(list_opt)
    binned_data = eval("data" + str(i))
```

Plot the figures.

```
    plt.subplot(3, 2, i)
    if called_with_titles:
      plt.title(titles[i - 1])
    for F, c, opt in zip(list_F, list_c, list_opt):
      x_vals = np.linspace(c, x_bounds[1], 200)
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds)
    plt.ylim(y_bounds)
    if with_legend:
      plt.legend(loc="upper right")
```

After the for-loop, plot the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 7.5)
  plt.savefig(filename)
```

```
  plt.close(plt.gcf())
```

Same but with loglog plots.

```
def loglog_graphs(data1, data2, data3,
    data4, data5, data6,
    list_F1, list_opt1,
    list_F2, list_opt2,
    list_F3, list_opt3,
    list_F4, list_opt4,
    list_F5, list_opt5,
    list_F6, list_opt6,
    x_bounds, y_bounds, *,
    filename, wgt=None, titles=[], with_legend=True):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 6:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

Now loop through the panels, create pointers, and check list properties.

```
  for i in range(1, 7):
    list_F = eval("list_F" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_F)
    check_list(list_opt)
    binned_data = eval("data" + str(i))
```

Create the figures.

```
    plt.subplot(3, 2, i)
    if called_with_titles:
      plt.title(titles[i - 1])
    for F, opt in zip(list_F, list_opt):
      x_vals = np.geomspace(*x_bounds, 200)
      y_vals = list(map(F, x_vals))
      plt.plot(x_vals, y_vals, **opt)
    plt.scatter(binned_data["mid"], binned_data["dens"], s=2, c="blue")
    plt.xlim(x_bounds)
    plt.ylim(y_bounds)
    plt.loglog()
    if with_legend:
      plt.legend(loc="lower left")
```

After the for-loop, plot the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 7.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

The `four_graphs` is similar to the `lin_loglog_graphs`, etc. in that it accepts lists of information for creating multiple figures on the subplot. It is slightly lower-level than previous plotting functions in that we feed it actual data rather than a DataFrame and column. The arguments are

- `x_data` for the horizontal axes
- Several lists of things, one for each panel of the figure
    1. `list_y1`, `list_y2`, `list_y3`, `list_y4`—lists of data for the vertiacal axis
    2. `list_plot1`, `list_plot2`, `list_plot3`, `list_plot4`—lists of plotting functions. Most of these will be `plt.plot()`
    3. `list_opt1`, `list_opt2`, `list_opt3`, `list_opt4`—lists of options for plotting functions
- A filename for the figure
- An optional list of titles for the subgraphs

We begin with error checking and setting the `called_with_titles` boolean.

```
def four_graphs(
    list_x1, list_y1, list_plot1, list_opt1,
    list_x2, list_y2, list_plot2, list_opt2,
    list_x3, list_y3, list_plot3, list_opt3,
    list_x4, list_y4, list_plot4, list_opt4, *,
    filename, titles=[], extra_code=""):
  plt.close(plt.gcf())
  check_list(titles)
  if len(titles) > 0 and len(titles) < 4:
    raise ValueError("Please specify zero or all titles")
  elif len(titles) == 0:
    called_with_titles = False
  else:
    called_with_titles = True
```

We loop through the four panels. On each iteration, we first create new pointers to the lists for that iteration and check that they are actually lists. Then we loop through the three lists of y-axis data, plotting functions, and options. We call the plotting function with the x-axis data, corresponding y-axis data, and corresponding plotting options.

```
  for i in range(1,5):
    list_x = eval("list_x" + str(i))
    list_y = eval("list_y" + str(i))
    list_plot = eval("list_plot" + str(i))
    list_opt = eval("list_opt" + str(i))
    check_list(list_x)
    check_list(list_y)
    check_list(list_plot)
    check_list(list_opt)
```

Now plot the data for this subplot.

```
    plt.subplot(2, 2, i)
    for x, y, plot, opt in zip(list_x, list_y, list_plot, list_opt):
```

```
      plot(x, y, **opt)
    if called_with_titles:
      plt.title(titles[i-1])
```

Execute any extra code. Used to provide a hook into the function.

```
  exec(extra_code)
```

Then save the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(6.5, 5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

A single-panel with more control. This function uses the same intervace as the `four_graphs` function.

```
def single_graph_ext(
    list_x, list_y, list_plot, list_opt, *,
    filename, title=None, extra_code=""):
  plt.close(plt.gcf())
  if isinstance(title, type(None)):
    called_with_title = False
  else:
    called_with_title = True
```

Now actually make the graph and execute the extra code.

```
  for x, y, plot, opt in zip(list_x, list_y, list_plot, list_opt):
    plot(x, y, **opt)
  if called_with_title:
    plt.title(title)
  exec(extra_code)
```

And save the figure.

```
  #plt.show()
  plt.gcf().set_size_inches(3.25, 2.5)
  plt.savefig(filename)
  plt.close(plt.gcf())
```

4 Making the Figures

The code in this section is adapted from `main.py`.

Figure 1: `CSS_lin_densities.pdf`. We begin with the plots of empirical and model density for past years of data. To keep things simple, we use data from 1967, 1995, and 2023. We define density functions for each year. To make it simple to change years later, we save the years in variables `year1`, `year2`, and `year3`.

```
print("Making CSS_lin_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return density["CSS_InvG"](x, {0}, phi_lin,
      [psi0_lin, psi1_lin, psi2_lin],
      CSS_InvG_lin_parameters.loc[{0}, "alpha"])""".format(i))
  exec("""c_{0} = psi0_lin + psi1_lin * {0} + \
    psi2_lin * CSS_InvG_lin_parameters.loc[{0}, "alpha"]""".format(i))
lin_loglog_graphs(
  sample_dens[year1], sample_dens[year2], sample_dens[year3],
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="CSS_lin_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 2: `CSS_prop_densities.pdf`. Same thing except with proportional relationship imposed on parameters.

```
print("Making CSS_prop_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return density["CSS_InvG_prop"](x, {0}, phi_prop,
      [psi0_prop, psi1_prop],
      CSS_InvG_prop_parameters.loc[{0}, "alpha"])""".format(i))
  exec("""c_{0} = CSS_InvG_prop_parameters.loc[{0}, "alpha"] * \
    (psi0_prop + psi1_prop * {0})""".format(i))
lin_loglog_graphs(
  sample_dens[year1], sample_dens[year2], sample_dens[year3],
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
```

```
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="CSS_prop_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 3: `InvG_densities.pdf`. Same thing except with inverse-gamma.

```
print("Making InvG_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return density["InvG"](x, InvG_parameters.loc[{0}])""".format(i))
  exec("""c_{0} = InvG_parameters.loc[{0}, "c"]""".format(i))
lin_loglog_graphs(
  sample_dens[year1], sample_dens[year2], sample_dens[year3],
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                 [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="InvG_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 4: `Fisk_densities.pdf`. Same thing except with Fisk densities.

```
print("Making Fisk_densities.pdf")
year1, year2, year3 = [1967, 1995, 2023]
for i in [year1, year2, year3]:
  exec("""def F_{0}(x):
    return density["Fisk"](x, Fisk_parameters.loc[{0}])""".format(i))
  exec("""c_{0} = Fisk_parameters.loc[{0}, "c"]""".format(i))
lin_loglog_graphs(
  sample_dens[year1], sample_dens[year2], sample_dens[year3],
  [eval("F_{0}".format(year1))], [eval("c_{0}".format(year1))],
                                 [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year2))], [eval("c_{0}".format(year2))],
```

```
                                  [{"c": "black", "lw": 0.7}],
  [eval("F_{0}".format(year3))], [eval("c_{0}".format(year3))],
                                  [{"c": "black", "lw": 0.7}],
  [-20000,100000], [-0.05e-4,2.05e-4], [1000,1100000], [1e-10,2e-4],
  filename="Fisk_densities", wgt="weight",
  titles=["{0} Data (Linear Scale)".format(year1),
          "{0} Data (Log Scale)".format(year1),
          "{0} Data (Linear Scale)".format(year2),
          "{0} Data (Log Scale)".format(year2),
          "{0} Data (Linear Scale)".format(year3),
          "{0} Data (Log Scale)".format(year3)])
```

Figure 5: `InvG_parameter_graphs.pdf`—The next set of graphs shows the graphs of inverse-gamma parameters. We have four panels: one for each parameter and one with the normalized parameters. Before we create the figure, we need a few more Series. The `norm` Series are inverse-gamma parameters normalized by the sum of that parameter across all years.

```
print("Making InvG_parameter_graphs.pdf")
years = InvG_parameters.index
norm_alpha = InvG_parameters["alpha"] / InvG_parameters["alpha"].sum()
norm_beta = InvG_parameters["beta"] / InvG_parameters["beta"].sum()
norm_c = InvG_parameters["c"] / InvG_parameters["c"].sum()
four_graphs(
  [years], [InvG_parameters["alpha"]], [plt.plot],
    [{"c": "black", "lw": 0.5}],
  [years], [InvG_parameters["beta"]], [plt.plot],
    [{"c": "black", "lw": 0.5}],
  [years], [InvG_parameters["c"]], [plt.plot],
    [{"c": "black", "lw": 0.5}],
  [*[years]*3],
    [norm_alpha, norm_beta, norm_c], [plt.plot, plt.plot, plt.plot],
    [{"c": "blue", "lw": 0.7, "ls": "--", "label": "Shape $\\alpha$"},
     {"c": "black", "lw": 0.5, "label": "Scale $\\beta$"},
     {"c": "red", "lw": 1.0, "ls": ":", "label": "Shift $c$"}],
  filename="InvG_parameter_graphs", titles=["Shape Parameter $\\alpha$",
    "Scale Parameter $\\beta$", "Shift Parameter $c$",
    "Normalized Parameters"],
    extra_code = """plt.subplot(2,2,4)
plt.legend()""")
```

Figure 6: `InvG_parameter_regression.pdf`—The next figure is a plot illustrating the relationship between the parameters. The upper row will show the difference and quotient between normalized shape and the other two parameters. The lower row will comparing actual and predicted values for β and c. The `beta_hat` and `c_hat` Series are the predicted values from the linear regression of β and c on α and the year.

```
print("Making InvG_parameter_regression.pdf")
beta_hat = (psi0_lin + psi1_lin * InvG_parameters.index + \
  psi2_lin * InvG_parameters["alpha"]) / phi_lin
```

```
c_hat = psi0_lin + psi1_lin * InvG_parameters.index + \
  psi2_lin * InvG_parameters["alpha"]
norm_beta_hat = beta_hat / beta_hat.sum()
norm_c_hat = c_hat / c_hat.sum()
```

We make numpy arrays out of the differences and quotients of the normalized parameters.

```
diff_beta_alpha = (norm_beta - norm_alpha).to_numpy()
diff_c_alpha = (norm_c - norm_alpha).to_numpy()
quot_beta_alpha = (norm_beta / norm_alpha).to_numpy()
quot_c_alpha = (norm_c / norm_alpha).to_numpy()
```

Now two linear regressions for the combinations of normalized parameters. We are approximating the quotient and differences as linear functions of time. First the differences.

```
year_cons = pd.DataFrame({"cons": 1, "year": years}).to_numpy()
reg_diff = np.linalg.inv(np.transpose(year_cons) @ year_cons) @ \
  np.transpose(year_cons) @ (0.5 * (diff_beta_alpha + diff_c_alpha))
reg_diff_vals = reg_diff[0] + reg_diff[1] * InvG_parameters.index
```

And quotient.

```
reg_quot = np.linalg.inv(np.transpose(year_cons) @ year_cons) @ \
  np.transpose(year_cons) @ (0.5 * (quot_beta_alpha + quot_c_alpha))
reg_quot_vals = reg_quot[0] + reg_quot[1] * InvG_parameters.index
```

Now make the figure. Every list of x-values to use will be the same, so we put copies of `years` in a list for each of those function arguments.

```
from matplotlib.markers import MarkerStyle
four_graphs(
  [*[years]*3],
  [diff_beta_alpha, diff_c_alpha, reg_diff_vals],
    [plt.scatter, plt.scatter, plt.plot],
    [{"c": "blue", "s": 2.0, "label": "$\\bar\\beta_t-\\bar\\alpha_t$"},
     {"c": "red", "s": 10.0, "label": "$\\bar c_t-\\bar\\alpha_t$",
       "marker": "2"},
     {"c": "black", "lw": 0.7, "ls": "--", "label": "Trendline"}],
  [*[years]*3],
  [quot_beta_alpha, quot_c_alpha, reg_quot_vals],
    [plt.scatter, plt.scatter, plt.plot],
    [{"c": "blue", "s": 2.0, "label": "$\\bar\\beta_t/\\bar\\alpha_t$"},
     {"c": "red", "s": 10.0, "label": "$\\bar c_t/\\bar\\alpha_t$",
       "marker": "2"},
     {"c": "black", "lw": 0.7, "ls": "--", "label": "Trendline"}],
  [*[years]*2],
  [InvG_parameters["beta"], beta_hat], [plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5, "label": "Observed"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Predicted"}],
  [*[years]*2],
  [InvG_parameters["c"], c_hat], [plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5, "label": "Observed"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Predicted"}],
  filename="InvG_parameter_regression",
```

```
  titles=["(Normalized) Differences",
    "(Normalized) Quotients", "Predicted Scale",
    "Predicted Shift"],
```

The extra code for this function call will be a loop through subplots, and on each iteration, we specify adding a legend for that subplot.

```
  extra_code= \
"""for i in range(1, 5):
  plt.subplot(2, 2, i)
  plt.legend()""")
```

Figure 7: `Fisk_parameters_normalized.pdf`—a graph of normalized Fisk parameters. We start by creating series of normalized parameters, and then we put them in a figure.

```
print("Making Fisk_parameters_normalized.pdf")
Fisk_alpha_norm = Fisk_parameters["alpha"] / Fisk_parameters["alpha"].sum()
Fisk_beta_norm = Fisk_parameters["beta"] / Fisk_parameters["beta"].sum()
Fisk_c_norm = Fisk_parameters["c"] / Fisk_parameters["c"].sum()
single_graph_ext(
  [*[years]*3],
    [Fisk_alpha_norm, Fisk_beta_norm, Fisk_c_norm],
    [plt.plot, plt.plot, plt.plot],
    [{"c": "blue", "lw": 0.7, "ls": "--", "label": "Shape $\\alpha$"},
     {"c": "black", "lw": 0.5, "label": "Scale $\\beta$"},
     {"c": "red", "lw": 1.0, "ls": ":", "label": "Shift $c$"}],
  filename="Fisk_parameters_normalized", title="Normalized Fisk Parameters",
    extra_code = "plt.legend()")
```

Figure 8: `CSS_InvG_parameters_graph.pdf`—A graph of the constant-shift-scale inverse-gamma parameter estimates.

```
print("Making CSS_InvG_parameters_graph.pdf")
single_graph(CSS_InvG_lin_parameters, "alpha",
  filename="CSS_InvG_parameters_graph",
  title="Shape Parameter $\\alpha$ Estimates")
```

Figure 9: `comparison_linear_graphs.pdf`—Now we make graphs for the different densities in 2023. First we load the files with parameter estimates saved as dictionaries. The dictionary `parameters` will save the parameters for each distribution.

```
print("Making comparison_linear_graphs.pdf")
```

Now we create the density functions and constants.

```
for i in ["GB2", "Dagum", "Burr", "Davis", "LogN_P_cut",
          "LogN_P_mix"]:
  exec("""def F_{0}(x):
    return density['{0}'](x, parameters['{0}'])""".format(i))
  exec("c_{0} = parameters['{0}'][-1]".format(i))
```

Constant-shift-scale inverse-gamma density.

```
def F_CSS_InvG(x):
  return density["CSS_InvG"](x, 2023, phi_lin,
    [psi0_lin, psi1_lin, psi2_lin],
```

```
    CSS_InvG_lin_parameters.loc[2023, "alpha"])
c_CSS_InvG = psi0_lin + psi1_lin * 2023 + \
  psi2_lin * CSS_InvG_lin_parameters.loc[2023, "alpha"]
```

Make the graphs. We create two sets of comparison graphs where we put the inverse-gamma density on each plot and one alternative model on each plot. The first set of graphs will be linear scale, and the second one will be loglog scale.

```
lin_graphs(*[sample_dens[2023]]*6,
  [F_CSS_InvG, F_GB2], [c_CSS_InvG, c_GB2],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Gen Beta II"}],
  [F_CSS_InvG, F_Dagum], [c_CSS_InvG, c_Dagum],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Dagum"}],
  [F_CSS_InvG, F_Burr], [c_CSS_InvG, c_Burr],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Burr"}],
  [F_CSS_InvG, F_Davis], [c_CSS_InvG, c_Davis],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Davis"}],
  [F_CSS_InvG, F_LogN_P_cut], [c_CSS_InvG, c_LogN_P_cut],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Cutoff"}],
  [F_CSS_InvG, F_LogN_P_mix], [c_CSS_InvG, c_LogN_P_mix],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Mixture"}],
  [-20000,100000], [-0.05e-5,2.05e-5],
  filename="comparison_linear_graphs", wgt="weight",
  titles=["Gen Beta II", "Dagum", "Burr", "Davis",
    "Log-Normal/Pareto Cutoff", "Log-Normal/Pareto Mix"])
```

Figure 10: `comparison_loglog_graphs.pdf`—Same thing with loglog scaling.

```
print("Making comparison_loglog_graphs.pdf")
loglog_graphs(*[sample_dens[2023]]*6,
  [F_CSS_InvG, F_GB2],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Gen Beta II"}],
  [F_CSS_InvG, F_Dagum],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Dagum"}],
  [F_CSS_InvG, F_Burr],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Burr"}],
  [F_CSS_InvG, F_Davis],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Davis"}],
  [F_CSS_InvG, F_LogN_P_cut],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Cutoff"}],
```

```
  [F_CSS_InvG, F_LogN_P_mix],
    [{"c": "black", "lw": 0.5, "label": "Inverse-Gamma"},
     {"c": "blue", "lw": 0.7, "ls": "--", "label": "Mixture"}],
  [1000,1100000], [1e-10,2e-4],
  filename="comparison_loglog_graphs", wgt="weight",
  titles=["Gen Beta II", "Dagum", "Burr", "Davis",
    "Log-Normal/Pareto Cutoff", "Log-Normal/Pareto Mix"])
```

Figure 11: `gini_compare_graphs.pdf`—For fun we make a final graphic about the Gini coefficient, which is

$$\frac{\beta}{\beta + (\alpha - 1)c} \frac{\Gamma(\alpha - \frac{1}{2})}{\sqrt{\pi}\Gamma(\alpha)} = \frac{1}{1 + (\alpha - 1)\phi} \frac{\Gamma(\alpha - \frac{1}{2})}{\sqrt{\pi}\Gamma(\alpha)}$$

We will use `four_graphs()` again. The `gini` Series contains values of the Gini coefficient in each year. The `gini_unshift` is the value of the shape parameter in the unshifted Gini coefficient formula. The third and fourth graphs show the Gini coefficient for different values of α.

```
print("Making gini_compare_graphs.pdf")
gini = 1 / (1 + (CSS_InvG_lin_parameters["alpha"] - 1) * phi_lin) * \
  G(CSS_InvG_lin_parameters["alpha"] - 0.5) / (np.sqrt(np.pi) *
  G(CSS_InvG_lin_parameters["alpha"]))
gini_unshift = \
  G(CSS_InvG_lin_parameters["alpha"] - 0.5) / (np.sqrt(np.pi) *
  G(CSS_InvG_lin_parameters["alpha"]))
```

Now make the Gini coefficient for different values of α. The graph with Gini coefficient of unshifted inverse-gamma distribution is straightforward. For shifted inverse-gamma distribution, we have a minimum value around 3. Setting the derivative equal to 0 gives us

$$\begin{aligned}
y &= \frac{1}{1 + (x - 1)\phi} \frac{\Gamma\left(x - \frac{1}{2}\right)}{\sqrt{\pi}\Gamma(x)} \\
\log y &= -\log(1 + (x - 1)\phi) + \log\Gamma\left(x - \frac{1}{2}\right) - \log\Gamma(x) - \log\sqrt{\pi} \\
\frac{y'}{y} &= -\frac{\phi}{1 + (x - 1)\phi} + \psi\left(x - \frac{1}{2}\right) - \psi(x) = 0
\end{aligned}$$

We can solve this equation numerically for α. The `right_singularity` variable is the x-value where the Gini coefficient has a singularity because the denominator blows up, in other words $1 + (\alpha - 1)\phi \to 0$. The `alpha_vals`, `gini_vals_shift`, and `gini_vals_unshift` lists are lists of x and y-values for plotting the Gini coefficient.

```
right_singularity = 1 - 1 / phi_lin
alpha_vals = np.linspace(0.6, right_singularity - 0.2, 200)
gini_vals_shift = 1 / (1 + (alpha_vals - 1) * phi_lin) * \
  G(alpha_vals - 0.5) / (np.sqrt(np.pi) *
  G(alpha_vals))
gini_vals_unshift = \
  G(alpha_vals - 0.5) / (np.sqrt(np.pi) *
  G(alpha_vals))
```

The `D_gini` function is the log-derivative y'/y of the Gini coefficient for shifted inverse-gamma distribution. As noted above, we can find the alpha corresponding to minimum Gini coefficient when $\texttt{D\_gini} = 0$. We set `min_alpha` and `max_alpha` to be the minimum and maximum values of α in the data.

```
def D_gini(x):
  return psi(x - 0.5) - psi(x) - phi_lin / (1 + (x - 1) * phi_lin)
min_gini = root(D_gini, bracket=[1, right_singularity - 0.2]).root
min_alpha = CSS_InvG_lin_parameters["alpha"].min()
max_alpha = CSS_InvG_lin_parameters["alpha"].max()
```

Now make the figure. We have several graphs on each subplot:

1. Shape parameter and horizontal lines for minimum and maximum values
2. Gini coefficient calculated from parameter estimates
3. Gini coefficient as a function of α under a shifted inverse-gamma distribution as well as vertical lines at the minimum and maximum values of α observed in the data
4. Gini coefficient as a function of α under an unshifted inverse-gamma distribution as well as vertical lines at the minimum and maximum values of α observed in the data

The file will be `gini_compare_graphs.pdf`.

```
four_graphs(
  [years, (years[0], years[-1]), (years[0], years[-1])],
    [CSS_InvG_lin_parameters["alpha"],
      (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  [years], [gini], [plt.plot],
    [{"c": "black", "lw": 0.5}],
  [alpha_vals, (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [gini_vals_shift, (0,2), (0,2)], [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  [alpha_vals, (min_alpha, min_alpha), (max_alpha, max_alpha)],
    [gini_vals_unshift, (0,2), (0,2)], [plt.plot, plt.plot, plt.plot],
    [{"c": "black", "lw": 0.5},
     {"c": "red", "lw": 0.7, "ls": "--"},
     {"c": "blue", "lw": 0.7, "ls": "--"}],
  filename="gini_compare_graphs",
  titles=["Shape Parameter $\\alpha$",
    "Gini Coefficient from Parameters",
    "Gini Coefficient Function (With Shift)",
    "Gini Coefficient Function (No Shift)"],
```

The `extra_code` for this function sets the vertical limits for the third and fourth subplots and adds a light blue rectangular patch to the third graph to denote the portion of the domain where the Gini coefficient is increasing.

```
    extra_code= \
"""plt.subplot(2, 2, 3)
plt.ylim([0,2])
plt.gca().add_patch(mpl.patches.Rectangle([{min_gini},0],
  {max_alpha} - {min_gini}, 2, color="aliceblue"))
plt.subplot(2, 2, 4)
plt.ylim([0,2])""".format(min_gini=min_gini, max_alpha=alpha_vals[-1]))
```

Done with figures.