Assignment 2

Jason Zhao u7493671

1 Plotting and fitting with gnuplot

First download the file using:

```
wget http://www.mso.anu.edu.au/~chfeder/teaching/
astr_4004_8004/material/mM4_10048_pdfs/
EXTREME_hdf5_plt_cnt_0050_dens.pdf_ln_data
```

Start from the provided gnuplot template:

```
wget http://www.mso.anu.edu.au/~chfeder/teaching/
astr_4004_8004/material/gnuplot.gp
```

Add a Gaussian fit and plot using:

```
# Gaussian fit
x0    = 1.0
sigma = 1.0
f(x) = 1/(sqrt(2.0*pi)*sigma) * exp(-(x-x0)**2/(2.0*sigma**2))
fit f(x) "EXTREME_hdf5_plt_cnt_0050_dens.pdf_ln_data" u 1:3 via x0, sigma

# Plot density PDF along with Gaussian fit
p [-10:10] [0:1] "EXTREME_hdf5_plt_cnt_0050_dens.pdf_ln_data" u 1:3 ls 1 t "Gas density PDF", \
    f(x) ls 16 t sprintf("Fit: x0=%.3f, sigma=%.3f", x0, sigma)
```

Finally save as a pdf file:

```
# Write out a pdf file with the plot
print outfile." created."
```

The result is shown in Figure 1.

2 Plotting multiple datasets and data manipulation with gnuplot

1. First download and extract the data using:

   ```
   wget -N "http://www.mso.anu.edu.au/~chfeder/teaching/
   astr_4004_8004/material/mM4_10048_pdfs/EXTREME_pdfs.tar.gz"
   tar xzf EXTREME_pdfs.tar.gz
   ```

   Starting from the template again, plot the three files together using:

   ```
   # Set axis
   set logscale y
   set yrange [1e-5:2]
   set xrange [-10:10]
   ```

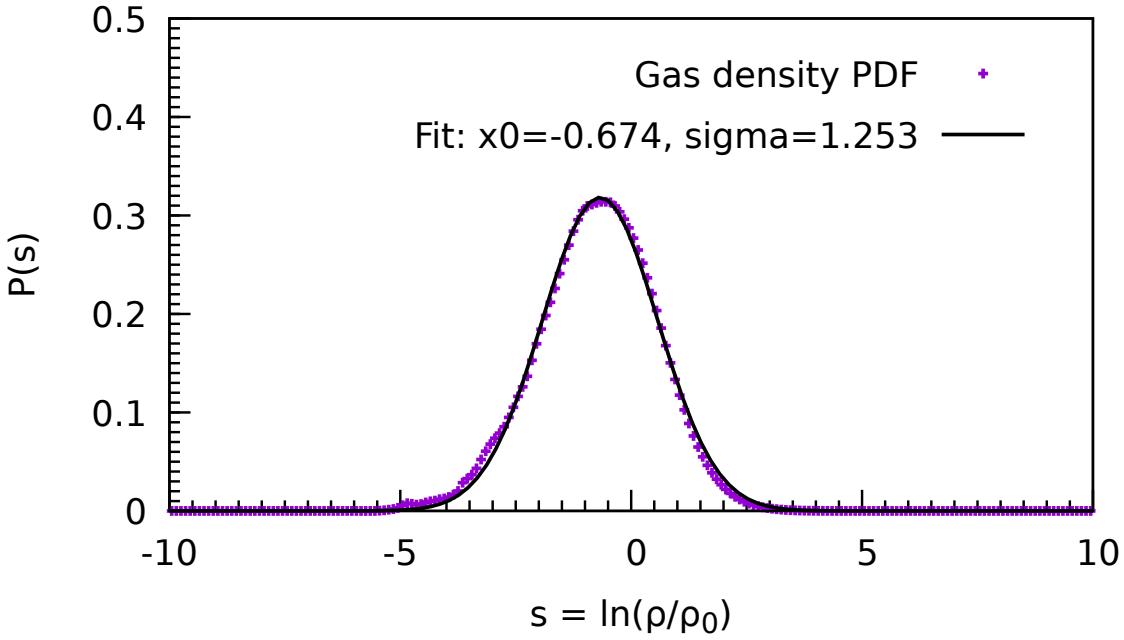

Figure 1: Gaussian fit for density PDF

```
# Read Data files
f20 = "EXTREME_hdf5_plt_cnt_0020_dens.pdf_ln_data"
f30 = "EXTREME_hdf5_plt_cnt_0030_dens.pdf_ln_data"
f40 = "EXTREME_hdf5_plt_cnt_0040_dens.pdf_ln_data"

# Plot datasets on the same plot
p \
    f20 u 1:3 ls 1 t "t=0020", \
    f30 u 1:(2.0*$3) ls 2 t "t=0030", \
    f40 u 1:(4.0*$3) ls 3 t "t=0040"

# Write out a pdf file with the plot
print outfile." created."
```

The result is shown in Figure 2.

2. Start from the script of Section 1, plot using:

```
# Axes
set xrange [-10:10]
set yrange [0:0.6]

f50 = "EXTREME_hdf5_plt_cnt_0050_dens.pdf_ln_data"

# Plot
p f50 u 1:(exp($1)*$3) ls 1 t "Mass-weighted PDF"

# Write out a pdf file with the plot
print outfile." created."
```

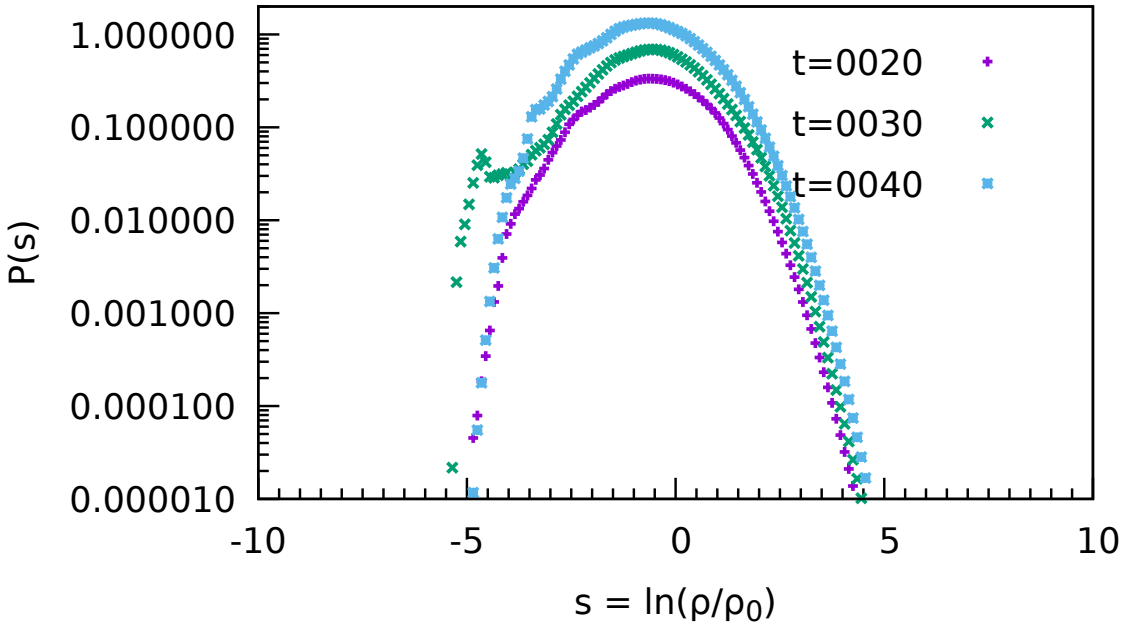

Figure 2: Three time files on the same plot

The result is shown in Figure 3.

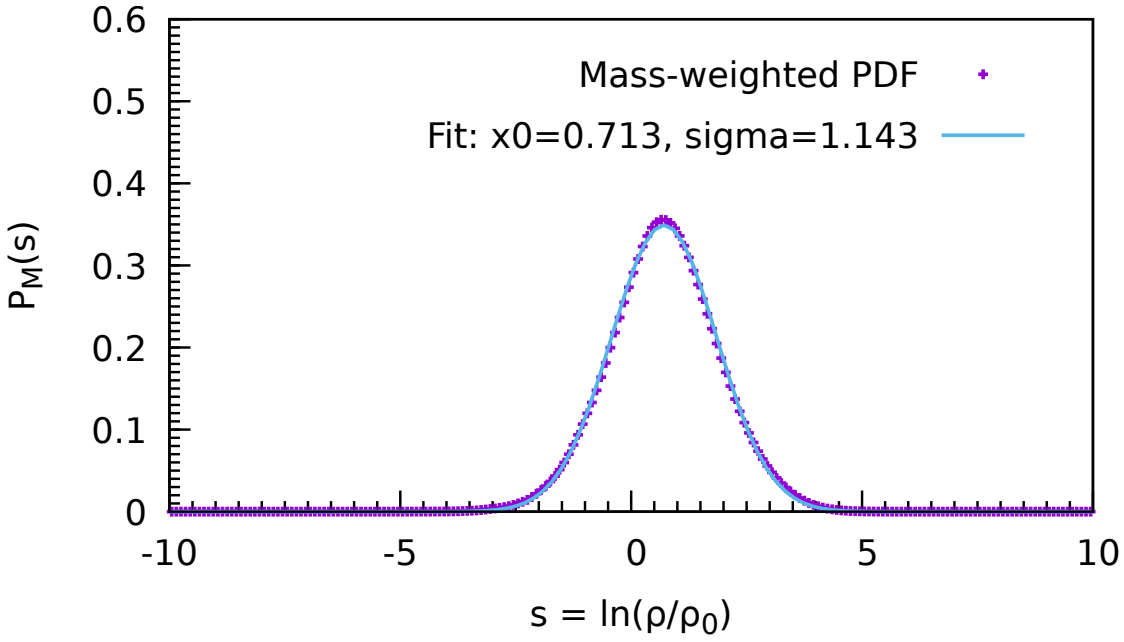

Figure 3: Mass weighted PDF

3. Perform Gaussian fit and print parameters using:

```
# Gaussian fit
x0     = 1.0
```

```
sigma = 1.0
f(x) = 1/(sqrt(2.0*pi)*sigma) * exp(-(x-x0)**2/(2.0*sigma**2))
fit f(x) f50 u 1:(exp($1)*$3) via x0, sigma

# Print fitted parameters to the shell
print sprintf("Fit results:")
print sprintf("mean (x0)= %.3f", x0)
print sprintf("sigma    = %.3f", sigma)

f(x) with lines ls 3 t sprintf("Fit: x0=%.3f, sigma=%.3f", x0, sigma)
```

The result is shown in Figure 3.

3 The stellar initial mass function

1. The Chabrier IMF is given by:

$$\frac{dN}{d\log_{10} M} = \begin{cases} 0.093 \exp\left[-\frac{(\log_{10} M - \log_{10} 0.2)^2}{2(0.55)^2}\right] & \text{for} \quad M \leq 1 \\ 0.041 M^{-1.35} & \text{for} \quad M > 1 \end{cases}$$

Using chain rule we have:

$$\begin{aligned} \text{IMF}(M) &= \frac{dN}{dM} \\ &= \frac{dN}{d\log_{10} M} \frac{d\log_{10} M}{dM} \\ &= \frac{1}{\ln(10) M} \frac{dN}{d\log_{10} M} \\ &= \begin{cases} \frac{0.093}{\ln(10) M} \exp\left[-\frac{(\log_{10} M - \log_{10} 0.2)^2}{2(0.55)^2}\right] & \text{for} \quad M \leq 1 \\ \frac{0.041}{\ln(10)} M^{-2.35} & \text{for} \quad M > 1 \end{cases} \end{aligned}$$

As required.

2. Script written as:

```
import numpy as np, matplotlib.pyplot as plt

# ---------- 2. ----------
# Define the Initial Mass Function (IMF)
def IMF(M_min, M_max, n_bins):
    # Create a logarithmically-scaled grid
    M_range = np.logspace(np.log10(M_min), np.log10(M_max), n_bins)
    IMF_values = []
    for M in M_range:
        if M <= 1:
            IMF_values.append(0.093/(np.log(10)*M) *
            np.exp(-(np.log10(M) - np.log10(0.2))**2 / (2*0.55**2)))
        else:
            IMF_values.append(0.041/np.log(10) * M**(-2.35))
    return np.array(M_range), np.array(IMF_values)
```

3. Plot the IMF using:

```
# --------- 3. ----------
# Plot IMF
n_bins = 1000
M_range, IMF_values = IMF(1e-2, 1e2, n_bins)
plt.figure(figsize=(10, 6))
plt.loglog(M_range, IMF_values, color='blue')
plt.xlabel(rf'M ($M_\odot$)')
plt.ylabel('IMF(M)')
plt.savefig('3.3.pdf')
```

The result is shown in figure 4.

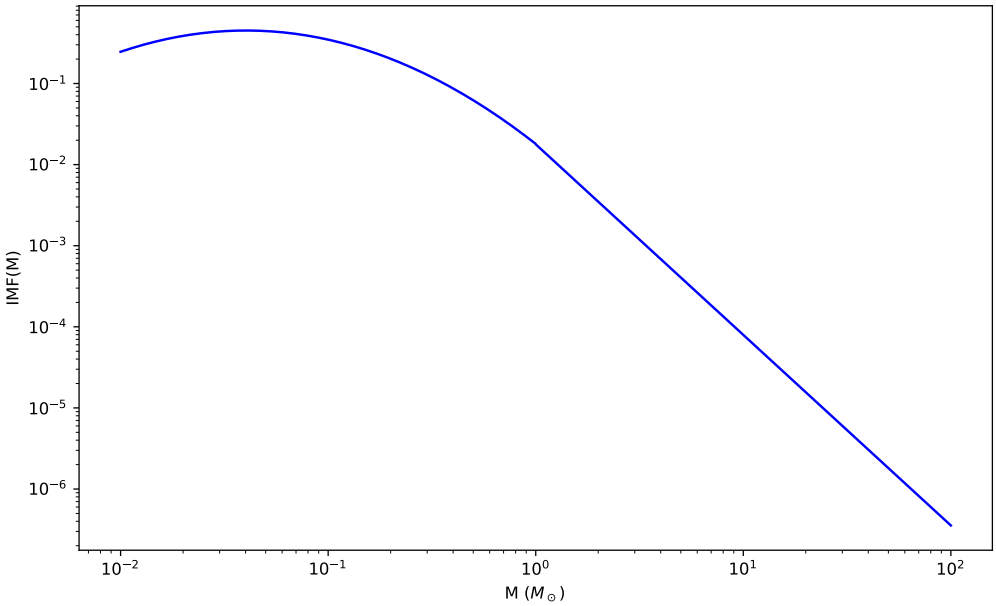

Figure 4: discretised IMF in log-log plot

4. The mode IMF mass is computed using:

```
# ---------- 4. ----------
# Compute the the mode mass of IMF(M)
mode_M = M_range[np.argmax(IMF_values)]
print(f'4. Mode mass of IMF(M): {mode_M}')
```

With the result: 4. Mode mass of IMF(M): 0.04023505548869293.

5. The mean mass can be computed by:

$$\langle M \rangle = \frac{\int M \frac{dN}{dM} dM}{\int \frac{dN}{dM} dM}$$

The staggered binning approach described in the assignment is equivalent to the trapezoidal rule, thus the numpy trapezoid function will be used for numerical integration:

```
# ---------- 5. ----------
```

```
# Compute the average mass of stars from IMF(M)
def average_mass(M_range, IMF_values):
    norm = np.trapezoid(IMF_values, M_range)
    mean_M = np.trapezoid(M_range * IMF_values, M_range) / norm
    return mean_M

mean_M = average_mass(M_range, IMF_values)
print(f'5. Average mass of stars from IMF(M): {mean_M}')
```

With the result: 5. Average mass of stars from IMF(M): 0.5458176702317216.

6. We can test how average star mass depends on the number of bins by approximate true average mass with a very high resolution, and iterate over a range of bin numbers to find the required number for 1% error, using the code:

```
# ---------- 6. ----------
# Test how average star mass depends on the number of bins
n_bins_range = np.logspace(1, 4, 100)

# Approximate true average mass with very high resolution
true_n_bins = int(1e7)
M_range_true, IMF_values_true = IMF(1e-2, 1e2, true_n_bins)
true_mean_M = average_mass(M_range_true, IMF_values_true)

# Compute average mass and error for different numbers of bins
average_masses = []
errors = []

for n_bins in n_bins_range:
    M_range, IMF_values = IMF(1e-2, 1e2, int(n_bins))
    mean_M = average_mass(M_range, IMF_values)
    error = abs(mean_M - true_mean_M) / true_mean_M * 100  # percentage error
    average_masses.append(mean_M)
    errors.append(error)

# Find number of bins needed for 1% error
for bins, err in zip(n_bins_range, errors):
    if err < 1:
        print(f'6. Minimum number of bins for <1% error: {int(bins)}')
        break

# Plot average mass vs number of bins
plt.figure(figsize=(10, 6))
plt.loglog(n_bins_range, average_masses, color='red')
plt.xlabel('Number of bins')
plt.ylabel('Average mass')
plt.savefig('3.6.pdf')
```

The plot is shown in figure 5, with the results: 6. Minimum number of bins for $< 1\%$ error: 11.

7. To compute the average mass for $M_{max} \to \infty$, we find the smallest M_{max} required for the average mass to converge within 2 significant figures using the code:

```
# ---------- 7. ----------
# Find average mass for M_max -> inf
Mmax_range = np.logspace(2, 5, 100)  # From 1e2 to 1e5
```

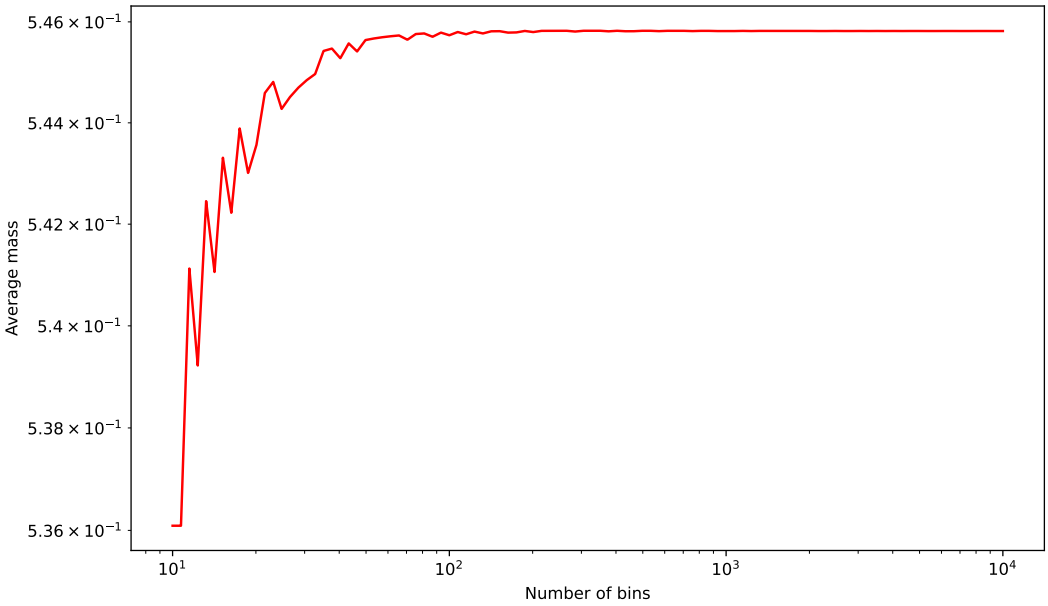

Figure 5: average mass as a function of the number of bins

```
average_masses_max = []

for M_max in Mmax_range:
    M_range, IMF_values = IMF(1e-2, M_max, 1000)
    mean_M = average_mass(M_range, IMF_values)
    average_masses_max.append(mean_M)

rounded_final_mass = round(average_masses_max[-1], 2)
for Mmax, avg_mass in zip(Mmax_range, average_masses_max):
    if round(avg_mass, 2) == rounded_final_mass:
        print(f'7. Average mass converges to {rounded_final_mass} at Mmax = {Mmax}')
        break
```

With the result: 7. Average mass converges to 0.62 at Mmax = 32745.491628777316.