delay_times

March 17, 2023

1 Study of the distribution of delay times of binary compact objects

1.1 Marco Giunta

2 Physical introduction

2.1 Binary compact objects and delay time t_d

Compact objects such as black holes and neutron stars hold the key to a lot of interesting physics: these objects are so extreme that they allow us to observe phenomena impossible to reproduce on earth. Binary compact objects are particularly interesting; for example the first detection of gravitational waves was thanks to a merger phenomenon between two inspiraling black holes. It is known that binary black holes usually form from massive binary systems; more generally most massive stars are in binary systems. In particular the binary systems where the two objects form/evolve together (as opposed to two initially independent objects that at some point become gravitationally bound) seem to be especially efficient in generating systems with black holes and/or neutron stars. For this reason it is interesting to study the simulated evolution of systems like these using *population synthesis codes*, so that we may e.g. infer general statistical properties about such systems. This problem is quite complex: the interaction between the two objects may happen via different mechanisms operating during many different significant evolutionary phases, and this complicates the already involved single-star evolution. For example there are several ways the two stars in a binary system may exchange matter (wind mass transfer, Roche lobe overflow, common envelope), with the effect of e.g. modifying important values like the masses of the final compact objects; processes like these are often described via semi-analytical models, that rely on free parameters to be given as input to the code.

We know that some binary systems merge in a finite time (i.e. in a time smaller than the age of the universe), while in others the two objects remain separate indefinitely. In the former case (i.e. eventual merger) there are several possible scenarios, involving 0, 1 or 2 compact objects in the final steps before merge. When there is a progenitor binary star system that evolves into a binary compact object the time between formation of the former and merger of the latter is called the *delay time*; the purpose of this work is to analyze how this quantity - when finite - depends on some key parameters of the system, that are described below.

2.2 Simulation free parameters: α common envelope and metallicity Z

The two parameters we will mostly focus on in the present work are the α *common envelope parameter* and the Z *metallicity parameter*; while the latter has a simple enough meaning the

former requires a nontrivial introduction, that will also allow us to recap different events that may occur during the evolution of a binary star system.

2.2.1 Roche Lobe dynamical instability and common envelope

One of the main mechanisms that allow mass transfer between the two objects is *Roche Lobe instability*, that may guide the evolution of the binary star system in several different directions; we will briefly describe it in order to define α.

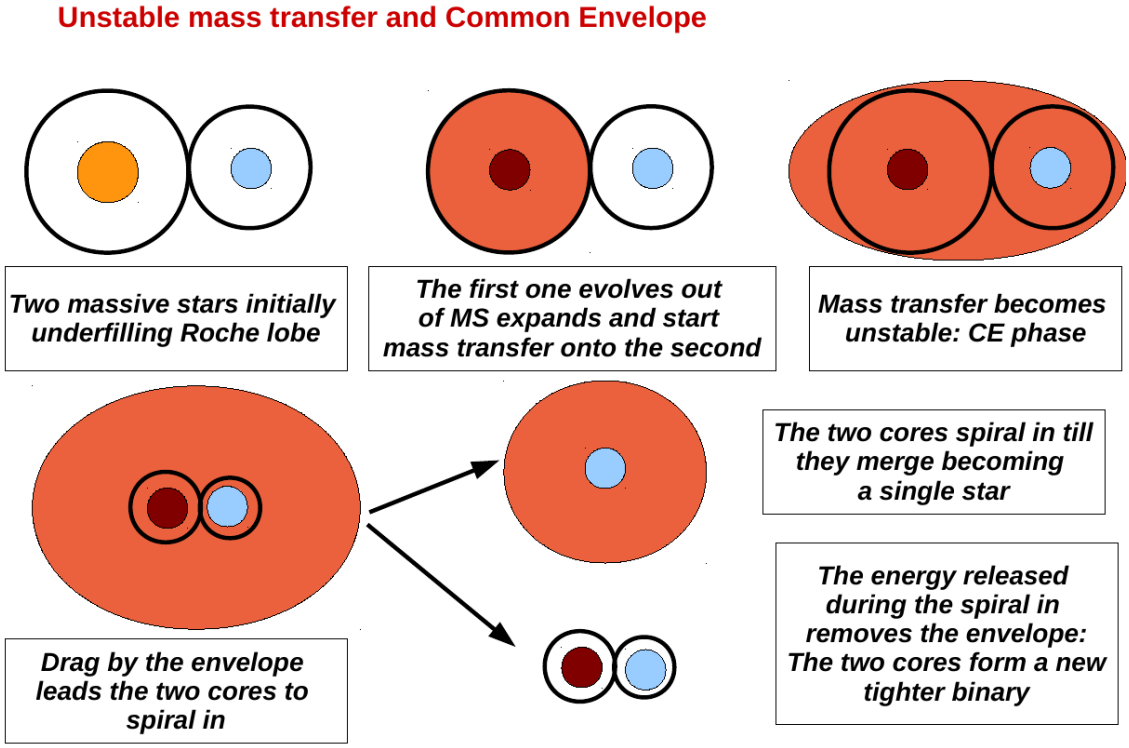
Imagine a particle orbitating near the edge of one of the stars; it will feel the gravitational attraction of both stars, but it will also feel an outward Coriolis force due to its rotation. Due to the superposition of these three potential the equipotential surfaces of the system have a very peculiar shape, that resembles an 8 figure in the space between/around the two stars; due to this if our particle is in a special position thanks to the shape of these surfaces it can flow freely from one star to the other, i.e. it can move from one star to another without an energy exchange. This leads to the concept of *Roche radius* as the distance from a star's center that allows this free/zero-energy movement. This is relevant because after exiting the main sequence a star can expand significantly, and if at some point its radius reaches Roche's critical value *matter from the edge of the star will be able to flow free falling towards the companion star*. This mechanism is often relevant because in later stages of their lives many stars expand filling their Roche lobe (i.e. the star radius becomes as large as the Roche one), which results in matter beginning to move from one star to the other; if *dynamical instability* is reached this mechanism can continue indefinitely, leading to phenomena like *common envelope* and *mergers* - which we now explain in more detail.

If one star starts to "donate" matter to the other the orbital separation will shrink; since $R_L \propto a$ the Roche radius will shrink, too. If the donor star is able to shrink faster than its Roche lobe the matter transfer will stop; otherwise an instability will take place, and more and more matter will be transferred as the stars get closer, leading to a final state where the outer gas shells are shared between the two stars. Under a Roche lobe instability two stars without cores will simply merge; if at least one has a heavy, dense core or is a compact object then the system will enter the *common envelope state*, where the dense objects are surrounded by a single gas shell. If we reach this state (unstable mass transfer with dense cores leading to common envelope) drag by the envelope will make the two objects closer as they spiral towards each other; this releases gravitational potential energy, that may or may not be used to unbind the envelope itself. In particular if the envelope is never ejected the two cores will eventually merge, becoming a single object; otherwise if the energy released during the inspiral is sufficient to eject the envelope the final state of the system will be a tighter binary. Notice that in all of these steps the cores may or may not be already compact objects: sometimes the two start as ones, and some others they later become e.g. black holes, maybe after ejecting the common envelope.

To recap: if a star expands filling its Roche lobe matter will start to flow freely (i.e. at no energy cost) towards the companion, leading to a decrease in orbital separation and therefore of the Roche radius itself. If the donor cannot shrink fast enough its radius will always be greater than its Roche lobe, leading to a dynamical instability i.e. a flow of matter and decrease in orbital separation that doesn't stop until the outer gas shells merge into a single *common envelope*. If there are no dense cores the two stars simply merge; otherwise the two cores will get closer and closer due to common envelope drag, and merge if the CE is not ejected. If the common envelope is ejected thanks to the energy released during the inspiralling phase the cores will simply become a tighter "naked" binary (that may still merge later via other processes, such as gravitational wave decay).

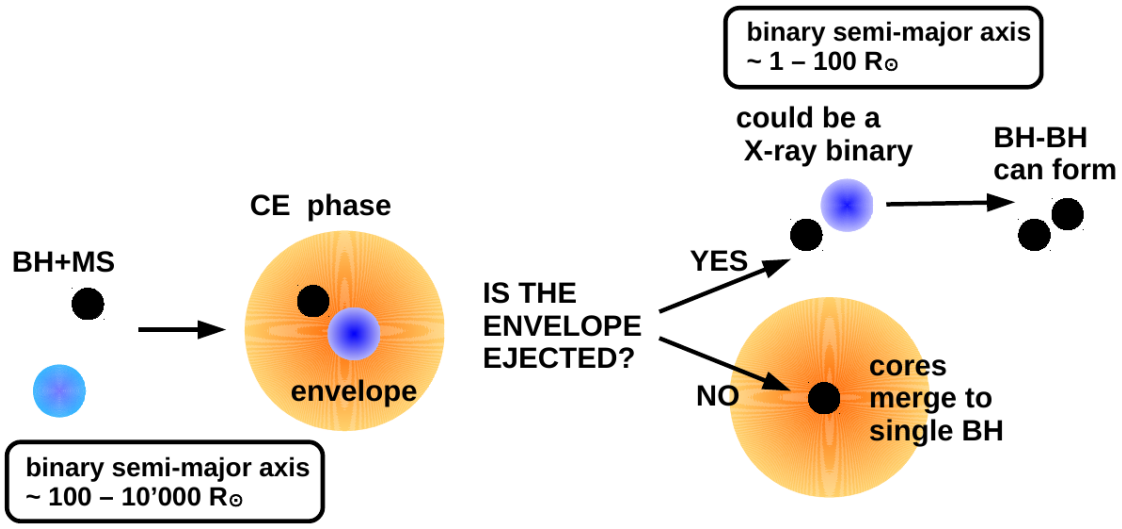
[1]:
```
from IPython.display import Image
Image('./images/common_envelope_dense_cores.png')
```

[1]:



[2]:
```
Image('./images/common_envelope_compact_objects.png')
```

[2]:



This complex physical interaction is often described in simulations using the α_{CE} *formalism*. Under this approach one assumes that the gravitational potential energy released by the inspiral of the cores is in part absorbed by the common envelope (e.g. by heating it up), and that sometimes *this single energy source* is enough to unbind the envelope. In particular we compute the initial gravitational binding energy of the common envelope as

$$E_i^{\rm bind} = -\frac{G}{\lambda}\left(\frac{M_1 M_1^{\rm env}}{r_1} + \frac{M_2 M_2^{\rm env}}{r_2}\right)$$

where λ is a geometrical factor to make this simple equation more realistic, and there are two terms because we assume that initially there may be two envelopes (if there is just one then one term disappears). Clearly in order to unbind the envelope it must absorb energy at least equal to the above; in the α formalism we assume this needed energy comes from the cores' binding energy:

$$E_{\rm orb} = -\frac{1}{2}\frac{G M_1^c M_2^c}{a}$$

As the two cores get closer a decreases and $E_{\rm orb}$ becomes more negative, reflecting the fact that they fall deeper in the potential well due to the positive energy they release; we assume that this released positive energy is in part absorbed by the envelope, i.e. the envelope absorbs an amount of energy equal to

$$\alpha \Delta E_{\rm orb} = \alpha (E_f^{\rm orb} - E_i^{\rm orb})$$

where we introduced an *efficiency parameter* α, i.e. a number between 0 and 1 that quantifies how efficient the energy removal/absorption is. This is the α common envelope parameter after which this model is named.

Clearly under these assumptions the common envelope will be ejected only if the above energy at least matches the initial binding energy, i.e. if at least the following condition is met:

$$E_i^{\rm bind} = \alpha \Delta E_{\rm orb}$$

To recap: in this model one assumes that the envelope is ejected if the amount of gravitational energy released by the cores during the inspiral (times an efficiency parameter) at least matches the initial envelope binding energy.

This model is actually incomplete: for example some observations suggest that we need to sometimes have $\alpha > 1$, even though according to the above this is unphysical. As unsatisfactory as this approach may be it still is one of the best; hence why many simulation codes use it. A good compromise is to use this formalism while leaving α as a *free parameter*, both in the sense that it is user-provided and that it may exceed 1 - so that one can in principle "catch" all actually physical situations. Indeed in what follows we will analyze data generated both with physical (between 0 and 1) and unphysical (beyond 1) α values.

As a final note we remark that in the following we will for simplicity use the symbol A instead of α.

2.2.2 Metallicity

When it comes to stellar evolution another relevant parameter is the *metallicity* Z, defined as the fraction of elements heavier than helium in a star's chemical composition. In `MOBSE` this is another free parameter, to be provided by the user in the initial conditions files.

2.2.3 Other processes and parameters

Of course a complete description of the problem's physics (or even just of the simulation codes' inner working) requires a lengthier discussion, revolving around other processes (e.g. supernova kick/mechanism models) and parameters (e.g. Maxwellian rms kick speed). For simplicity in the present work we will focus on just A and Z, leaving everything else as in the `MOBSE` default. From now on therefore we always treat A and Z as free parameters with a nontrivial effect on the system's evolution and ignore the other relevant ones.

3 Introduction

The delay time t_d of a COB is defined as the time between formation of the progenitor binary star and the time at which the two compact objects merge; the purpose of this notebook is to analyze how the delay time of a compact object binary system depends on its system's parameters.

In particular there are several interesting questions to be addressed: - Heuristical arguments suggest that the delay time's distribution is distributed as $1/t$, i.e. $\mathrm{d}N/\mathrm{d}t \propto 1/t$, and yet experimental evidence shows deviations for small t values. Can we quantify how strong these deviations are? - How is t influenced by the system's metallicity and/or α common envelope parameter, if at all? What about the type of system (i.e. BH-BH, NS-NS, BH-NS)? - Can we find simple fitting formulae for t?

To quantify the deviations of $\mathrm{d}N/\mathrm{d}t$ from $1/t$ we will produce some plots and compute statistics based on information theory and frequentist hypothesis testing; to study the dependence between t and A, Z or the type of binary system we will use basic statistics and information theory. Finally we will tackle the last problem using basic, interpretable algorithms.

[3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from os import listdir
import re
from matplotlib.cm import get_cmap
from scipy.stats import gaussian_kde
from scipy.integrate import quad
from itertools import product
from scipy.stats import ks_2samp
from scipy.interpolate import interp1d
from scipy.stats import linregress
from matplotlib.colors import LinearSegmentedColormap
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.tree import DecisionTreeRegressor
```

The dataset employed in this notebook has been generated using MOBSE with the default parameters and the provided initial conditions files (which implies that the metallicity in the dataset is either 0.002 or 0.0002); the free parameter A was set equal to $0.01, 0.1, 1, 3, 5, 10$, similarly to `Giacobbo & MM 2018` (as explained above unphysical $A > 1$ values are allowed).

[4]:
```
df_list = []
folder_list = [folder for folder in listdir('./data/') if folder.
 ↪startswith('MOBSE_')]

for folder in folder_list:
    A = float(re.findall('MOBSE_A(.+)_Z2e-[3-4]', folder)[0])
    Z = 2 * 10 ** (-int(folder[-1]))
    path = f'./data/{folder}/output/A{A}/{Z}/mergers_out.csv'
    tmp = pd.read_csv(path)
    tmp['A'] = A
    tmp['Z'] = Z
    tmp = tmp[['ID', 'A', 'Z', 'min1', 'min2', 'tform', 'sepform', 'eccform',␣
 ↪'k1form', 'm1form', 'k2form', 'm2form', 'tmerg', 'k1', 'm1', 'k2', 'm2',␣
 ↪'sep', 'ecc', 'label']]
    df_list.append(tmp)
```

[5]:
```
df = pd.concat(df_list, axis = 0)
df
```

[5]:

	ID	A	Z	min1	min2	tform	sepform	eccform	\
0	1152	0.01	0.0020	33.1212	29.9952	8.2941	8.0334	0.428470	
1	1406	0.01	0.0020	140.6925	109.7713	4.3999	6.0554	0.039460	
2	1730	0.01	0.0020	60.8764	26.8249	7.8760	28.1370	0.015422	
3	2642	0.01	0.0020	113.1747	66.1777	5.2060	32.5130	0.015831	
4	3769	0.01	0.0020	119.9626	41.8603	6.1373	9.2060	0.009211	
…	…	…	…	…	…	…	…	…	
58816	1999898	5.00	0.0002	11.4633	8.5653	33.4815	8.7363	0.935360	
58817	1999929	5.00	0.0002	15.7566	5.6165	38.4649	8.2960	0.870980	
58818	1999933	5.00	0.0002	8.3023	7.2136	49.1333	24.8040	0.960700	
58819	1999951	5.00	0.0002	17.1728	16.2198	14.0191	15.4520	0.887200	
58820	2000000	5.00	0.0002	22.9087	8.1225	22.6456	5.6983	0.723010	

	k1form	m1form	k2form	m2form	tmerg	k1	m1	k2	m2	\
0	14	11.1725	14	15.7688	76.7688	14	26.9413	15	16.2688	
1	14	41.1254	14	38.0058	6.1337	15	41.5341	14	79.1312	
2	14	17.6762	14	14.7456	11615.6035	15	18.0984	14	32.4219	
3	14	34.0775	14	33.2437	2353.3020	15	34.8363	14	67.3212	
4	14	27.3130	14	26.9694	34.9528	15	27.0728	14	54.2824	
…	…	…	…	…	…	..	…	..	…	
58816	13	1.2797	13	1.2899	259.0427	15	3.8223	13	2.5696	
58817	13	1.8542	13	1.2355	923.4760	14	3.0897	15	3.5445	
58818	13	1.2611	13	1.2355	3028.5356	13	2.4966	15	3.6697	
58819	13	2.1700	13	1.5211	3979.4468	14	3.6911	15	4.5999	
58820	14	3.4260	13	1.6358	488.0887	14	5.0619	15	5.0251	

	sep	ecc	label
0	0.0	0.0	COELESCE

```
1       0.0  0.0  COELESCE
2       0.0  0.0  COELESCE
3       0.0  0.0  COELESCE
4       0.0  0.0  COELESCE
…     …  …       …
58816   0.0  0.0  COELESCE
58817   0.0  0.0  COELESCE
58818   0.0  0.0  COELESCE
58819   0.0  0.0  COELESCE
58820   0.0  0.0  COELESCE

[269909 rows x 20 columns]
```

The mergers.out computed by MOBSE Contains synthetic information on compact object binaries (COBs) that merge within Hubble time, i.e. the ones with physical t_d. Notice that files have been pre-converted to the `csv` format for easy `pandas` access.

The meaning of the columns in the above dataframe is as follows: - min1/2: ZAMS mass of primary/secondary star (units = M sun) - tform: formation time of the second compact object, Myr=1e6 yr - sepform: semi-major axis at formation of the second compact object formation (R sun 6.95e10 cm) - eccform: orbital eccentricity at formation of the second compact object formation - k1form/k2form: type of primary/secondary compact object (13 = neutron star, 14 = black hole) - m1form/m2form: mass of the primary/secondary compact object /Msun - tmerg: merger time, Myr - k1: final type of star 1, after the merger (13 = NS, 14 = BH, 15 = star does not exist anymore) - m1: final mass of star 1 (could be zero or m1form+m2form in Msun) - k2: final type of star 2, after the merger (13 = NS, 14 = BH, 15 = star does not exist anymore) - m2: final mass of star 2 (could be zero or m1form+m2form in Msun) - sep: final orbital separation (should be zero, because compact object merged) - ecc: final eccentricity (should be zero) - label: status of the binary (COELESCE: merged system)

Notice that the last 3 columns are identically equal to zero/COELESCE, since by definition this file only contains systems that satisfy these equalities; therefore these columns are useless and we can safely discard them.

[6]:
```
# to confirm these columns are constant:
print((df.sep == 0.0).all())
print((df.sep == 0.0).all())
print((df.label == 'COELESCE').all())
```

```
True
True
True
```

[7]:
```
# we remove constant columns
df = df.drop(['sep', 'ecc', 'label'], axis = 1)
```

[8]:
```
# A/Z values used to generate the dataset
A_vec, Z_vec = np.sort(df.A.unique()), np.sort(df.Z.unique())
```

```
print('unique A values: ', A_vec)
print('unique Z values: ', Z_vec)
```

```
unique A values:  [ 0.01  0.1   1.    3.    5.   10.  ]
unique Z values:  [0.0002 0.002 ]
```

Finally since later on we will want to discriminate systems depending on their type (BH-NS, NS-NS or BH-BH) we add an integer column taking on values 0, 1 or 2 depending on the system type, which can be inferred by comparing the `k1form` and `k2form` columns.

[9]:
```
df['bin_system_type'] = 0

df.loc[(df.k1form != df.k2form), 'bin_system_type'] = 0 # ns/bh or bh/ns
df.loc[(df.k1form == df.k2form) & (df.k1form == 13), 'bin_system_type'] = 1 #␣
 ↪double ns
df.loc[(df.k1form == df.k2form) & (df.k1form == 14), 'bin_system_type'] = 2 #␣
 ↪double bh

bst_types = {0:'BH-NS', 1:'NS-NS', 2:'BH-BH'} # useful to convert our "one-hot␣
 ↪encoding" to a human-readable format

df
```

[9]:

	ID	A	Z	min1	min2	tform	sepform	eccform	\
0	1152	0.01	0.0020	33.1212	29.9952	8.2941	8.0334	0.428470	
1	1406	0.01	0.0020	140.6925	109.7713	4.3999	6.0554	0.039460	
2	1730	0.01	0.0020	60.8764	26.8249	7.8760	28.1370	0.015422	
3	2642	0.01	0.0020	113.1747	66.1777	5.2060	32.5130	0.015831	
4	3769	0.01	0.0020	119.9626	41.8603	6.1373	9.2060	0.009211	
…	…	…	…	…	…	…	…	…	
58816	1999898	5.00	0.0002	11.4633	8.5653	33.4815	8.7363	0.935360	
58817	1999929	5.00	0.0002	15.7566	5.6165	38.4649	8.2960	0.870980	
58818	1999933	5.00	0.0002	8.3023	7.2136	49.1333	24.8040	0.960700	
58819	1999951	5.00	0.0002	17.1728	16.2198	14.0191	15.4520	0.887200	
58820	2000000	5.00	0.0002	22.9087	8.1225	22.6456	5.6983	0.723010	

	k1form	m1form	k2form	m2form	tmerg	k1	m1	k2	m2	\
0	14	11.1725	14	15.7688	76.7688	14	26.9413	15	16.2688	
1	14	41.1254	14	38.0058	6.1337	15	41.5341	14	79.1312	
2	14	17.6762	14	14.7456	11615.6035	15	18.0984	14	32.4219	
3	14	34.0775	14	33.2437	2353.3020	15	34.8363	14	67.3212	
4	14	27.3130	14	26.9694	34.9528	15	27.0728	14	54.2824	
…	…	…	…	…	…	..	…	..	…	
58816	13	1.2797	13	1.2899	259.0427	15	3.8223	13	2.5696	
58817	13	1.8542	13	1.2355	923.4760	14	3.0897	15	3.5445	
58818	13	1.2611	13	1.2355	3028.5356	13	2.4966	15	3.6697	
58819	13	2.1700	13	1.5211	3979.4468	14	3.6911	15	4.5999	
58820	14	3.4260	13	1.6358	488.0887	14	5.0619	15	5.0251	

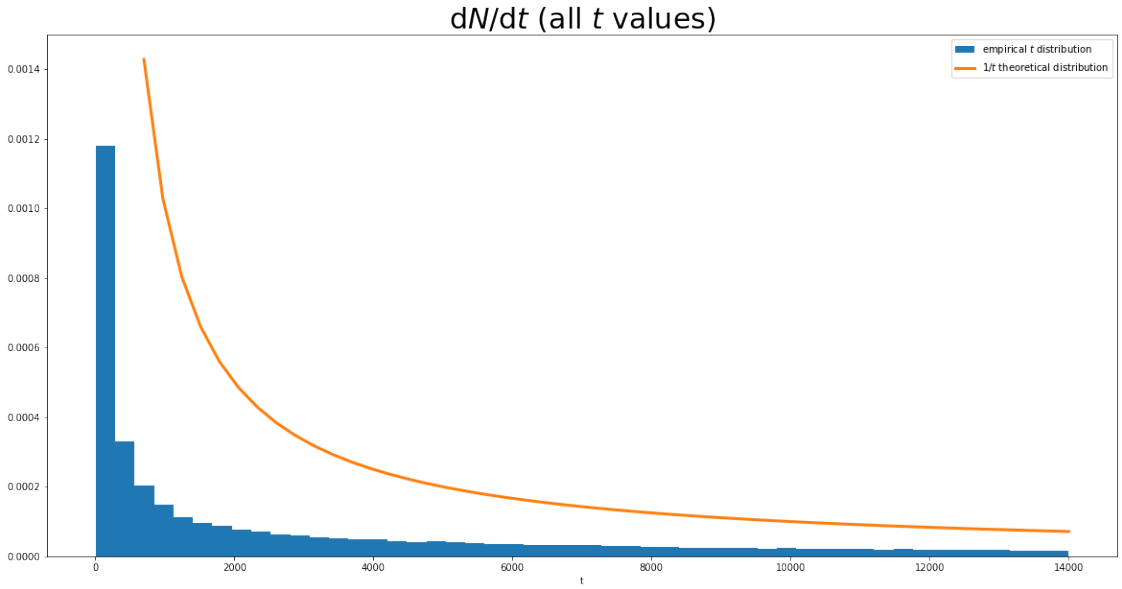
```
       bin_system_type
0                    2
1                    2
2                    2
3                    2
4                    2
…                    …
58816                1
58817                1
58818                1
58819                1
58820                0

[269909 rows x 18 columns]
```

4 Plotting $\mathrm{d}N/\mathrm{d}t$

4.1 Histogram plot in real space

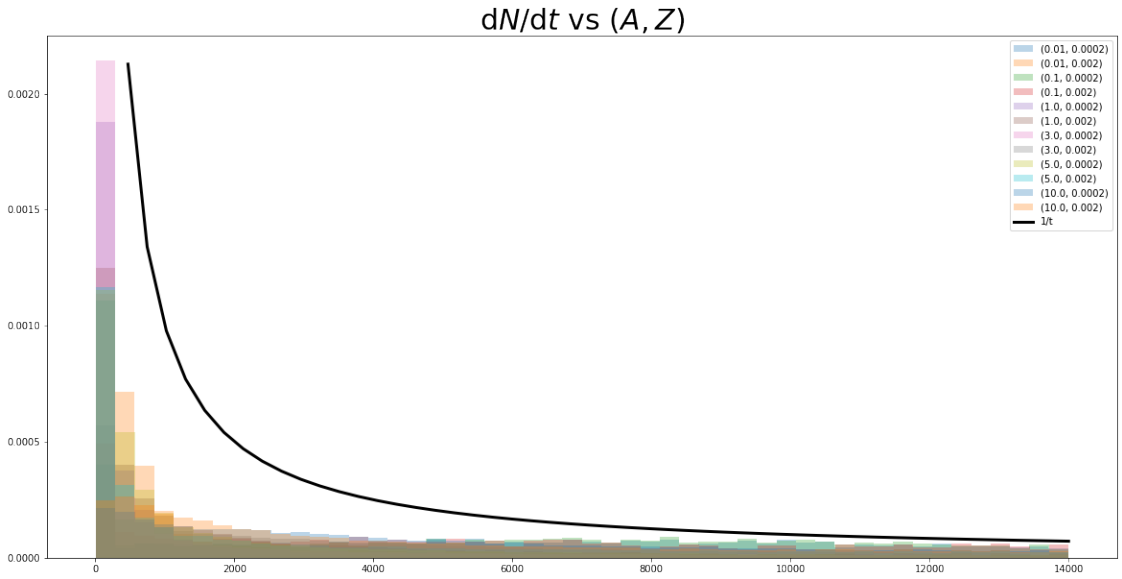
Let us start by plotting the histogram of t's density.

[10]:
```
plt.figure(figsize = (20, 10))
plt.hist(df.tmerg, bins = 50, density = True, label = 'empirical $t$␣
 ↪distribution')
tv = np.linspace(700, 14e3)
plt.plot(tv, 1/tv, label = '$1/t$ theoretical distribution', linewidth = 3)
plt.xlabel('t')
plt.legend(loc = 1)
plt.title(r'$\mathrm{d}N/\mathrm{d}t$ (all $t$ values)', size = 30);
```

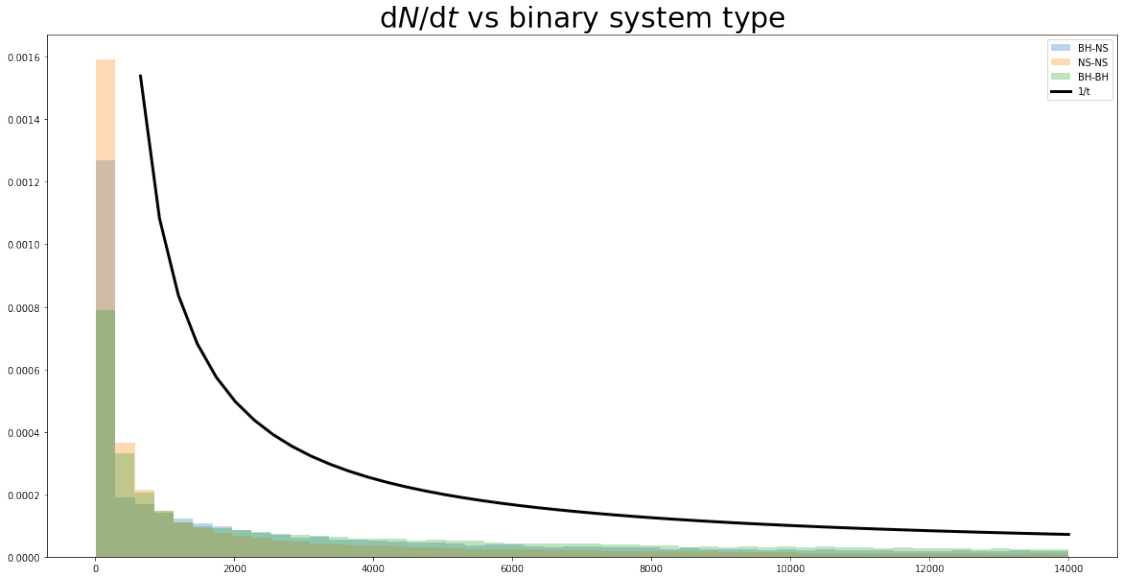


Below we repeat the previous histogram plot but grouping by (A, Z) or by binary system type.

[11]:
```
fig, ax = plt.subplots(1, 1, figsize = (20, 10))
tv = np.linspace(470, 14e3)

for A in A_vec:
    for Z in Z_vec:
        ax.hist(df.loc[(df.A == A) & (df.Z == Z)].tmerg, bins = 50, label =␣
 ↪f'{(A, Z)}', alpha = 0.3, density = True)

ax.plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
ax.legend()
ax.set_title(r'$\mathrm{d}N/\mathrm{d}t$ vs $(A, Z)$', size = 30)
ax;
```



[12]:
```
fig, ax = plt.subplots(1, 1, figsize = (20, 10))
tv = np.linspace(650, 14e3)

for bst in range(3):
    ax.hist(df.loc[df.bin_system_type == bst, 'tmerg'], bins = 50, label =␣
 ↪f'{bst_types[bst]}', alpha = 0.3, density = True)

ax.plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
ax.legend()
ax.set_title(r'$\mathrm{d}N/\mathrm{d}t$ vs binary system type', size = 30)
ax;
```



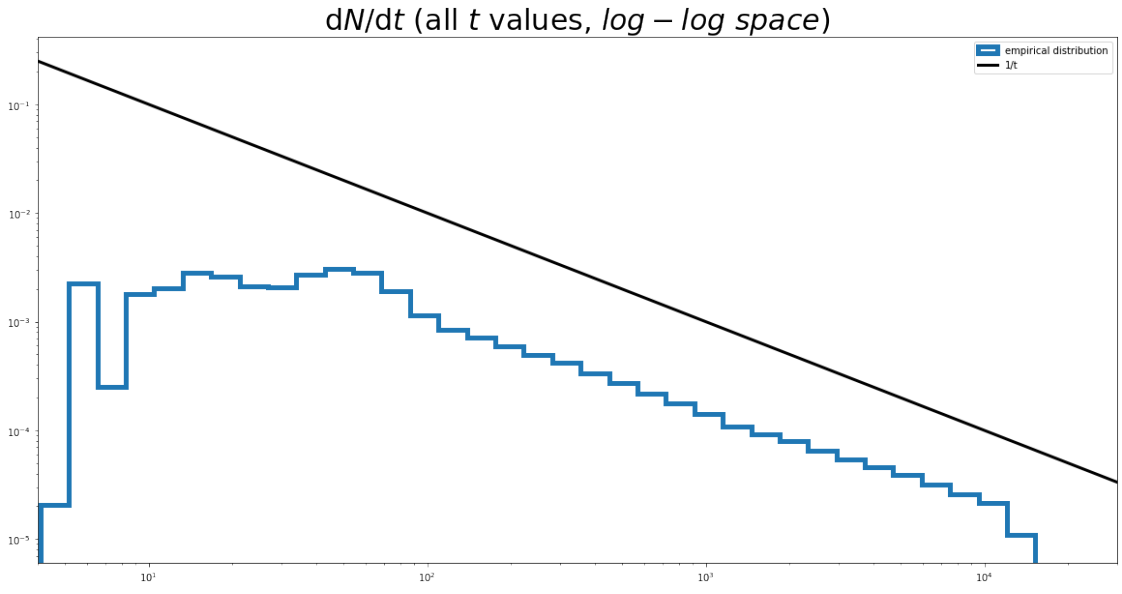
By looking at any of the above histograms we cannot see clearly the deviations from $1/t$, as these are significant only for small t values - which are overshadowed by the others. This is due to the huge range in t values on the horizontal axis; similarly the leftmost bin appears so tall that the plot needs to have a large vertical range, which in turn makes the other bins' heights too close to each other for us to be able to discern them clearly.

4.2 Log-log plot

To make the plot more legible we can repeat it in its *logarithmic* version, i.e. by plotting the logarithms of all values on both axes. This has the effect of linearly plotting the orders of magnitude of the values, instead of the value themselves - which in turn "compresses" nonlinearly both axes.

[13]:
```
num_plots = 12

cmap = get_cmap('inferno')
colors = [cmap(i) for i in np.linspace(0, num_plots, num_plots) / num_plots]

fig, ax = plt.subplots(1, 1, figsize = (20, 10))

# in order to obtain fixed-width bins in a logarithmic plot we need␣
 ↪exponentially increasing bin widths.
# This "linear spacing in log space" can be achieved via np.logspace; we only␣
 ↪need to convert the extrema we'd use with np.linspace.
# Adapting https://gist.github.com/hamishmorgan/3342260 :
MIN_VALUE = 1
MAX_VALUE = 14e3

# Log base we shall be using though-out
```

```
XBASE = 10
YBASE = 10

# Calculate the min and max powers:
start_power = int(np.floor(np.log(MIN_VALUE) / np.log(XBASE)))
end_power = int(np.ceil(np.log(MAX_VALUE) / np.log(XBASE)))

num_bins = 50
bins = np.logspace(start_power, end_power, num_bins, base = XBASE)

ax.hist(df.tmerg, bins = bins, label = 'empirical distribution', histtype =␣
 ↪'step', linewidth = 5,
        density = True, log = True)


tv = np.linspace(4, 1e5)
ax.plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
ax.set_xscale('log')
ax.set_xlim((4, 3e4))
ax.legend()
ax.set_title(r'$\mathrm{d}N/\mathrm{d}t$ (all $t$ values, $log-log \ space$)',␣
 ↪size = 30)
ax;
```



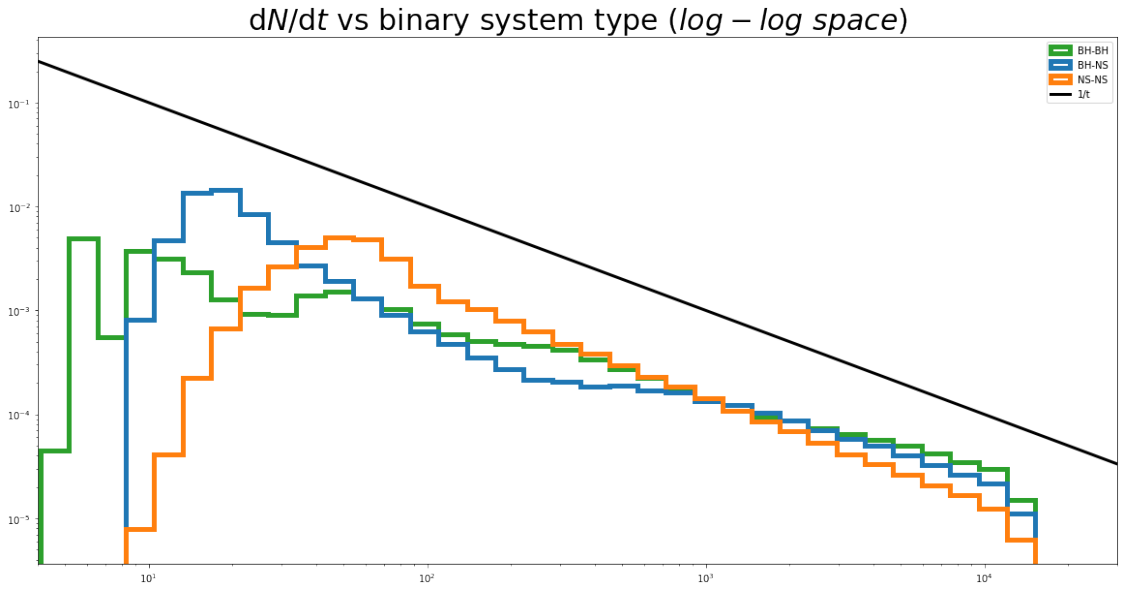
Now the each order of magnitude occupies the same horizontal space in the above plot. Due to this the small t values (i.e. those of order $10^1 - 10^2$) can be more clearly seen; similarly thanks to the logarithm on the vertical axis we can reduce the height difference between bins and better appreciate the small differences between most bins (previously the tallest bin overshadowed the

others).

Indeed we notice that as t decreases every distributions drops instead of increasing, as we would expect from the $1/t$ power law; we also observe peaks and valleys, corresponding to more general deviations from the $1/t$ law.

We also remark that taking the logarithm has straightened the $1/t$ function, turning it into a line with -1 slope. This is because in log-log space power laws become straight lines; this general property will be described more in detail later, as it will simplify the fitting procedure. For now it suffices to notice that at large t values the distribution mostly aligns to the black line, corresponding to is counterpart in real space aligning itself with the hyperbole; at small t values, instead, we observe a general trend of increasing before decreasing according to the black line, confirming the failure of the $1/t$ heuristics at small t values.

[14]:
```
fig, ax = plt.subplots(1, 1, figsize = (20, 10))

MIN_VALUE = 1
MAX_VALUE = 14e3

XBASE = 10
YBASE = 10

colors_bst_list = ['C0', 'C1', 'C2']

# Calculate the min and max powers:
start_power = int(np.floor(np.log(MIN_VALUE) / np.log(XBASE)))
end_power = int(np.ceil(np.log(MAX_VALUE) / np.log(XBASE)))

num_bins = 50
bins = np.logspace(start_power, end_power, num_bins, base=XBASE)

for bst in [2, 0, 1]:
    ax.hist(df.loc[df.bin_system_type == bst, 'tmerg'], bins = bins, label =␣
 ↪f'{bst_types[bst]}', histtype = 'step', linewidth = 5,
            density = True, log = True, color = colors_bst_list[bst])

tv = np.linspace(4, 1e5)
ax.plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
ax.set_xscale('log')
ax.set_xlim((4, 3e4))
ax.legend()
ax.set_title(r'$\mathrm{d}N/\mathrm{d}t$ vs binary system type ($log-log \␣
 ↪space$)', size = 30)
ax;
```

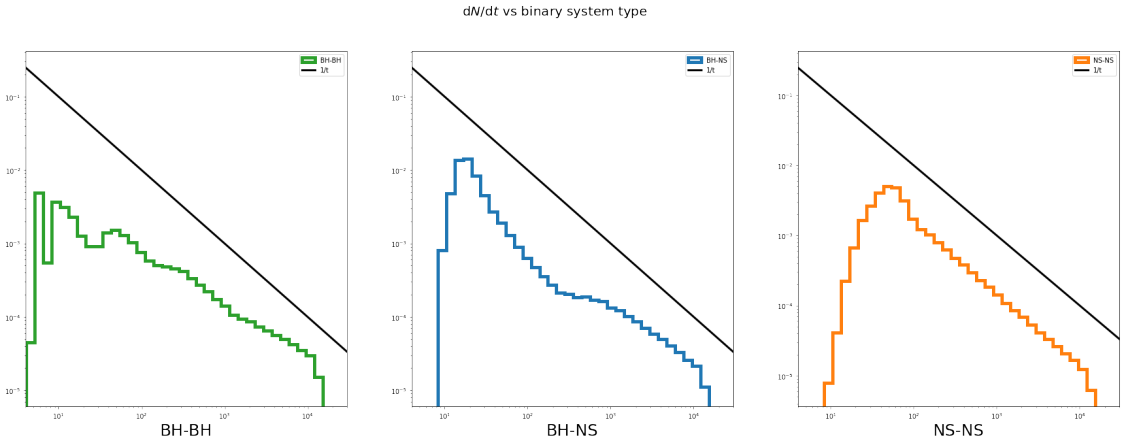


If we group by binary system type we observe the same trend as before, i.e. the distribution first increases then starts to decrease approximately linearly.

An interesting result is that the more neutron stars the system has, the larger the delay times tend to be i.e. on average according to this data black holes tend to accelerate the merging process. Notice that indeed systems with two black holes have on average smaller delay times, but their distribution is more irregular for small t compared to the systems 1-2 with neutron stars; the green histogram has 3 peaks instead of 1 like the others.

To better appreciate the (ir)regularity of the histograms we can also plot them separately:

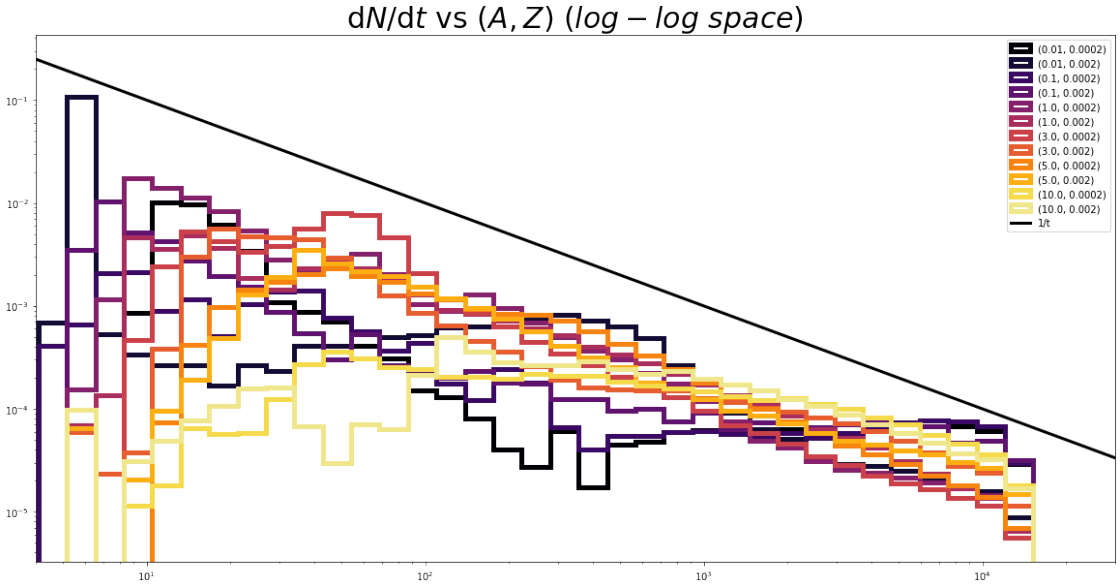
[15]:
```
fig, ax = plt.subplots(1, 3, figsize = (30, 10))
tv = np.linspace(4, 1e5)

for i, bst in enumerate([2, 0, 1]):
    ax[i].hist(df.loc[df.bin_system_type == bst, 'tmerg'], bins = bins,␣
 ↪histtype = 'step', linewidth = 5,
                  density = True, log = True, color = f'C{bst}', label =␣
 ↪bst_types[bst])
    ax[i].set_xscale('log')
    ax[i].set_xlim((4, 3e4))
    ax[i].set_xlabel(f'{bst_types[bst]}', size = 25)
    ax[i].plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
    ax[i].legend()

fig.suptitle(r'$\mathrm{d}N/\mathrm{d}t$ vs binary system type', size = 20)
ax;
```



Let us now plot in log-log space the delay time distributions grouped by the (A, Z) values.

[16]:
```
num_plots = 12

cmap = get_cmap('inferno')
colors = [cmap(i) for i in np.linspace(0, num_plots, num_plots) / num_plots]
colors[-1] = 'khaki'

fig, ax = plt.subplots(1, 1, figsize = (20, 10))

MIN_VALUE = 1
MAX_VALUE = 14e3

XBASE = 10
YBASE = 10

# Calculate the min and max powers:
start_power = int(np.floor(np.log(MIN_VALUE) / np.log(XBASE)))
end_power = int(np.ceil(np.log(MAX_VALUE) / np.log(XBASE)))

num_bins = 50
bins = np.logspace(start_power, end_power, num_bins, base=XBASE)

i = 0

for A in A_vec:
    for Z in Z_vec:
        ax.hist(df.loc[(df.A == A) & (df.Z == Z), 'tmerg'], bins = bins, label␣
 ↪= f'{(A, Z)}', histtype = 'step', linewidth = 5, color = colors[i],
                density = True, log = True)
        i += 1
```

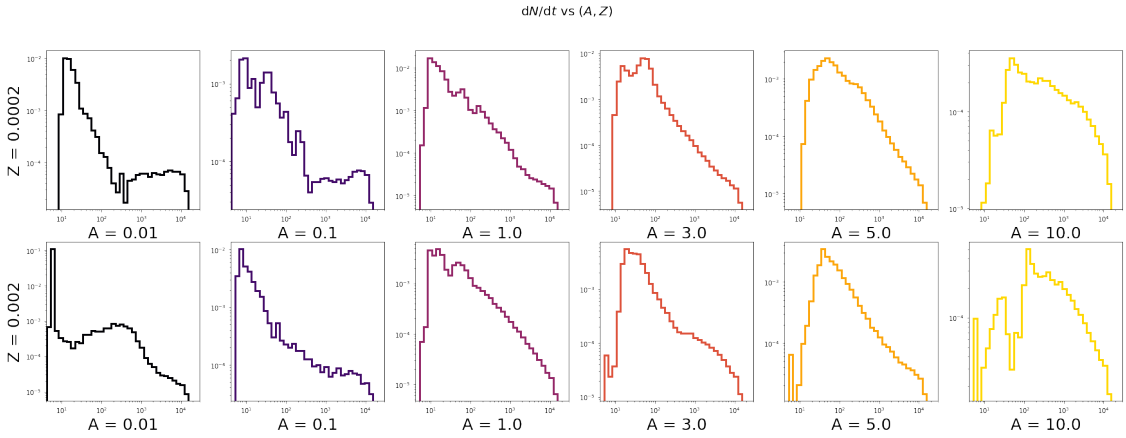
```
tv = np.linspace(4, 1e5)
ax.plot(tv, 1/tv, linewidth = 3, label = '1/t', color = 'black')
ax.set_xscale('log')
ax.set_xlim((4, 3e4))
ax.legend()
ax.set_title(r'$\mathrm{d}N/\mathrm{d}t$ vs $(A, Z)$ ($log-log \ space$)', size␣
 ↪= 30)
ax;
```



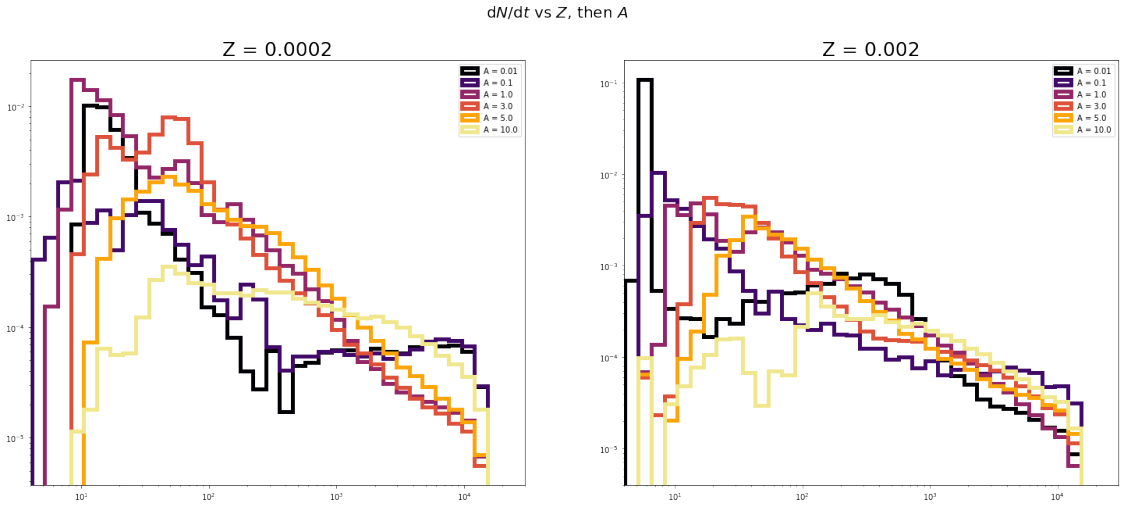
In the above plot it is quite hard to analyze the overall trends, both because there are too many lines and because the color gradient only has meaning if it represents one variable increasing, not two; to tackle both issues we plot each curve separately, then again by grouping by metallicity/common envelope parameter.

[17]:
```
num_plots = 6
fig, ax = plt.subplots(len(Z_vec), len(A_vec), figsize = (30, 10))
cmap = get_cmap('inferno')
colors = [cmap(i) for i in np.linspace(0, num_plots, num_plots) / num_plots]
colors[-1] = 'gold'

for j, Z in enumerate(Z_vec):
    for i, A in enumerate(A_vec):
        ax[j, i].hist(df.loc[(df.A == A) & (df.Z == Z)].tmerg, bins = bins,␣
 ↪histtype = 'step', linewidth = 3,
                      density = True, log = True, color = colors[i]) # alpha = 0.
 ↪5,
        ax[j, i].set_xscale('log')
        ax[j, i].set_xlim((4, 3e4))
```

```
        ax[j, i].set_xlabel(f'A = {A}', size = 25)
    ax[j, 0].set_ylabel(f'Z = {Z}', size = 25)

fig.suptitle(r'$\mathrm{d}N/\mathrm{d}t$ vs $(A, Z)$', size = 20)
ax;
```



[18]:
```
num_plots = 6

cmap = get_cmap('inferno')
colors = [cmap(i) for i in np.linspace(0, num_plots, num_plots) / num_plots]
colors[-1] = 'khaki'

fig, ax = plt.subplots(1, len(Z_vec), figsize = (25, 10))

# same bins
for j, Z in enumerate(Z_vec):
    for i, A in enumerate(A_vec):
        ax[j].hist(df.loc[(df.A == A) & (df.Z == Z)].tmerg, bins = bins, label␣
 ↪= f'A = {A}', histtype = 'step', linewidth = 5, color = colors[i],
                   density = True, log = True)

    ax[j].set_xscale('log')
    ax[j].set_xlim((4, 3e4))
    ax[j].legend()
    ax[j].set_title(f'Z = {Z}', size = 25)

fig.suptitle(r'$\mathrm{d}N/\mathrm{d}t$ vs $Z$, then $A$', size = 20)
ax;
```



Can larger A values slow down mergers? For all metallicity values we notice that increasing A tends to "move mass" to the right: the peak of the distribution moves towards larger t values while becoming shorter, i.e. both the mean and the mode move towards larger t values - a possible interpretation of this result is the following. Imagine that the energy needed to unbind the common envelope is kept fixed; if we increase A we are increasing the efficiency with which the common envelope will "steal" energy from the inspiraling cores - which means that in order to reach the same target energy we need a smaller energy budget. To be more specific: as A increases the system's common envelope will need to absorb on average a smaller and smaller fraction of the $\Delta E_{\rm orb}$ energy released when the cores get closer, as follows form the envelope unbind condition

$$E_i^{\rm bind} = \alpha \Delta E_{\rm orb}$$

If we imagine that not only the target envelope unbind energy $E_i^{\rm bind}$ is fixed but also the total *maximum* available energy $\Delta E_{\rm orb}$ then it is clear that increasing α makes the above condition easier to reach: larger $\alpha \implies$ smaller maximum $\Delta E_{\rm orb}$ needed to reach $E_i^{\rm bind}$.

This means that *systems with larger A will on average eject their common envelopes more easily.* This is relevant because if a system ejects its common envelope it will only be able to use gravitational wave decay to get the cores to be closer and therefore to eventually merge; if instead the common envelope is kept for a longer time the drag on the cores due to the envelope itself will help "steal" energy from the cores and get them closer. This line of reasoning may be a possible explanation of the above result.

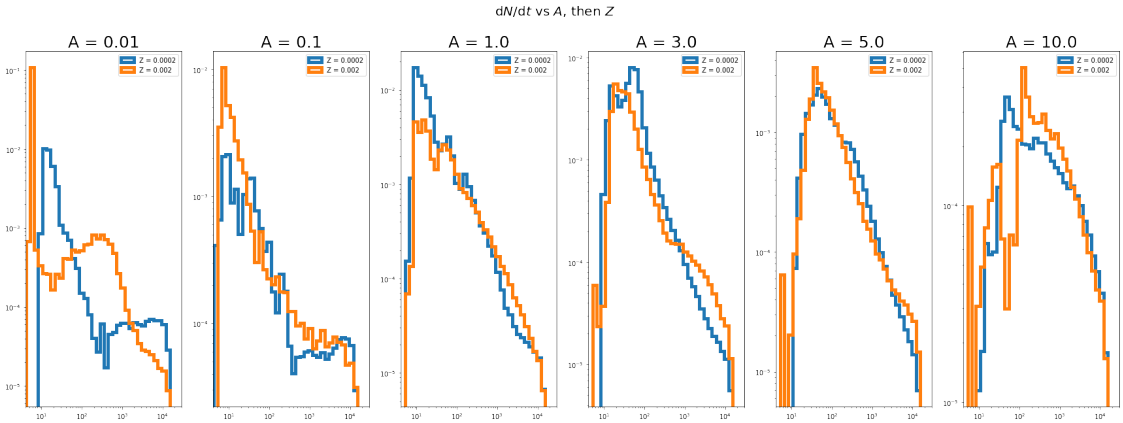
To recap: on average increasing α makes ejecting the common envelope easier (as a smaller fraction of the total energy budget needs to be absorbed by the CE), which in turns removes CE drag from the system, i.e. a mechanism that can speed up the merger; without it this task is left to gravitational wave decay only. For this reason we expect to see larger delay times on average when A increases, and this seems to be consistent with the above results.

Other A trends We notice that the smallest A values (0.01 and 0.1) tend to make the distribution more irregular: instead of having a sharp increase and then a mostly linear decrease (consistently

with the almost $1/t$ law) they have several peaks/valleys. This characteristic is actually shared by the $A = 10$ graphs; in this case we also notice a gentler drop at large t values, i.e. there are "too many" large delay times compared to the expected ones - again, this seems to be consistent with the "drag-less" argument discussed above.

The remaining intermediate A values align quite nicely with the $1/t$ law with tiny t values cut off - consistent with the most basic semi-heuristical prescription discussed throughout this work.

[19]:
```
colors_Z = ['C0', 'C1'] #['blue', 'orange']
fig, ax = plt.subplots(1, len(A_vec), figsize = (30, 10))

# same bins
for i, Z in enumerate(Z_vec):
    for j, A in enumerate(A_vec):
        ax[j].hist(df.loc[(df.A == A) & (df.Z == Z)].tmerg, bins = bins, label␣
 ↪= f'Z = {Z}', histtype = 'step', linewidth = 5, color = colors_Z[i],
                    density = True, log = True)

        ax[j].set_xscale('log')
        ax[j].set_xlim((4, 3e4))
        ax[j].legend()
        ax[j].set_title(f'A = {A}', size = 25)

fig.suptitle(r'$\mathrm{d}N/\mathrm{d}t$ vs $A$, then $Z$', size = 20)
ax;
```



When it comes to comparing different Z values it is harder to pinpoint a well-defined trend. We notice that the "irregular" A values $(0.01, 0.1, 10)$ all have distributions that become more peaked if we increase Z, with more mass on the left on average ($A = 10$ is the exception to this). For the "regular" A values $(1, 3, 5)$ we notice that the metallicity doesn't seem to affect the distribution much, since all available Z values lead to almost the same, almost $1/t$-like distribution. In general there do not seem to be clearly distinguishable trends; this may change if we were to add more Z values to our dataset.

5 Does t depend on $A/Z/BST$?

Physically we expect the delay time to depend on A, Z and the type of binary system: - A is the free parameter that measures the *energy removal efficiency* in the expression of the orbital energy needed to unbind the envelope; as discussed above our reasonable expectation that it may influece the delay time is backed by data. - Similarly we expect the matter composition (in part described by Z) and the type of compact objects in the binary system (described by BST) to influence t.

We qualitatively observed the impact of these relations above; now let us explore them with a quantitative approach.

5.1 Normalized covariance: r coefficient

A simple tool to estimate the influence of one variable over another is to compute the *covariance*, or even better the *Pearson correlation matrix*, where the off-diagonal terms are normalized between 0 and 1 (in absolute value) and thus provide a more interpretable quantity.

In particular the r coefficients (off-diagonal Pearson correlation matrix terms) assign a score to the dependence between pairs of variables *assuming a linear dependence*, i.e. $|r| \approx 1$ is an almost perfect linear dependence, while $r \approx 0$ means almost no linear dependence. Of course we expect the relation between variables to be highly nonlinear, given the complex physical nature of the dataset; still computing these coefficients is a natural starting point, since they're trivial to obtain.

[20]:
```
corrcoef_mat = df[['A', 'Z', 'bin_system_type', 'tmerg']].corr(method =␣
 ↪'pearson')
# display(corrcoef_mat)
corrcoef_t_A, corrcoef_t_Z, corrcoef_t_S = corrcoef_mat.loc['tmerg', 'A'],␣
 ↪corrcoef_mat.loc['tmerg', 'Z'], corrcoef_mat.loc['tmerg', 'bin_system_type']
print(f'Pearson coeff. between t and A: {corrcoef_t_A:.3%}')
print(f'Pearson coeff. between t and Z: {corrcoef_t_Z:.3%}')

print(f'Pearson coeff. between t and BST: {corrcoef_t_S:.3%}')
```

```
Pearson coeff. between t and A: 18.656%
Pearson coeff. between t and Z: 5.812%
Pearson coeff. between t and BST: 14.575%
```

We observe values closer to 0 than 1. This makes sense, because we expect the true relation between variables to be far from linear; and yet all r coefficients are between 5% and 20%, suggesting that a linear approximation may not be so far fetched (we will indeed make use of this observation later).

In any case the fact that these scores aren't too close to 0 means that even when assuming a simplistic linear relation our dataset seems to suggest a non-negligible dependence between these variables.

We also notice that the r coefficients associated with A and BST are more than double the one associated to Z. This makes sense: compared to Z we intuitively expect A and BST to more directly/more strongly influence orbital properties, and therefore how much time is needed to decrease orbital separation below the critical value.

5.2 Mutual information

5.2.1 Theoretical definition

As said above the r coefficient quickly becomes irrelevant with nonlinear functional dependencies (e.g. a quadratic *deterministic* relation between variables will yield $r = 0$); therefore we turn our attention to *mutual information*, a quantity that allows us to construct a nonlinear equivalent of r - at least in the sense of information theory.

To explain how to do so we first need to setup a Bayesian framework for our dataset, so that information theory may enter the problem more easily. In particular we model t, A, Z and BST as random variables in the Bayesian sense, i.e. in the sense that their values have an uncertainty associated to them. For example even if we restrict the problem to a specific A value the dataset will still contain many different t values, reflecting this Bayesian uncertainty.

This intrinsic uncertainty is quantified by information theory's *entropy*, defined as

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log p(x) \quad \text{or} \quad H(X) = -\int_{x\in\mathcal{X}} p(x)\log p(x)\ \mathrm{d}x \tag{1}$$

depending on whether X is a discrete or continuous random variable. Notice that this uncertainty essentially represents the information content of the random variable: intuitively specifying the value of a highly uncertain variable tells us much more than specifying the value of a certain variable (e.g. a constant variable). Also notice that H is measured either in *bits* or *nats* depending on whether we use the base 2 or base e logarithm (the two most common conventions).

If two variables X and Y are not independent they must share some information: telling someone the value of X will reduce their uncertainty over Y, as these variables are related. *Mutual information* measures exactly how much information is shared between X and Y, i.e. how many bits/nats less we have in Y's uncertainty if we specify the value of X. Mutual information is defined as

$$I(X;Y) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right) \quad \text{or} \quad I(X;Y) = \int_{x\in\mathcal{X}}\int_{y\in\mathcal{Y}} p(x,y)\log\left(\frac{p(x,y)}{p(x)p(y)}\right)\ \mathrm{d}x\ \mathrm{d}y \tag{2}$$

depending on whether the random variables are discrete or continuous. One can easily prove that this quantity has the required properties; for example if X and Y are independent then $p(X,Y) = p(X)p(Y)$, which makes the logarithm identically 0 and therefore makes $I = 0$.

Assume we compute the mutual information between two random variables X and Y, and that we specify that X actually equals to some value; then we know that the new Y entropy is

$$H'(Y) = H(Y) - I(X;Y) \tag{3}$$

because by definition $I(X;Y)$ is the information content shared between the two. This means that we were able to reduce the uncertainty over Y by $\Delta H(Y) = I(X;Y)$, i.e. that we *fractionally* reduced the uncertainty by

$$c_Y = \frac{\Delta H(Y)}{H(Y)} = \frac{I(X;Y)}{H(Y)} \tag{4}$$

Notice that this quantity is dimensionless, equal to 0 for independent variables and closer and closer to 1 the stronger the dependence between X and Y is. Indeed for independent variables $I = 0$, as shown above, which implies $c = 0$; instead in the limit where one variable deterministically determines the other (e.g. $Y = X^2$ exactly) the information content of X is completely shared with

Y's as they are the same variable, which implies $c = 1$ (one can trivially verify that in this limit the expression of I becomes the same as H's).

An interesting property of c is that the above behaviour (i.e. being in $[0, 1]$ depending on how independent/dependent the variables are) is that *this does not depend on the specific dependence between X and Y*, only on whether they are independent or not, i.e. on whether they share information or not. This is the reason why it makes sense to use c as a nonlinear replacement for r.

5.2.2 Practical computation

In our problem we can model the strength of the dependence between t and A, Z, BST using the above framework - namely, we ask the question: *"how much uncertainty over t's value is reduced by specifying $A/Z/BST$'s value?"* For example if the relation between t and A is strong specifying A should greatly reduce the range of possible t values compatible with that A, which would make $c(t; A)$ close to 1. Of course in practice we expect all c's to be closer to 0 than 1: the complexity of the problem makes it unlikely that a single variable is able to alone almost completely determine the value of t - if this was the case computing t would be a trivial task. The main difference with the previous computation is that now the normalized quantity has an unambiguous meaning, as it measures the strength of the functional dependence *irrespective of the functional dependence itself.*

Now we only have to actually compute the c coefficients. In order to do so we remark a few properties of this specific problem: - t can be considered a continuous random variable, as the dataset has a huge range of possible real t values; instead A and Z should be considered discrete random variables, as only a few fixed values are available. Due to this the mutual information will be in a "mixed form", with a sum and an integral. - In order to compute the 1D marginals of the discrete variables ($p(A)$, $p(Z)$, $p(BST)$) we can use a simple maximum likelihood estimator, i.e. compute each probability as the ratio of the number of rows with that value and the total number of dataset points. - Estimating the 1D marginal of t is less trivial. A possible way is to approximate it using the midpoints of each bin in the above histograms (but then the result will depend on the number of bins used), while another is to use a *kernel density estimation* (kde) algorithm (but then the result will depend on the chosen smoothing kernel properties); this unavoidable ambiguity means the final estimate of c_t may be slightly inaccurate, but for this problem in particular it doesn't seem to change much. Therefore in this section we will use a simple gaussian kde, while later on we will resort to computing the mid-bin histogram points. - A possible way to compute the joint pdf in the mutual information is to rewrite it using the conditional pdf; for example with A one obtains $p(t, A) = p(t|A)p(A)$. This makes sense because we already know how to compute the discrete 1D marginals ($p(A)$ in this example), and obtaining the conditional is also trivial: for any fixed A value $p(t|A)$ is a 1D distribution, and can be estimated like $p(t)$ - we only need to first restrict the dataset to the rows with that specific A value. This of course works analogously for the other variables.

Combining all the above points we obtain the following expression for the mutual information:

$$I(t; X) = \sum_{x \in \mathcal{X}} \int_{t \in \mathcal{T}} p(t, x) \log \left(\frac{p(t, x)}{p(t)p(x)} \right) \, \mathrm{d}t = \sum_{x \in \mathcal{X}} \int_{t \in \mathcal{T}} p(t|x) \, p(x) \log \left(\frac{p(t|x)}{p(t)} \right) \, \mathrm{d}t = \sum_{x \in \mathcal{X}} p(x) \int_{t \in \mathcal{T}} p(t|x) \, \log \left(\frac{}{\sum_{} \right. \tag{5}$$

where X can be any of the variables A, Z or BST. Actually X need not be a single number, only a variable taking values from a set $\mathcal{X}$; this set can also be the cartesian product of our variables, like for example the set of all (A, Z) pairs.

Once the mutual information has been computed the ratio of it with the entropy

$$H(t) = -\int_{t\in\mathcal{T}} p(t)\log p(t)\ \mathrm{d}t \tag{6}$$

will yield the desired coefficient c_t.

Mutual information between t and either A or Z

[21]:
```
# in the following we adopt the convention of using log2
def mutual_info_integral(data, A = None, Z = None):
    data_values = data.A if A is not None else data.Z
    value = A if A is not None else Z
    values = np.unique(data_values)
    prob_values = data.value_counts('A' if A is not None else 'Z', normalize =␣
 ↪True, sort = False).to_dict()
    pdf_dict = {v:gaussian_kde(data.loc[data_values == v, 'tmerg']) for v in␣
 ↪values}
    f = lambda t: pdf_dict[value].pdf(t) * (np.log2(pdf_dict[value].pdf(t)) -␣
 ↪np.log2(sum((prob_values[(v,)] * pdf_dict[v].pdf(t) for v in values))))
    integral = quad(f, data.tmerg.min(), data.tmerg.max())
    return integral[0] * prob_values[(value,)]

def mutual_info(data, var):
    values = np.unique(data.A if var == 'A' else data.Z)
    if var == 'A':
        return sum([mutual_info_integral(data = data, A = v) for v in values])
    else:
        return sum([mutual_info_integral(data = data, Z = v) for v in values])
```

[22]:
```
I_A = mutual_info(df, 'A')
print('mutual information between t and A:', I_A)
```

```
mutual information between t and A: 0.08422411758081408
```

[23]:
```
I_Z = mutual_info(df, 'Z')
print('mutual information between t and Z:', I_Z)
```

```
mutual information between t and Z: 0.005696618590593822
```

[24]:
```
g = gaussian_kde(df.tmerg)
fg = lambda t: g.pdf(t) * np.log2(g.pdf(t))
H = -quad(fg, df.tmerg.min(), df.tmerg.max())[0]
print("H(t) entropy:", H)
```

```
H(t) entropy: 11.301538666633007
```

[25]:
```
print(f'A: percent entropy reduction over t = {I_A/H:%}, new/old entropy = {1 -␣
 ↪I_A/H:%}')
```

```
print(f'Z: percent entropy reduction over t = {I_Z/H:%}, new/old entropy = {1 -␣
 ↪I_Z/H:%}')
```

```
A: percent entropy reduction over t = 0.745245%, new/old entropy = 99.254755%
Z: percent entropy reduction over t = 0.050406%, new/old entropy = 99.949594%
```

We notice that both I/H ratios are almost zero. This is reasonable: the task at hand is predicting a scalar quantity that varies over a huge range, even after specifying the value of A/Z; therefore simply stating e.g. that $A = 1$ doesn't tell much about t, which may still be equal to any value in a very large interval.

The fact that these coefficients aren't exactly zero means that t isn't completely independent from A and Z, but that these correlations are so weak that they aren't necessarily useful; one may therefore conclude that t is independent w.r.t. A/Z to excellent approximation - consistent with the fact that there are many other parameters needed to determine t, and which more directly/strongly influence the value of t itself.

Finally notice that the mutual information associated to A is more than double compared to Z's, consistent with the result obtained when computing the Pearson coefficients. This suggests once again that A plays a more important role in determining t than Z, which makes sense since A can more directly influence the orbital properties of the binary system.

Mutual information between t and both A and Z Let us now compute how much the uncertainty over t is reduced if we specify both A and Z's values.

[26]:
```
prob_values = {}
pdf_dict = {}

for A, Z in product(A_vec, Z_vec): # basta un nested for loop
    prob_values[(A, Z)] = df.loc[(df.A == A) & (df.Z == Z)].shape[0] / df.
 ↪shape[0]
    pdf_dict[(A, Z)] = gaussian_kde(df.loc[(df.A == A) & (df.Z == Z), 'tmerg'])

def f2(t, a, z):
    return pdf_dict[(a, z)].pdf(t) * (np.log2(pdf_dict[(a, z)].pdf(t)) - np.
 ↪log2(sum((prob_values[(A, Z)] * pdf_dict[(A, Z)].pdf(t) for (A, Z) in␣
 ↪product(A_vec, Z_vec)))))

I_AZ = sum([quad(f2, df.tmerg.min(), df.tmerg.max(), args = tup)[0] *␣
 ↪prob_values[tup] for tup in product(A_vec, Z_vec)])
print('mutual information between t and (A, Z):', I_AZ)
```

```
mutual information between t and (A, Z): 0.11644805354894672
```

[27]:
```
print(f'(A, Z): percent entropy reduction over t = {I_AZ/H:%}, new/old entropy␣
 ↪= {1 - I_AZ/H:%}')
```

```
(A, Z): percent entropy reduction over t = 1.030373%, new/old entropy =
98.969627%
```

We notice that the mutual information has increased to the point that c is larger than 1%. This makes sense; intuitively the c coefficient should increase as we're specifying more information and therefore removing even more of t's indetermination.

Even now, though, c is much closer to 0 than 1, which means that the correlation is still weak and that to a good approximation t can be considered independent from A *and* Z. This is consistent with the fact that specifying both A and Z cannot significantly decrease t's variability range.

Mutual information between t and BST Let us now compute how much the uncertainty over t is reduced if we specify BST, the *binary system type*.

[28]:
```
def mutual_info_integral_bst(data, bst):
    data_values = data.bin_system_type
    value = bst
    values = np.unique(data_values)
    prob_values = data.value_counts('bin_system_type', normalize = True, sort =␣
 ↪False).to_dict()
    pdf_dict = {v:gaussian_kde(data.loc[data_values == v, 'tmerg']) for v in␣
 ↪values}
    f = lambda t: pdf_dict[value].pdf(t) * (np.log2(pdf_dict[value].pdf(t)) -␣
 ↪np.log2(sum((prob_values[v] * pdf_dict[v].pdf(t) for v in values))))
    integral = quad(f, df.tmerg.min(), df.tmerg.max())
    return integral[0] * prob_values[value]

def mutual_info_bst(data):
    values = np.unique(data.bin_system_type)
    return sum([mutual_info_integral_bst(data = data, bst = v) for v in values])
```

[29]:
```
I_BST = mutual_info_bst(df)
print('mutual information between t and BST:', I_BST)
```

```
mutual information between t and BST: 0.041380805083778824
```

[30]:
```
print(f'BST: percent entropy reduction over t = {I_BST/H:%}, new/old entropy =␣
 ↪{1 - I_BST/H:%}')
```

```
BST: percent entropy reduction over t = 0.366152%, new/old entropy = 99.633848%
```

The value of this c coefficient is in between the A and Z ones, meaning that specifying the binary system type is more useful in determining t than Z is, but not as much as fixing A. This makes sense: we showed before that increasing A tends to on average increase t with a trend slightly stronger than introducing more neutron stars in the system did. Here we obtain the same result, but in a more quantitative fashion.

What about a "super" mixed mutual information? Similarly to what we did above when combining A and Z it would be interesting to add BST to the mix, i.e. compute the mutual information between t and (A, Z, BST). In practice this is not actually feasible; to see why let us see how many rows have $A = 0.01$, $Z = 0.002$ and $BST = 1$.

[31]:
```
df.loc[(df.A == 0.01) & (df.Z == 0.002) & (df.bin_system_type == 1)]
```

[31]:

	ID	A	Z	min1	min2	tform	sepform	eccform	k1form	\
600	224698	0.01	0.002	5.8928	5.3416	86.3515	12.028	0.96724	13	

	m1form	k2form	m2form	tmerg	k1	m1	k2	m2	bin_system_type
600	1.2608	13	1.2608	191.4093	15	1.2591	13	2.5216	1

As we can see there is only one data sample with the required values. This means that if we try to estimate its probability as the ratio between favorable and total number of cases this quantity will essentially be zero, which when used as the argument of the logarithm in the mutual information formula will make the integral diverge. Similarly if we tried to use a kde algorithm it would give an error, because a single point isn't enough to estimate a probability.

Finally notice that the mutual information integrals are relatively complex, so adding more and more variables to the mix means multiplicatively increasing the number of integrals to perform - making the computation less and less trivial.

For all these reasons we declare computing a "super" c coefficient unfeasible with the available dataset; a more comprehensive one would probably make this task easier to accomplish.

6 Deviations of $\mathrm{d}N/\mathrm{d}t$ from $1/t$

Above we plotted several variations of $\mathrm{d}N/\mathrm{d}t$ and were able to notice that (especially for small t values) the empirical distribution somewhat departs from the heuristical $1/t$ assumption.

Now we would like to quantitatively assign a score to the strength and/or significance of these deviations. To do this we can define a couple of statistics that, when computed on the dataset, will yield a normalized value that we can easily interpret. Notice that depending on how we define "deviations from $1/t$" (or lack thereof) we should use different metrics; the following three have been adopted here. - If we want to know how well $\mathrm{d}N/\mathrm{d}t$ can be fitted to $1/t$ a possible approach is to use standard fit statistics, such as the *residual sum of squares*; we can also use a change of coordinates to be able to use Pearson's r coefficient. - If we want to know how similar the 2 distributions are we could for example measure the difference in their information content; information theory makes it easy to compute such "distance", via the *Kullback-Leibler divergence*, and more specifically via the *Jensen-Shannon divergence*. - If instead we want to study whether the empirical and theoretical dataset can be considered as coming from the same distribution (and assign a p-value to this) we can use a frequentist hypothesis test, such as the *Kolmogorov-Smirnov* one.

6.1 Linear fit of $\mathrm{d}N/\mathrm{d}t$

6.1.1 1-parameter linear fit

Let us revisit the property that *in log-log space power laws become straight lines*. Our target distribution is proportional to $1/t$, i.e.

$$p(t) = \frac{k}{t} \tag{7}$$

Let us take the logarithm of both sides:

$$\log p(t) = \log k - \log t = -\log t + \log k \tag{8}$$

Let us set $y = \log p(t)$, $x = \log t$, $q_f = \log k$. Then the equation becomes:

$$y = -x + q_f \tag{9}$$

This means that with this simple change of variables we can fit our distribution to a line with fixed -1 slope and a variable q_f intercept, whose exponential is the constant factor in the original power law. We remind ourselves that the slope is fixed with using the f subscript in q_f.

6.1.2 2-parameter linear fit

This approach also allows us to investigate more general power law. Suppose that we relax the $1/t$ assumption, instead assuming a more general power law:

$$p(t) = kt^{\gamma} \tag{10}$$

with γ not necessarily equal to -1. Performing the same steps as above we obtain:

$$\log p(t) = \gamma \log t + \log k \implies y = \gamma x + q \tag{11}$$

In this case the power law exponent has become the line slope, while the *logarithm* of the multiplicative constant becomes the line intercept once again.

Below we also use this second approach to the linear fitting of $\mathrm{d}N/\mathrm{d}t$; even though we're straying from the simpler target distribution (the only one with a well-defined physical origin) it is interesting to see whether we can improve the fit quality by modifying the exponent in the original law (and if so how much). This is actually nothing new in physics; a simple modification to a heuristically-derived law can be useful, similarly to how unphysical $A > 1$ can still matter.

6.1.3 Details about fit computation

In this section we avoid any smoothing via e.g. kde, instead relying directly on the mid-bin points from the histogram (this makes the procedure bin-dependent, but some ambiguity will always remain with a finite number of samples).

In order to perform the 2-parameter linear fit we can use standard linear least squares, using e.g. `scipy.stats.linregress`; when it comes to the 1-parameter fit we can trivially compute the least squares estimate of the parameter as follows.

Suppose we want to fit our dataset to the fixed-slope line $y = -x + q_f$ using least squares; this means we need to minimize the (squared) $L2$ norm of the vector difference between predicted and true values, i.e.

$$L(q_f) = ||y - (-x + q_f)||_2^2 = \sum_{i=1}^{N} (y_i + x_i - q_f)^2 \tag{12}$$

To minimize we can set $L(q_f)$'s derivative to zero:

$$\frac{\mathrm{d}L}{\mathrm{d}q_f} = -2 \sum_{i=1}^{N} (y_i + x_i - q_f) = 0 \tag{13}$$

The estimator $\hat{q}_f$ that satisfies the above equation also satisfies

$$\sum_{i=1}^{N} (y_i + x_i - \hat{q}_f) = 0 \implies N\hat{q}_f = \sum_{i=1}^{N} (y_i + x_i) \tag{14}$$

from which it trivially follows that

$$\hat{q}_f = \frac{1}{N}\sum_{i=1}^{N}(y_i + x_i) = \langle y + x \rangle \tag{15}$$

i.e. that q_f can be estimated as the arithmetic mean of $y + x$.

[32]:
```
def compute_log_space_hist_bins(n_bins = 50, min_value = 1, max_value = 14e3,␣
↪x_base = 10):
    MIN_VALUE = min_value
    MAX_VALUE = max_value

    XBASE = x_base

    start_power = int(np.floor(np.log(MIN_VALUE) / np.log(XBASE)))
    end_power = int(np.ceil(np.log(MAX_VALUE) / np.log(XBASE)))
    num_bins = n_bins

    hist_bins = np.logspace(start_power, end_power, num_bins, base = XBASE)
    return hist_bins

def compute_histogram_values(vec, hist_bins):
    hist_values, hist_edges = np.histogram(vec, bins = hist_bins, density =␣
↪True)
    return hist_values, hist_edges

def compute_mid_points_hist_edges(hist_edges):
    mid_points_hist_edges = hist_edges[:-1] + (hist_edges[1:] - hist_edges[:
↪-1]) / 2 # np.convolve(hist_edges, [0.5, 0.5], 'valid')
    return mid_points_hist_edges

def loglog_transform(time_vec, eps = 1e-10, n_bins = 50, min_value = 1,␣
↪max_value = 14e3, x_base = 10):
    hist_bins = compute_log_space_hist_bins(n_bins = n_bins, min_value =␣
↪min_value, max_value = max_value, x_base = x_base)
    hist_values, hist_edges = compute_histogram_values(time_vec, hist_bins)
    mid_points_hist_edges = compute_mid_points_hist_edges(hist_edges)

    points_to_keep_mask = hist_values > eps # remove too small values to avoid␣
↪fitting errors
    mid_points_hist_edges = mid_points_hist_edges[points_to_keep_mask]
    hist_values = hist_values[points_to_keep_mask] # linregress propagates␣
↪nan's -> remove them

    x, y = np.log10(mid_points_hist_edges), np.log10(hist_values) # log10␣
↪convention
    return x, y
```

[33]:
```
def linear_fit(time_vec, eps = 1e-10, n_bins = 50, min_value = 1, max_value =␣
↪14e3, x_base = 10):
    x, y = loglog_transform(time_vec, eps, n_bins, min_value, max_value, x_base)
    l = linregress(x, y)
    rss_normalized = np.mean((y - (l.slope * x + l.intercept)) ** 2)
    return l, rss_normalized
```

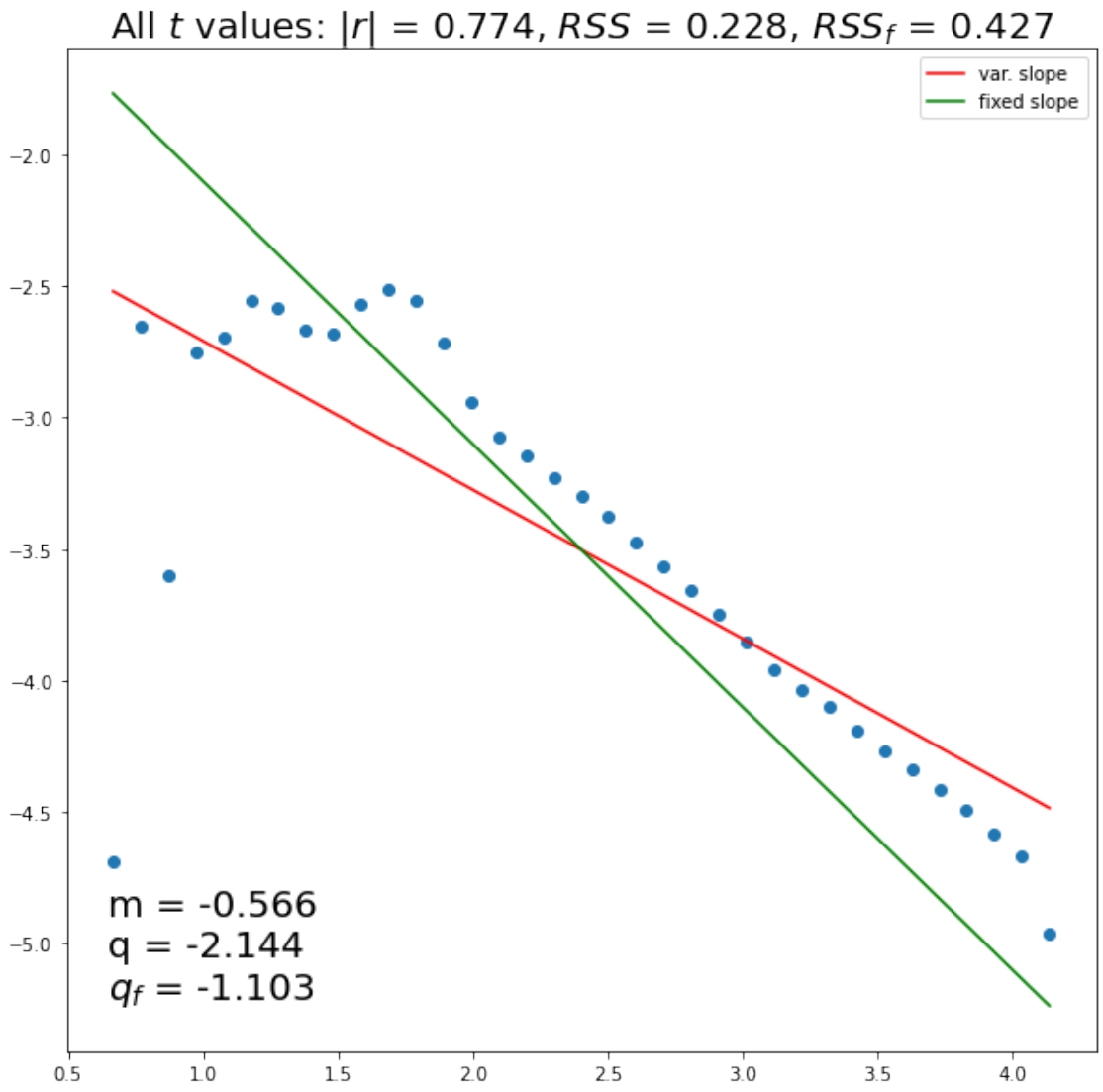
[34]:
```
def linear_fit_fixed_slope(time_vec, eps = 1e-10, n_bins = 50, min_value = 1,␣
↪max_value = 14e3, x_base = 10):
    x, y = loglog_transform(time_vec, eps, n_bins, min_value, max_value, x_base)
    q = np.mean(x + y)
    rss_normalized = np.mean((y - (-x + q)) ** 2)
    return q, rss_normalized
```

6.1.4 r and RSS

Thanks to log-log space our target distribution has become a straight line; this means that we can use Pearson's coefficient to obtain an easy to interpret statistic measuring how much the empirical distribution deviates from the heuristic one. Notice that due to the negative slopes all r values will be negative, so for simplicity we only consider the absolute values.

Another possible metric is the *residual sum of squares*, i.e. the $L2$ loss between the predicted and true values divided by the number of samples. This metric is not normalized, so it's not trivial to say e.g. when it is "close enough" to zero; for this reason we prefer to use r to estimate the strength of the deviations between distributions. Still the RSS is useful to estimate how much the fit improves if we allow a variable slope in log-log space i.e. a more general power law in real space.

[35]:
```
xx, yy = loglog_transform(df.tmerg)
l, rss = linear_fit(df.tmerg)
q_f, rss_f = linear_fit_fixed_slope(df.tmerg)

fig, ax = plt.subplots(1, 1, figsize = (10, 10))

ax.scatter(xx, yy)
ax.plot(xx, l.slope * xx + l.intercept, label = 'var. slope', c = 'red')

ax.plot(xx, - xx + q_f, label = 'fixed slope', c = 'green')
ax.set_title(f'All $t$ values: $|r|$ = {-l.rvalue:.3f}, $RSS$ = {rss:.3f},␣
↪$RSS_f$ = {rss_f:.3f}', size = 20)
ax.legend(loc = 1)

xt, yt = ax.get_xlim()[0] + 0.15, ax.get_ylim()[0] + 0.2
ax.text(xt, yt, f'm = {l.slope:.3f}\nq = {l.intercept:.3f}\n$q_f$ = {q_f:.3f}',␣
↪size = 20);
```



When including *all* dataset points we notice a good agreement with the target distribution for the not small t values; we observe this both visually and by noticing that $|r|$ is relatively high, larger than 0.75. The RSS is also relatively low.

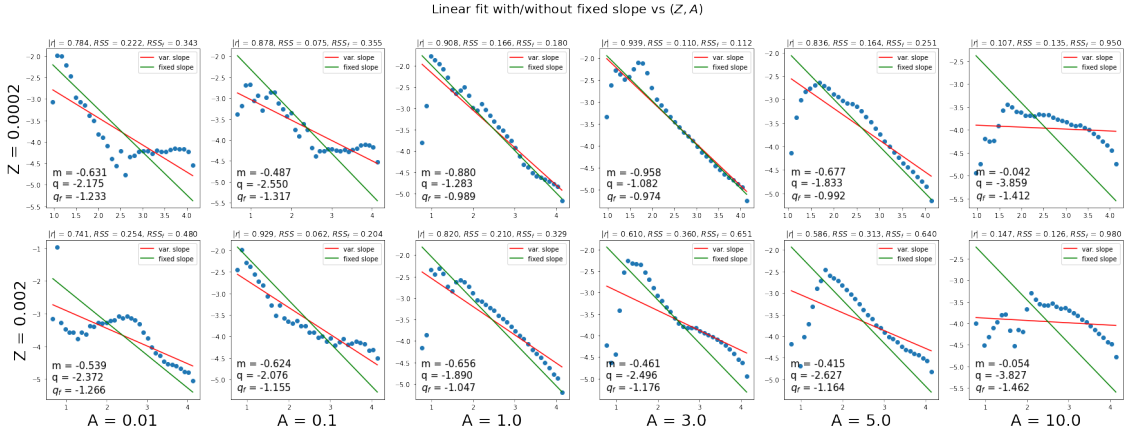
If we allow the line slope to vary we notice that it tilts to the left, clearly to accomodate for the values at small t; by doing so the RSS approximately halves. The red line is closer to the outliers (thus significantly reducing the RSS), but for this very reason no longer aligns nicely with the large t values - breaking away from the heuristical assumption-based distribution.

We also remark that according to this dataset a better quality power law fit can be obtained by approximately halving the -1 exponent.

[36]:
```
lin_fit_dict = {}
lin_fit_fixed_slope_dict = {}
```

```
for A, Z in product(A_vec, Z_vec):
    lin_fit_dict[(A, Z)] = linear_fit(df.loc[(df.A == A) & (df.Z == Z),␣
 ↪'tmerg'])
    lin_fit_fixed_slope_dict[(A, Z)] = linear_fit_fixed_slope(df.loc[(df.A ==␣
 ↪A) & (df.Z == Z), 'tmerg'])
```

[37]:
```
fig, ax = plt.subplots(len(Z_vec), len(A_vec), figsize = (30, 10))

for j, Z in enumerate(Z_vec):
    for i, A in enumerate(A_vec):
        xx, yy = loglog_transform(df.loc[(df.A == A) & (df.Z == Z), 'tmerg'])
        l, rss = lin_fit_dict[(A, Z)]

        ax[j, i].scatter(xx, yy)
        ax[j, i].plot(xx, l.slope * xx + l.intercept, label = 'var. slope', c =␣
 ↪'red')

        q_f, rss_f = lin_fit_fixed_slope_dict[(A, Z)]
        ax[j, i].plot(xx, - xx + q_f, label = 'fixed slope', c = 'green')
        ax[j, i].set_title(f'$|r|$ = {-l.rvalue:.3f}, $RSS$ = {rss:.3f},␣
 ↪$RSS_f$ = {rss_f:.3f}')
        ax[j, i].legend(loc = 1)

        xt, yt = ax[j, i].get_xlim()[0] + 0.15, ax[j, i].get_ylim()[0] + 0.2
        ax[j, i].text(xt, yt, f'm = {l.slope:.3f}\nq = {l.intercept:.3f}\n$q_f$␣
 ↪= {q_f:.3f}', size = 15)

        # ax[j, i].set_xlabel(f'A = {A}', size = 25)
        ax[1, i].set_xlabel(f'A = {A}', size = 25)
    ax[j, 0].set_ylabel(f'Z = {Z}', size = 25)

fig.suptitle('Linear fit with/without fixed slope vs $(Z, A)$', size = 20)
ax;
```



Now that we have some quantitative metrics we can revisit the qualitative discussion of the (ir)regularity of the distributions. For example we noted above that the "intermediate" A values are the ones that align better to the target distribution; here we can appreciate this even more. Notice that the distributions with $A = 1, 3, 5$ and $Z = 0.0002$ align almost perfectly with the fixed slope line (both because the large t values are nicely aligned and because the small t outliers don't disrupt this alignment much); this is reflected in their high $|r|$ scores (all larger than $0.8 - 0.9$) and in the fact that varying the slope cannot improve much the fit quality (the lines are almost identical, and so are the tow RSS scores). This alignment is a bit perturbed at the higher metallicity value, but still remains significant.

The low A distributions clearly have a less straight shape, but still retain r and RSS scores quite close to the intermediate A distributions because their general shapes are still those of decreasing lines. In particular even though these distributions are more "bumpy" compared to the intermediate A ones their outliers are relatively close to the other points, whereas the intermediate A distributions' outliers are quite far from the others; this compensation is what gives us the almost identical r values. If e.g. the outliers or the small t values should be given more weight one may want to use a weighted least squares scheme; here for simplicity we chose not to, also to not make potentially arbitrary choices.

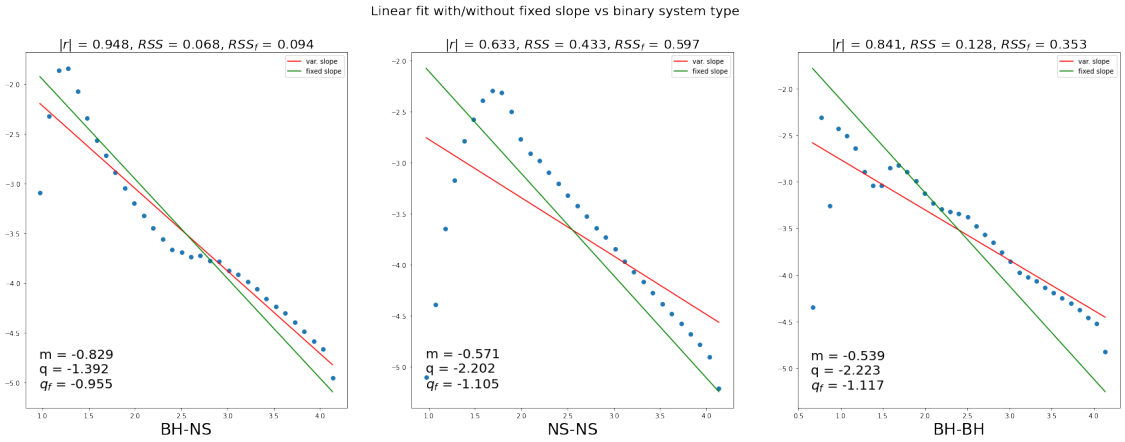
We noticed above that as A increases more "mass" in the distributions moves to the right. This is especially noticeable here in the bottom $Z = 0.002$ row: except for the first A value we have that RSS_f steadily increases, i.e that the distributions become flatter and flatter and therefore further from the -1 slope straight line. Indeed the $A = 10$ distributions are so flat that since they have low values at very low/high t they become almost parabolic in shape - hence their almost $r = 0$ score. This behaviour is less noticeable at the lowest metallicity value (first row) because RSS_f decreases for a bit before starting to increase; evidently the lower metallicity acts as some sort of "regularizer", keeping the distribution closer to the target for a wider range of A values.

A final, general remark: we notice that in almost all graphs by switching to a variable slope line fit the RSS scores improves by at most a factor of 2, which isn't much considering all obtained RSS values are smaller than 1. This means that even though a more customized power law can increase the fit quality the limiting factor is the functional form itself; in order to *significantly* decrease the RSS scores we would need to use more complex non-monotonic functions, capable of capturing the local minima and maxima. At this point of course we are straying away from physical arguments and simply trying to fit a dataset, and for this reason this approach has not been pursued in this work.

[38]:
```
lin_fit_bst_dict = {}
lin_fit_fixed_slope_bst_dict = {}

for bst in [0, 1, 2]:
    lin_fit_bst_dict[bst] = linear_fit(df.loc[df.bin_system_type == bst,␣
 ↪'tmerg'])
    lin_fit_fixed_slope_bst_dict[bst] = linear_fit_fixed_slope(df.loc[df.
 ↪bin_system_type == bst, 'tmerg'])
```

```
bst_types = {0:'BH-NS', 1:'NS-NS', 2:'BH-BH'}
```

[39]:
```
fig, ax = plt.subplots(1, 3, figsize = (30, 10))

for i, bst in enumerate([0, 1, 2]):
    xx, yy = loglog_transform(df.loc[df.bin_system_type == bst, 'tmerg'])
    l, rss = lin_fit_bst_dict[bst]

    ax[i].scatter(xx, yy)
    ax[i].plot(xx, l.slope * xx + l.intercept, label = 'var. slope', c = 'red')

    q_f, rss_f = lin_fit_fixed_slope_bst_dict[bst]
    ax[i].plot(xx, - xx + q_f, label = 'fixed slope', c = 'green')
    ax[i].set_title(f'$|r|$ = {-l.rvalue:.3f}, $RSS$ = {rss:.3f}, $RSS_f$ =␣
 ↪{rss_f:.3f}', size = 20)
    ax[i].legend(loc = 1)

    xt, yt = ax[i].get_xlim()[0] + 0.15, ax[i].get_ylim()[0] + 0.2
    ax[i].text(xt, yt, f'm = {l.slope:.3f}\nq = {l.intercept:.3f}\n$q_f$ = {q_f:
 ↪.3f}', size = 20)

    ax[i].set_xlabel(f'{bst_types[i]}', size = 25)

fig.suptitle('Linear fit with/without fixed slope vs binary system type', size␣
 ↪= 20)
ax;
```



When we group by binary type we notice that the main factor in determining whether $|r|$ is high is how far away the outliers are. All distributions are in good agreement with the fixed slope line; visually the best alignment at large t is the one in the NS-NS type, and yet its $|r|$ is the lowest because it has the largest number of far away outliers. Once again this may suggest that a more thorough treatment, that deals with outliers and/or small t values differently, may produce more

satisfying metrics, i.e. metrics that better align with intuition.

Finally we remark that similarly to the "complete dataset" plot mixing together samples with different (A, Z) values seems to have a regularizing effect, smoothing the shape of the distribution and in part making it closer to the target line.

6.2 Jensen-Shannon divergence

6.2.1 Mathematical definition

We now turn to information theory to assign a score to the similarity between the empirical and theoretical distribution, depending on how much information content is shared between them.

A well known quantity in information theory is the *Kullback-Leibler divergence*, a quantity that represents a pseudo-distance between distributions: each distribution has 0 "distance" from itself, while different distributions will have a larger and larger KL-divergence as the difference between their information content grows. The mathematical definition in the case of a discrete distribution is:

$$KL(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

Notice that even though in our case we're technically dealing with continuous distributions in practice we use their discretely sampled versions, i.e. we have a vector of probability values corresponding to a certain x vector.

An important fact about the KL divergence is that it's not an actual distance: in particular it is not symmetrical (i.e. $KL(P||Q) \neq KL(Q||P)$), so if we wanted to use it as a similarity score between P and Q there would be an ambiguity regarding the order of the distributions. Another issue is that the KL divergence is not normalized: for different distributions it can grow indefinitely, so we would need to set an arbitrary threshold to decide when two distributions are "close enough" or not.

Fortunately we can fix both issues using the *Jensen-Shannon divergence*, a quantity computed by symmetryizing and normalizing the KL-divergence. One defines:

$$JS(P||Q) = \frac{1}{2} \left(KL(P||M) + KL(Q||M) \right), \quad \text{where} \quad M = \frac{1}{2}(P + Q)$$

It is possible to show that the JS divergence is symmetric (i.e. $JS(P||Q) = JS(Q||P)$) and normalized (i.e. $JS(P||Q) \in [0, 1]$ $\forall P, Q$ if we use base 2 logarithms inside KL). This means that the JS divergence can be used as an actual information distance between distributions.

For simplicity we will actually compute $1 - JS$ instead of JS; this way we can use this quantity as a "information similarity score", i.e. 1 represents identical distributions and values close to 0 completely different distributions. This is needed because the JS divergence is zero when we feed it identical distributions, and instead equals 1 if the two distributions refer to independent variables, i.e. share no information (this comes from the fact that the JS divergence is actually the mutual information between the two random variables written above).

To recap: $1-JS$ is the *percent of information shared by the two distributions*: $1-JS = 0\%$ means the two distributions share no information (i.e. they describe independent variables), while $1 - JS = 100\%$ means that the two distributions have the same information and therefore are identical, i.e. they describe the same variables. Values in between represent *some* shared information, and

therefore a certain degree of similarity between the two distributions. For this reason $1 - JS$ is a good similarity index (based on information theory).

6.2.2 Real or log-log space?

In the linear fit section we showed that it was possible to fit the same coefficients in both the real and log-log spaces. When it comes to information content, though, there is no guarantee that a coordinate transformation will preserve e.g. the JS divergence; for this reason the only JS similarity score to be trusted should be the one computed in real space. Still it's interesting to see what happens in log-log space, so even if probably unreliable we will compute the JS divergence in both coordinates.

6.2.3 JS similarity between empirical and theoretical distributions

All data points

[40]:
```
def compute_pdf(time_vec, eps = 1e-10, n_bins = 50, min_value = 1, max_value =␣
↪14e3, x_base = 10):
    hist_bins = compute_log_space_hist_bins(n_bins = n_bins, min_value =␣
↪min_value, max_value = max_value, x_base = x_base)
    hist_values, hist_edges = compute_histogram_values(time_vec, hist_bins)
    mid_points_hist_edges = compute_mid_points_hist_edges(hist_edges)
    points_to_keep_mask = hist_values > eps
    mid_points_hist_edges = mid_points_hist_edges[points_to_keep_mask]
    hist_values = hist_values[points_to_keep_mask]
    x, y = mid_points_hist_edges, hist_values
    return x, y
```

[41]:
```
def KL_div(p_probs, q_probs):
    KL_div = p_probs * np.log2(p_probs / q_probs) # log2 to have JS in [0, 1]␣
↪instead of [0, ln(2)] (ln)
    return np.sum(KL_div)

def JS_Div(p, q):
    # normalize
    p /= p.sum()
    q /= q.sum()
    m = (p + q) / 2
    return (KL_div(p, m) + KL_div(q, m)) / 2
```

[42]:
```
# all data points
xx, yy = loglog_transform(df.tmerg)
l, rss = linear_fit(df.tmerg)
q_f, rss_f = linear_fit_fixed_slope(df.tmerg)

# log-log space is base 10 -> we invert the log-log transform using 10^x
# JS div is instead base 2

x2, y2 = compute_pdf(df.tmerg)
```

```
k, gamma = 10 ** l.intercept, l.slope
k_f = 10 ** q_f

print(f'JS div., all data points in original space, FIXED SLOPE: {JS_Div(y2,␣
 ↪k_f / x2):.3f}, percent JS similarity: {(1 - JS_Div(y2, k_f / x2)):.3%}')
print(f'JS div., all data points in original space, VARIABLE SLOPE: {JS_Div(y2,␣
 ↪k * (x2 ** gamma)):.3f}, percent JS similarity: {(1 - JS_Div(y2, k * (x2 **␣
 ↪gamma))):.3%}')
print('\n')
print(f'JS div., all data points in loglog space, FIXED SLOPE: {JS_Div(yy, -xx␣
 ↪+ q_f):.3f}, percent JS similarity: {(1 - JS_Div(yy, -xx + q_f)):.3%}')
print(f'JS div., all data points in loglog space, VARIABLE SLOPE: {JS_Div(yy, l.
 ↪slope * xx + l.intercept):.3f}, percent JS similarity: {(1 - JS_Div(yy, l.
 ↪slope * xx + l.intercept)):.3%}')
```

```
JS div., all data points in original space, FIXED SLOPE: 0.274, percent JS
similarity: 72.561%
JS div., all data points in original space, VARIABLE SLOPE: 0.140, percent JS
similarity: 85.973%


JS div., all data points in loglog space, FIXED SLOPE: 0.007, percent JS
similarity: 99.278%
JS div., all data points in loglog space, VARIABLE SLOPE: 0.003, percent JS
similarity: 99.651%
```

Using the *complete* original dataset and target distribution (proportional to $1/t$) we find a JS similarity score of about 73%, which means that on average all deviations amount to modifying about a quarter of the information originally contained in the target distribution.

If we use the more general power law we obtain an even higher similarity in the information content between the empirical and theoretical distribution (about 86%); this makes sense, because by adding parameters one can always improve any fit (with an increased overfitting risk), which makes the difference between the two distributions even smaller.

An interesting result is that transforming the coordinates seems to smooth the difference between distributions, at least when it comes to their information content: in log-log space they become almost indistinguishable. As said above coordinate transformations may not preserve the original information content and for this reason should be considered unreliable; indeed by using log-log space one may declare these deviations not significant, which is clearly not the case - at least in the original space.

Group by (A, Z)

[43]:
```
def compute_JS_div_all(time_vec):
    xx, yy = loglog_transform(time_vec)
    l, rss = linear_fit(time_vec)
    q_f, rss_f = linear_fit_fixed_slope(time_vec)
    x2, y2 = compute_pdf(time_vec)
```

```
    k, gamma = 10 ** l.intercept, l.slope
    k_f = 10 ** q_f
    return { # already in "human-readable" form
        'original_fixed':1-JS_Div(y2, k_f / x2),
        'original_variable':1-JS_Div(y2, k * (x2 ** gamma)),
        'loglog_fixed':1-JS_Div(yy, -xx + q_f),
        'loglog_variable':1-JS_Div(yy, l.slope * xx + l.intercept)
    }
```

[89]:
```
# group by A/Z
JS_AZ_dict = {}

for i, (A, Z) in enumerate(product(A_vec, Z_vec)):
    tmp = compute_JS_div_all(df.loc[(df.A == A) & (df.Z == Z), 'tmerg'])
    tmp['A'], tmp['Z'] = A, Z
    JS_AZ_dict[i] = tmp

JS_AZ_df = pd.DataFrame.from_dict(JS_AZ_dict, 'index')[['A', 'Z',␣
 ↪'original_fixed', 'original_variable', 'loglog_fixed', 'loglog_variable']]

rg_cmap = LinearSegmentedColormap.from_list('rg', ['r', 'g'], N = 256)

print('JS similarity (data grouped by (A, Z)):')
JS_AZ_df
# JS_AZ_df.style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap =␣
 ↪rg_cmap,
#                                    subset = ['original_fixed',␣
 ↪'original_variable', 'loglog_fixed', 'loglog_variable'])
```

```
JS similarity (data grouped by (A, Z)):
```

[89]:

	A	Z	original_fixed	original_variable	loglog_fixed	loglog_variable
0	0.01	0.0002	0.862606	0.815145	0.995888	0.996844
1	0.01	0.0020	0.483565	0.440631	0.991896	
2	0.10	0.0002	0.848723	0.930782	0.995198	
3	0.10	0.0020	0.956936	0.924006	0.997340	
4	1.00	0.0002	0.813951	0.832274	0.996448	
5	1.00	0.0020	0.768389	0.856961	0.994261	
6	3.00	0.0002	0.810292	0.819144	0.997758	
7	3.00	0.0020	0.616924	0.753629	0.990062	
8	5.00	0.0002	0.759532	0.856870	0.995814	
9	5.00	0.0020	0.558332	0.774734	0.990270	
10	10.00	0.0002	0.531173	0.923378	0.988582	
11	10.00	0.0020	0.484555	0.904153	0.987996	

```
1           0.995156
2           0.998983
3           0.998996
4           0.996826
5           0.996653
6           0.997830
7           0.994587
8           0.997528
9           0.995519
10          0.998510
11          0.998558
```

If we group by A/Z we notice that once again the log-log transform essentially removes the interesting information content difference (all similarity scores become larger than 0.995), which once again means we need to focus on the original, untransformed dataset.

In original space we notice that the scores actually vary quite a bit: some reach $80-90\%$ information similarity, and are therefore consistent with small deviations; others go as low as $\sim 50\%$, meaning there are (A, Z) pairs that can significantly increase the significance of the deviations.

[91]:
```
print('JS similarity (data grouped by (A, Z)):')
print('mean:')
display(pd.DataFrame(JS_AZ_df.mean(axis = 0).drop(['A', 'Z'])).transpose())
# display(pd.DataFrame(JS_AZ_df.mean(axis = 0).drop(['A', 'Z'])).transpose().
 ↪style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap = rg_cmap,
#                                                                          ␣
 ↪                          subset = ['original_fixed', 'original_variable',␣
 ↪'loglog_fixed', 'loglog_variable']))
print('std:')
display(pd.DataFrame(JS_AZ_df.std(axis = 0).drop(['A', 'Z'])).transpose())
```

```
JS similarity (data grouped by (A, Z)):
mean:
```

	original_fixed	original_variable	loglog_fixed	loglog_variable
0	0.707915	0.819309	0.993459	0.997166

```
std:
```

	original_fixed	original_variable	loglog_fixed	loglog_variable
0	0.16402	0.132846	0.003508	0.001503

If we average the above scores we recover values in line with the ones computed with the whole dataset (which were about 73% and 86% depending on which power law we use). This is an intuitive result.

We now more carefully observe which (A, Z) are the ones that yield a high/low similarity score. We immediately notice that they are the same pairs that resulted in low r values in the linear fit section, as can be seen below:

[93]:
```
JS_AZ_df_vs_r = JS_AZ_df.copy().drop(['loglog_fixed', 'loglog_variable'], axis␣
 ↪= 1)
JS_AZ_df_vs_r['r'] = list(map(lambda x: -x[0].rvalue, lin_fit_dict.values()))
print('JS similarity and r (data grouped by (A, Z)):')
JS_AZ_df_vs_r
# JS_AZ_df_vs_r.style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap␣
 ↪= rg_cmap,
#                                         subset = ['original_fixed',␣
 ↪'original_variable', 'r'])
```

```
JS similarity and r (data grouped by (A, Z)):
```

[93]:

	A	Z	original_fixed	original_variable	r
0	0.01	0.0002	0.862606	0.815145	0.783938
1	0.01	0.0020	0.483565	0.440631	0.740661
2	0.10	0.0002	0.848723	0.930782	0.877938
3	0.10	0.0020	0.956936	0.924006	0.928538
4	1.00	0.0002	0.813951	0.832274	0.907611
5	1.00	0.0020	0.768389	0.856961	0.819984
6	3.00	0.0002	0.810292	0.819144	0.938504
7	3.00	0.0020	0.616924	0.753629	0.610048
8	5.00	0.0002	0.759532	0.856870	0.836325
9	5.00	0.0020	0.558332	0.774734	0.586457
10	10.00	0.0002	0.531173	0.923378	0.106833
11	10.00	0.0020	0.484555	0.904153	0.147238

This essentially means that *the JS similarity scores are consistent with the r scores computed above*. Indeed there is a good correlation between them and r, at least when using the simpler power law:

[94]:
```
print('normalized correlation between JS and r (data grouped by (A, Z)):')
JS_AZ_df_vs_r[['original_fixed', 'original_variable', 'r']].corr('pearson')
# JS_AZ_df_vs_r[['original_fixed', 'original_variable', 'r']].corr('pearson').
 ↪style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap = rg_cmap,
#                                                                          ␣
 ↪                         subset = ['original_fixed', 'original_variable',␣
 ↪'r'])
```

```
normalized correlation between JS and r (data grouped by (A, Z)):
```

[94]:

	original_fixed	original_variable	r
original_fixed	1.000000	0.446229	0.773915
original_variable	0.446229	1.000000	-0.136999
r	0.773915	-0.136999	1.000000

The correlation with the 2-parameter power law is much weaker due to a few outliers, like the $A = 10$ rows:

[95]:
```
print('normalized correlation between JS and r (data grouped by (A, Z), A != 10.
 ↪0):')
JS_AZ_df_vs_r.loc[JS_AZ_df.A != 10.0, ['original_fixed', 'original_variable',␣
 ↪'r']].corr('pearson')
# JS_AZ_df_vs_r.loc[JS_AZ_df.A != 10.0, ['original_fixed', 'original_variable',␣
 ↪'r']].corr('pearson').style.background_gradient(axis = None, vmin = 0, vmax␣
 ↪= 1, cmap = rg_cmap,
#                                                                              ␣
 ↪                                                 subset = ['original_fixed',␣
 ↪'original_variable', 'r'])
```

```
normalized correlation between JS and r (data grouped by (A, Z), A != 10.0):
```

[95]:

	original_fixed	original_variable	r
original_fixed	1.000000	0.819973	0.781059
original_variable	0.819973	1.000000	0.454392
r	0.781059	0.454392	1.000000

Still even without these outliers the correlation remains weaker, because with the extra fitting parameter we artificially increase similarity scores even when the original agreement wasn't as good. This seems to suggest that the more realistic JS similarity scores should be computed using the original power law and coordinates.

To recap: the JS divergence can be used to compute a satisfying similarity score based on information content, that works according to intuition: the same (A, Z) pairs that yielded good/bad quality fits (according to graphical plots and r values) yield similarly good/bad JS similarity scores.

Group by *BST*

[97]:
```
JS_BST_dict = {}

for i, bst in enumerate([0, 1, 2]):
    tmp = compute_JS_div_all(df.loc[df.bin_system_type == bst, 'tmerg'])
    tmp['BST'] = bst_types[bst]
    JS_BST_dict[i] = tmp

JS_BST_df = pd.DataFrame.from_dict(JS_BST_dict, 'index')[['BST',␣
 ↪'original_fixed', 'original_variable', 'loglog_fixed', 'loglog_variable']]
print('JS similarity (data grouped by BST):')
JS_BST_df
# JS_BST_df.style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap =␣
 ↪rg_cmap,
#                                     subset = ['original_fixed',␣
 ↪'original_variable', 'loglog_fixed', 'loglog_variable'])
```

```
JS similarity (data grouped by BST):
```

[97]:

	BST	original_fixed	original_variable	loglog_fixed	loglog_variable
0	BH-NS	0.868190	0.871954	0.998296	0.998627
1	NS-NS	0.621735	0.751570	0.991055	0.993844
2	BH-BH	0.813753	0.904055	0.994144	0.998019

[99]:
```
print('JS similarity (data grouped by BST):')
print('mean:')
display(pd.DataFrame(JS_BST_df.drop(['BST'], axis = 1).mean(axis = 0)).
 ↪transpose())
# display(pd.DataFrame(JS_BST_df.drop(['BST'], axis = 1).mean(axis = 0)).
 ↪transpose().style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap␣
 ↪= rg_cmap, subset = ['original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable']))
print('std:')
display(pd.DataFrame(JS_BST_df.drop(['BST'], axis = 1).std(axis = 0)).
 ↪transpose())
```

```
JS similarity (data grouped by BST):
mean:
```

	original_fixed	original_variable	loglog_fixed	loglog_variable
0	0.767893	0.842526	0.994498	0.99683

```
std:
```

	original_fixed	original_variable	loglog_fixed	loglog_variable
0	0.12947	0.080389	0.003634	0.002604

[100]:
```
JS_BST_df_vs_r = JS_BST_df.copy().drop(['loglog_fixed', 'loglog_variable'],␣
 ↪axis = 1)
JS_BST_df_vs_r['r'] = list(map(lambda x: -x[0].rvalue, lin_fit_bst_dict.
 ↪values()))
print('JS similarity and r (data grouped by BST):')
JS_BST_df_vs_r
# JS_BST_df_vs_r.style.background_gradient(axis = None, vmin = 0, vmax = 1,␣
 ↪cmap = rg_cmap,
#                                          subset = ['original_fixed',␣
 ↪'original_variable', 'r'])
```

```
JS similarity and r (data grouped by BST):
```

[100]:

	BST	original_fixed	original_variable	r
0	BH-NS	0.868190	0.871954	0.948279
1	NS-NS	0.621735	0.751570	0.632608
2	BH-BH	0.813753	0.904055	0.840816

[101]:
```
print('normalized correlation between JS and r (data grouped by BST):')
JS_BST_df_vs_r[['original_fixed', 'original_variable', 'r']].corr('pearson')
```

```
# JS_BST_df_vs_r[['original_fixed', 'original_variable', 'r']].corr('pearson').
 ↪style.background_gradient(axis = None, vmin = 0, vmax = 1, cmap = rg_cmap,
#                                                                          ␣
 ↪                          subset = ['original_fixed', 'original_variable',␣
 ↪'r'])
```

normalized correlation between JS and r (data grouped by BST):

[101]:

	original_fixed	original_variable	r
original_fixed	1.000000	0.915991	0.991617
original_variable	0.915991	1.000000	0.856472
r	0.991617	0.856472	1.000000

If we group by binary system type we find essentially the same results as above, i.e. that log-log space is unrealiable because it artificially inflates the similarity score, that averaging everything yields about the overall dataset values, that the 2-parameter fit yields an artificially better similarity. Regarding this last point this time the correlation with r seems to be almost perfect, but of course since this is a 3 element dataset it doesn't mean much; still we can say again that the simpler power law better aligns with the r scores, and therefore is probably better to use in practice.

6.2.4 JS similarity between different empirical distributions

Another interesting use of the JS similarity score employed above is comparing different empirical distributions: before we measured the strength of the difference in information content between an empirical and the corresponding theoretical distribution, instead now we compute the similarity between empirical distributions associated e.g. to different A values. This allows us to quickly obtain a numerical value that (as shown above) both in theory and in practice can reasonably compare distributions.

Group by A

[102]:
```
M_js_a = np.zeros([len(A_vec), len(A_vec)])

for i, A1 in enumerate(A_vec):
    for j, A2 in enumerate(A_vec):
        x1, y1 = compute_pdf(time_vec = df.loc[df.A == A1, 'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.A == A2, 'tmerg'])

        t_range = np.linspace(max(x1.min(), x2.min()), min(x1.max(), x2.max()))

        i1 = interp1d(x1, y1)
        i2 = interp1d(x2, y2)

        M_js_a[i, j] = 1 - JS_Div(i1(t_range), i2(t_range))

print('JS similarity between different A values:')
pd.DataFrame(M_js_a, index = A_vec, columns = A_vec)
```

```
# pd.DataFrame(M_js_a, index = A_vec, columns = A_vec).style.
  ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None)
```

JS similarity between different A values:

[102]:

	0.01	0.10	1.00	3.00	5.00	10.00
0.01	1.000000	0.915151	0.255335	0.165625	0.153986	0.157139
0.10	0.915151	1.000000	0.735604	0.730251	0.675553	0.776243
1.00	0.255335	0.735604	1.000000	0.979967	0.987989	0.914818
3.00	0.165625	0.730251	0.979967	1.000000	0.989297	0.969721
5.00	0.153986	0.675553	0.987989	0.989297	1.000000	0.925632
10.00	0.157139	0.776243	0.914818	0.969721	0.925632	1.000000

Group by Z

[103]:
```
M_js_z = np.zeros([len(Z_vec), len(Z_vec)])

for i, Z1 in enumerate(Z_vec):
    for j, Z2 in enumerate(Z_vec):
        x1, y1 = compute_pdf(time_vec = df.loc[df.Z == Z1, 'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.Z == Z2, 'tmerg'])

        t_range = np.linspace(max(x1.min(), x2.min()), min(x1.max(), x2.max()))

        i1 = interp1d(x1, y1)
        i2 = interp1d(x2, y2)

        M_js_z[i, j] = 1 - JS_Div(i1(t_range), i2(t_range))

print('JS similarity between different Z values:')
pd.DataFrame(M_js_z, index = Z_vec, columns = Z_vec)
# pd.DataFrame(M_js_z, index = Z_vec, columns = Z_vec).style.
  ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None).
  ↪to_html()
```

JS similarity between different Z values:

[103]:

	0.0002	0.0020
0.0002	1.000000	0.994301
0.0020	0.994301	1.000000

Group by (A, Z)

[104]:
```
M_js_az = np.zeros([len(A_vec) * len(Z_vec), len(A_vec) * len(Z_vec)])

for i, (A1, Z1) in enumerate(product(A_vec, Z_vec)):
    for j, (A2, Z2) in enumerate(product(A_vec, Z_vec)):
```

```
        x1, y1 = compute_pdf(time_vec = df.loc[(df.A == A1) & (df.Z == Z1),␣
 ↪'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[(df.A == A2) & (df.Z == Z2),␣
 ↪'tmerg'])

        t_range = np.linspace(max(x1.min(), x2.min()), min(x1.max(), x2.max()))

        i1 = interp1d(x1, y1)
        i2 = interp1d(x2, y2)

        M_js_az[i, j] = 1 - JS_Div(i1(t_range), i2(t_range))

print('JS similarity between different (A, Z) values:')
pd.DataFrame(M_js_az, index = list(product(A_vec, Z_vec)), columns =␣
 ↪list(product(A_vec, Z_vec)))
# pd.DataFrame(M_js_az, index = list(product(A_vec, Z_vec)), columns =␣
 ↪list(product(A_vec, Z_vec))).style.background_gradient(cmap = rg_cmap, vmin␣
 ↪= 0, vmax = 1, axis = None).to_html()
```

JS similarity between different (A, Z) values:

[104]:

	(0.01, 0.0002)	(0.01, 0.002)	(0.1, 0.0002)	(0.1, 0.002)	\
(0.01, 0.0002)	1.000000	0.734208	0.970397	0.868086	
(0.01, 0.002)	0.734208	1.000000	0.761302	0.769712	
(0.1, 0.0002)	0.970397	0.761302	1.000000	0.882880	
(0.1, 0.002)	0.868086	0.769712	0.882880	1.000000	
(1.0, 0.0002)	0.584779	0.269193	0.793831	0.696998	
(1.0, 0.002)	0.757345	0.169213	0.765862	0.637813	
(3.0, 0.0002)	0.838512	0.966642	0.844300	0.805340	
(3.0, 0.002)	0.834501	0.157008	0.874092	0.697503	
(5.0, 0.0002)	0.403741	0.975432	0.718777	0.587525	
(5.0, 0.002)	0.789874	0.163830	0.853893	0.683065	
(10.0, 0.0002)	0.836945	0.808808	0.729623	0.578688	
(10.0, 0.002)	0.819107	0.171545	0.873046	0.704109	

	(1.0, 0.0002)	(1.0, 0.002)	(3.0, 0.0002)	(3.0, 0.002)	\
(0.01, 0.0002)	0.584779	0.757345	0.838512	0.834501	
(0.01, 0.002)	0.269193	0.169213	0.966642	0.157008	
(0.1, 0.0002)	0.793831	0.765862	0.844300	0.874092	
(0.1, 0.002)	0.696998	0.637813	0.805340	0.697503	
(1.0, 0.0002)	1.000000	0.976487	0.628257	0.910923	
(1.0, 0.002)	0.976487	1.000000	0.864871	0.954447	
(3.0, 0.0002)	0.628257	0.864871	1.000000	0.870883	
(3.0, 0.002)	0.910923	0.954447	0.870883	1.000000	
(5.0, 0.0002)	0.364185	0.691392	0.675651	0.899173	
(5.0, 0.002)	0.966427	0.979647	0.891912	0.978639	
(10.0, 0.0002)	0.242919	0.545488	0.826507	0.993526	

(10.0, 0.002)	0.915413	0.957243	0.863222	0.998082

	(5.0, 0.0002)	(5.0, 0.002)	(10.0, 0.0002)	(10.0, 0.002)
(0.01, 0.0002)	0.403741	0.789874	0.836945	0.819107
(0.01, 0.002)	0.975432	0.163830	0.808808	0.171545
(0.1, 0.0002)	0.718777	0.853893	0.729623	0.873046
(0.1, 0.002)	0.587525	0.683065	0.578688	0.704109
(1.0, 0.0002)	0.364185	0.966427	0.242919	0.915413
(1.0, 0.002)	0.691392	0.979647	0.545488	0.957243
(3.0, 0.0002)	0.675651	0.891912	0.826507	0.863222
(3.0, 0.002)	0.899173	0.978639	0.993526	0.998082
(5.0, 0.0002)	1.000000	0.969130	0.887891	0.927210
(5.0, 0.002)	0.969130	1.000000	0.965149	0.977178
(10.0, 0.0002)	0.887891	0.965149	1.000000	0.992116
(10.0, 0.002)	0.927210	0.977178	0.992116	1.000000

Even though these values are not necessarily easy to interpret overall this still shows that using the JS divergence it's easy to compare any two empirical distributions.

Group by *BST*

[105]:
```
M_js_bst = np.zeros([3, 3])

for i in range(3):
    for j in range(3):
        x1, y1 = compute_pdf(time_vec = df.loc[df.bin_system_type == i,␣
 ↪'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.bin_system_type == j,␣
 ↪'tmerg'])

        t_range = np.linspace(max(x1.min(), x2.min()), min(x1.max(), x2.max()))

        i1 = interp1d(x1, y1)
        i2 = interp1d(x2, y2)

        M_js_bst[i, j] = 1 - JS_Div(i1(t_range), i2(t_range))

print('JS similarity between different BST values:')
pd.DataFrame(M_js_bst, index = list(bst_types.values()), columns =␣
 ↪list(bst_types.values()))
# pd.DataFrame(M_js_bst, index = list(bst_types.values()), columns =␣
 ↪list(bst_types.values())).style.background_gradient(cmap = rg_cmap, vmin =␣
 ↪0, vmax = 1, axis = None).to_html()
```

```
JS similarity between different BST values:
```

[105]:

	BH-NS	NS-NS	BH-BH
BH-NS	1.000000	0.836752	0.925295

```
NS-NS   0.836752   1.000000   0.643277
BH-BH   0.925295   0.643277   1.000000
```

[57]:
```
print(f'Percent JS similarity between BH-NS and NS-NS: {M_js_bst[0, 1]:.3%}')
print(f'Percent JS similarity between BH-NS and BH-BH: {M_js_bst[0, 2]:.3%}')
print(f'Percent JS similarity between NS-NS and BH-BH: {M_js_bst[1, 2]:.3%}')
```

```
Percent JS similarity between BH-NS and NS-NS: 83.675%
Percent JS similarity between BH-NS and BH-BH: 92.530%
Percent JS similarity between NS-NS and BH-BH: 64.328%
```

With this grouping it is easier to interpret the values, both due to the small number of values and their "discrete" meaning. In particular we obtain an intuitive result: the more compact objects are in common, the higher the similarity score will be.

6.3 Kolmogorov-Smirnov test

6.3.1 Mathematical definition

Another possible way of defining a similarity score is using the *Kolmogorov-Smirnov test*, that relies on frequentist statistics; in particular this algorithm defines an hypothesis test about whether a certain dataset may be coming from a certain distribution. The KS test is used to either compare a dataset with a target distribution or two datasets' underlying distributions.

From Wikipedia: > In essence, the test answers the question "How likely is it that we would see a collection of samples like this if they were drawn from that probability distribution?" or, in the second case, "How likely is it that we would see two sets of samples like this if they were drawn from the same (but unknown) probability distribution?".

The test revolves around computing the Kolmogorov test statistic $KS = \sup_x |P(x) - Q(x)|$, which can be shown to be in the bounded interval $[0, 1]$; values near zero indicate the densities are evenly distributed between two samples, while values near one indicate densities are not evenly distributed between two samples. As before we can therefore use $1 - KS$ to define a similarity score: values close to zero will mean poor similarity, while values close to 1 will mean strong similarity.

One can also perform the actual frequentist hypothesis test, since the KS's distribution is known: the null hypothesis for KS test states that there is no difference between the two distributions, while the alternative hypothesis states that the two datasets are from different distributions. By computing this p-value we can therefore give a straight "yes or no" answer to whether the differences between the empirical and theoretical distributions are "significant" in the frequentist hypothesis test sense.

6.3.2 Using the KS statistics

Due to the above we can use KS in two distinct ways: - The first approach is to use KS alone (or rather $1 - KS$) as a similarity score, exactly as we did with JS. Now this score is based on a probabilistic approach instead of an information theory-based one, but it still gives a continuous answer to the question "how significant is the difference between distributions?", i.e. the answer is a scalar variable. - The second approach is to perform the full hypothesis test, i.e. we compute the p-values associated to each obtained KS and, depending on whether or not $p < 0.05$, we answer

the same question with a straight "yes or no", i.e. swap a scalar variable with a *categorical* one. Of course one can use any threshold for the p-value; 5% is simply a conventional value.

Let us explain more in detail how one may employ the second approach. We know that as $1 - KS$ decreases the corresponding p-value does too; this follows from the interpretation of these quantities - $1 - KS$ is close to 0 for very different distributions, the same for which the p-value will be low (as this p-value correspond to the null hypothesis, stating there are no differences between distributions). Therefore by checking when the p-value is significant we can deduce a threshold for KS itself, i.e. a value that will yield the "yes or no" answer based on the similarity score itself.

6.3.3 KS vs JS

This is an advantage compared to the JS similarity score, as we now have a non-arbitrary way of converting the continuous answer into a categorical one, i.e. we no longer need to arbitrarily fix how close is "close enough". This more refined sense of what it means to be "close enough" or "similar enough" comes at a cost, though, as the KS similarity score has a less intuitive interpretation. Indeed we argue that "distance based on shared information content" is an easy-to-grasp concept, while frequentist hypothesis tests and p-values are arguably controversial and unintuitive concepts, especially when compared to their bayesian/information theory counterparts. In particular we remark that the JS similarity is more rigorous than the KS one, because while the former is well-defined on its own the latter should only technically be used to compute the corresponding p-value: the correlation between low/high KS value and "similarity between distributions" is only indirect, with the p-values acting as liaison.

We also argue that in general a nuanced, continuous answer to the question about the significance of distribution deviations is more useful than a categorical one; hence the main advantage of the KS similarity score is not as important.

For these reasons we prefer the JS similarity score; one can still use the KS one, but with caution (i.e. only in its original "discrete" hypothesis testing context or by accompanying the standalone computation of KS with other metrics, like JS).

6.3.4 Real or log-log space?

Even though entropy/mutual information/Jensen-Shannon divergence are not necessarily preserved under a change of coordinates the Kolmogorov-Smirnov test relies only on probability distributions; since probability is conserved under a change of coordinates we expect that the choice of space won't make a difference, i.e. that the KS values will be the same in either coordinates - and as a consequence that the p-values will, too.

6.3.5 KS similarity between empirical and theoretical distributions

All data points

[58]:
```
def compute_KS_stat_all_statistic(time_vec):
    xx, yy = loglog_transform(time_vec)
    l, rss = linear_fit(time_vec)
    q_f, rss_f = linear_fit_fixed_slope(time_vec)
    x2, y2 = compute_pdf(time_vec)
    k, gamma = 10 ** l.intercept, l.slope
    k_f = 10 ** q_f
```

```
    f = lambda x: 1 - x.statistic

    return {
        'original_fixed':f(ks_2samp(y2, k_f / x2)),
        'original_variable':f(ks_2samp(y2, k * (x2 ** gamma))),
        'loglog_fixed':f(ks_2samp(yy, -xx + q_f)),
        'loglog_variable':f(ks_2samp(yy, l.slope * xx + l.intercept))
    }
```

[59]:
```
def compute_KS_stat_all_pvalue(time_vec):
    xx, yy = loglog_transform(time_vec)
    l, rss = linear_fit(time_vec)
    q_f, rss_f = linear_fit_fixed_slope(time_vec)
    x2, y2 = compute_pdf(time_vec)
    k, gamma = 10 ** l.intercept, l.slope
    k_f = 10 ** q_f

    f = lambda x: x.pvalue

    return {
        'original_fixed':f(ks_2samp(y2, k_f / x2)),
        'original_variable':f(ks_2samp(y2, k * (x2 ** gamma))),
        'loglog_fixed':f(ks_2samp(yy, -xx + q_f)),
        'loglog_variable':f(ks_2samp(yy, l.slope * xx + l.intercept))
    }
```

[60]:
```
KS_stat = compute_KS_stat_all_statistic(df.tmerg)
KS_pvalue = compute_KS_stat_all_pvalue(df.tmerg)

print('KS test statistic, all data points:')
print(f"original space, FIXED SLOPE: stat = {KS_stat['original_fixed']:.3f},␣
 ↪pvalue = {KS_pvalue['original_fixed']:.3f}")
print(f"original space, VARIABLE SLOPE: stat = {KS_stat['original_variable']:.
 ↪3f}, pvalue = {KS_pvalue['original_variable']:.3f}")
print('\n')
print(f"loglog space, FIXED SLOPE: stat = {KS_stat['loglog_fixed']:.3f}, pvalue␣
 ↪= {KS_pvalue['loglog_fixed']:.3f}")
print(f"loglog space, VARIABLE SLOPE: stat = {KS_stat['loglog_variable']:.3f},␣
 ↪pvalue = {KS_pvalue['loglog_variable']:.3f}")
```

```
KS test statistic, all data points:
original space, FIXED SLOPE: stat = 0.771, pvalue = 0.323
original space, VARIABLE SLOPE: stat = 0.800, pvalue = 0.492


loglog space, FIXED SLOPE: stat = 0.771, pvalue = 0.323
```

```
loglog space, VARIABLE SLOPE: stat = 0.800, pvalue = 0.492
```

We find results consistent with the JS similarity score; interestingly enough the actual numerical values are quite close, as in both cases we have a score between 0.7 and 0.8 in the same $[0, 1]$ interval. This is interesting because even though these two scores have a quite different meaning the fact that they are both normalized in the same interval makes them almost comparable - not only between themselves, actually, but also with $|r|$, as we will see later on.

As before we find the intuitive result that adding an extra parameter to the target distribution increases similarity (consequence of the fact that the theoretical distribution can be "bent" to be closer to the empirical one). This time, though, we find that real and log-log space yield the same value; for this reason we can compute KS in either coordinates.

An interesting remark is that both p-values are much larger than the usual 0.05 threshold; this seems to suggest that in our problem KS values larger than about 0.77 should be considered as "close enough" to 1, at least from the yes/no frequentist perspective. We will revisit this below, once we actually get p-values below the critical threshold.

Group by (A, Z) Let us repeat the KS test but with data samples grouped by (A, Z).

[106]:
```
KS_stat_AZ_statistic_dict = {}

for i, (A, Z) in enumerate(product(A_vec, Z_vec)):
    tmp = compute_KS_stat_all_statistic(df.loc[(df.A == A) & (df.Z == Z),␣
 ↪'tmerg'])
    tmp['A'], tmp['Z'] = A, Z
    KS_stat_AZ_statistic_dict[i] = tmp

KS_stat_AZ_statistic_df = pd.DataFrame.from_dict(KS_stat_AZ_statistic_dict,␣
 ↪'index')[['A', 'Z', 'original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable']]
print('KS similarity (data grouped by (A, Z)):')
KS_stat_AZ_statistic_df
# KS_stat_AZ_statistic_df.style.background_gradient(cmap = rg_cmap, vmin = 0,␣
 ↪vmax = 1, subset = ['original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable'])
```

```
KS similarity (data grouped by (A, Z)):
```

[106]:

	A	Z	original_fixed	original_variable	loglog_fixed	\
0	0.01	0.0002	0.781250	0.718750	0.781250	
1	0.01	0.0020	0.685714	0.828571	0.685714	
2	0.10	0.0002	0.714286	0.800000	0.714286	
3	0.10	0.0020	0.735294	0.823529	0.735294	
4	1.00	0.0002	0.911765	0.911765	0.911765	
5	1.00	0.0020	0.852941	0.882353	0.852941	
6	3.00	0.0002	0.937500	0.906250	0.937500	
7	3.00	0.0020	0.823529	0.794118	0.823529	
8	5.00	0.0002	0.806452	0.870968	0.806452	

```
9    5.00  0.0020        0.818182           0.757576      0.818182
10  10.00  0.0002        0.656250           0.468750      0.656250
11  10.00  0.0020        0.696970           0.545455      0.696970

    loglog_variable
0          0.718750
1          0.828571
2          0.800000
3          0.823529
4          0.911765
5          0.882353
6          0.906250
7          0.794118
8          0.870968
9          0.757576
10         0.468750
11         0.545455
```

Like in the JS section we notice that (this time irrespective of the coordinates set) some (A, Z) yield a much higher similarity than others. In particular the same pairs that gave a high similarity with the r and JS scores do here, too; the same holds for the poor similarity (A, Z) pairs. We notice that some rows reach a similarity score around 0.5; intuitively we expect scores like these to represent a significant difference between distributions, and we can check this by looking at the respective p-values.

[109]:
```
KS_stat_AZ_pvalue_dict = {}

for i, (A, Z) in enumerate(product(A_vec, Z_vec)):
    tmp = compute_KS_stat_all_pvalue(df.loc[(df.A == A) & (df.Z == Z), 'tmerg'])
    tmp['A'], tmp['Z'] = A, Z
    KS_stat_AZ_pvalue_dict[i] = tmp

KS_stat_AZ_pvalue_df = pd.DataFrame.from_dict(KS_stat_AZ_pvalue_dict,␣
 ↪'index')[['A', 'Z', 'original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable']]
print('KS p-values (data grouped by (A, Z)):')
# print('KS p-values (data grouped by (A, Z)) (p < 0.05 in blue):')
KS_stat_AZ_pvalue_df
# KS_stat_AZ_pvalue_df.style.apply(lambda x: ['background: royalblue' if v < 0.
 ↪05 else '' for v in x],
#                                  subset = ['original_fixed',␣
 ↪'original_variable', 'loglog_fixed', 'loglog_variable'])
```

KS p-values (data grouped by (A, Z)):

[109]:
```
        A       Z  original_fixed  original_variable  loglog_fixed  \
0    0.01  0.0002        0.433736           0.160069      0.433736
```

1	0.01	0.0020	0.062548	0.690184	0.062548
2	0.10	0.0002	0.115077	0.491645	0.115077
3	0.10	0.0020	0.185920	0.672684	0.185920
4	1.00	0.0002	0.999609	0.999609	0.999609
5	1.00	0.0020	0.863227	0.976192	0.863227
6	3.00	0.0002	1.000000	0.999326	1.000000
7	3.00	0.0020	0.672684	0.472802	0.672684
8	5.00	0.0002	0.615054	0.963430	0.615054
9	5.00	0.0020	0.654343	0.289924	0.654343
10	10.00	0.0002	0.044862	0.000174	0.044862
11	10.00	0.0020	0.096546	0.001897	0.096546

	loglog_variable
0	0.160069
1	0.690184
2	0.491645
3	0.672684
4	0.999609
5	0.976192
6	0.999326
7	0.472802
8	0.963430
9	0.289924
10	0.000174
11	0.001897

We observe that some (A, Z) pairs yield p-values smaller than the 5% threshold. We know that the corresponding KS values can therefore be considered "low enough", i.e. there is a significant difference between the empirical and target distributions - at least in the frequentist sense.

This is precisely what allows us to define a "yes or no" threshold value, as explained above. We therefore now display the KS values associated to these low p-values.

[111]:
```
tmp1, tmp2 = KS_stat_AZ_statistic_df.copy(), KS_stat_AZ_pvalue_df.copy()
tmp1 = tmp1.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'KS_f', 'original_variable':'KS_v'}, axis = 1)
tmp2 = tmp2.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'p_f', 'original_variable':'p_v'}, axis = 1)
tmp1.merge(tmp2, on = ['A', 'Z'])[['A', 'Z', 'KS_f', 'p_f', 'KS_v', 'p_v']]\
    # .style\
    # .apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in␣
 ↪x], subset = ['p_f', 'p_v'])\
    # .background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, subset =␣
 ↪['KS_f', 'KS_v'])\
    # .format(precision = 2, subset = 'A')\
    # .format(precision = 4, subset = 'Z')
```

[111]:

	A	Z	KS_f	p_f	KS_v	p_v
0	0.01	0.0002	0.781250	0.433736	0.718750	0.160069
1	0.01	0.0020	0.685714	0.062548	0.828571	0.690184
2	0.10	0.0002	0.714286	0.115077	0.800000	0.491645
3	0.10	0.0020	0.735294	0.185920	0.823529	0.672684
4	1.00	0.0002	0.911765	0.999609	0.911765	0.999609
5	1.00	0.0020	0.852941	0.863227	0.882353	0.976192
6	3.00	0.0002	0.937500	1.000000	0.906250	0.999326
7	3.00	0.0020	0.823529	0.672684	0.794118	0.472802
8	5.00	0.0002	0.806452	0.615054	0.870968	0.963430
9	5.00	0.0020	0.818182	0.654343	0.757576	0.289924
10	10.00	0.0002	0.656250	0.044862	0.468750	0.000174
11	10.00	0.0020	0.696970	0.096546	0.545455	0.001897

We notice indeed that the rows with $A = 10$, whose KS values are approximately in the $[0.5, 0.7]$ range, are significantly different from the target distribution according to the frequentist interpretation of "significance". This means that 0.7 can be used as an approximate threshold value for when the distribution differences become significant in the frequentist sense.

Notice that above, when using the complete dataset and the $p < 0.05$ criterion, we were able to state that $KS > 0.77$ was a good threshold for "probably insignificant distribution difference"; using the new information we can confirm this statement, consistent with the $KS \lesssim 0.7$ condition, which has been found to be a good criterion for "probably significant distribution difference":

According to the above we may therefore simply look for KS values below 0.7. If we do we obtain the following:

[113]:
```
tmp1, tmp2 = KS_stat_AZ_statistic_df.copy(), KS_stat_AZ_pvalue_df.copy()
tmp1 = tmp1.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'KS_f', 'original_variable':'KS_v'}, axis = 1)
tmp2 = tmp2.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'p_f', 'original_variable':'p_v'}, axis = 1)
tmp1.merge(tmp2, on = ['A', 'Z'])[['A', 'Z', 'KS_f', 'p_f', 'KS_v', 'p_v']]\
    .loc[(tmp1.KS_f < 0.7) | (tmp1.KS_v < 0.7)]\
    # .style\
    # .apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in␣
 ↪x], subset = ['p_f', 'p_v'])\
    # .background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, subset =␣
 ↪['KS_f', 'KS_v'])\
    # .format(precision = 2, subset = 'A')\
    # .format(precision = 4, subset = 'Z')
```

[113]:

	A	Z	KS_f	p_f	KS_v	p_v
1	0.01	0.0020	0.685714	0.062548	0.828571	0.690184
10	10.00	0.0002	0.656250	0.044862	0.468750	0.000174
11	10.00	0.0020	0.696970	0.096546	0.545455	0.001897

Evidently this criterion can fail: above we catched the $(A, Z) = (0.01, 0.002)$ pair, whose p-value is slightly larger than the 5% threshold (at least for the fixed slope case). This means that this p-value

based approach is probably too "yes or no"; it is probably better to keep some nuance, i.e. using KS as a similarity score instead of as a quantity to be compared with a critical value. This way one can judge on a case-by-case basis whether the deviations are significant or not, especially with the help of the other metrics.

To achieve this we now display and compare the values all similarity metrics defined in this section.

[115]:
```
all_stat_df_AZ = JS_AZ_df_vs_r.copy().rename(columns = {'original_fixed':
 ↪'JS_f', 'original_variable':'JS_v'})
all_stat_df_AZ[['KS_f', 'KS_v']] = KS_stat_AZ_statistic_df[['original_fixed',␣
 ↪'original_variable']].rename(columns = {'original_fixed':'KS_f',␣
 ↪'original_variable':'KS_v'})
all_stat_df_AZ = all_stat_df_AZ[['A', 'Z', 'r', 'JS_f', 'KS_f', 'JS_v', 'KS_v']]
print('all statistics (data grouped by (A, Z)):')
all_stat_df_AZ#.style.background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1,␣
 ↪subset = ['r', 'JS_f', 'KS_f', 'JS_v', 'KS_v'])
```

```
all statistics (data grouped by (A, Z)):
```

[115]:

	A	Z	r	JS_f	KS_f	JS_v	KS_v
0	0.01	0.0002	0.783938	0.862606	0.781250	0.815145	0.718750
1	0.01	0.0020	0.740661	0.483565	0.685714	0.440631	0.828571
2	0.10	0.0002	0.877938	0.848723	0.714286	0.930782	0.800000
3	0.10	0.0020	0.928538	0.956936	0.735294	0.924006	0.823529
4	1.00	0.0002	0.907611	0.813951	0.911765	0.832274	0.911765
5	1.00	0.0020	0.819984	0.768389	0.852941	0.856961	0.882353
6	3.00	0.0002	0.938504	0.810292	0.937500	0.819144	0.906250
7	3.00	0.0020	0.610048	0.616924	0.823529	0.753629	0.794118
8	5.00	0.0002	0.836325	0.759532	0.806452	0.856870	0.870968
9	5.00	0.0020	0.586457	0.558332	0.818182	0.774734	0.757576
10	10.00	0.0002	0.106833	0.531173	0.656250	0.923378	0.468750
11	10.00	0.0020	0.147238	0.484555	0.696970	0.904153	0.545455

As said above the JS score especially becomes less reliable in the "fixed slope" case, so to compare only the metrics that are for sure useful we should remove the `JS_v` column. For "fair play" we also remove the corresponding `KS_v` column; this also simplifies the similarity score-based approach, since we only need to compare three numbers for every (A, Z) configuration.

[116]:
```
print('all reliable/useful statistics (data grouped by (A, Z)):')
mini_all_stat_df_AZ = all_stat_df_AZ.copy().drop(['JS_v', 'KS_v'], axis = 1).
 ↪rename({'JS_f':'JS', 'KS_f':'KS'}, axis = 1)
mini_all_stat_df_AZ#.style.background_gradient(cmap = rg_cmap, vmin = 0, vmax =␣
 ↪1, subset = ['r', 'JS', 'KS'])
```

```
all reliable/useful statistics (data grouped by (A, Z)):
```

[116]:

	A	Z	r	JS	KS
0	0.01	0.0002	0.783938	0.862606	0.781250

1	0.01	0.0020	0.740661	0.483565	0.685714
2	0.10	0.0002	0.877938	0.848723	0.714286
3	0.10	0.0020	0.928538	0.956936	0.735294
4	1.00	0.0002	0.907611	0.813951	0.911765
5	1.00	0.0020	0.819984	0.768389	0.852941
6	3.00	0.0002	0.938504	0.810292	0.937500
7	3.00	0.0020	0.610048	0.616924	0.823529
8	5.00	0.0002	0.836325	0.759532	0.806452
9	5.00	0.0020	0.586457	0.558332	0.818182
10	10.00	0.0002	0.106833	0.531173	0.656250
11	10.00	0.0020	0.147238	0.484555	0.696970

A simple way to quantify the agreement between different similarity scores is to compute the (normalized) covariance between them.

[117]:
```
print('normalized correlation between all statistics (data grouped by (A, Z)):')
all_stat_df_AZ.drop(['A', 'Z'], axis = 1).corr('pearson')#.style.
 ↪background_gradient(cmap = rg_cmap, axis = None, vmin = 0, vmax = 1)
```

```
normalized correlation between all statistics (data grouped by (A, Z)):
```

[117]:

	r	JS_f	KS_f	JS_v	KS_v
r	1.000000	0.773915	0.577101	-0.136999	0.929729
JS_f	0.773915	1.000000	0.417217	0.446229	0.559992
KS_f	0.577101	0.417217	1.000000	0.035298	0.707710
JS_v	-0.136999	0.446229	0.035298	1.000000	-0.268679
KS_v	0.929729	0.559992	0.707710	-0.268679	1.000000

Without removing the unreliable scores the correlation is terrible; if we remove them the above matrix improves quite a bit.

[118]:
```
print('normalized correlation between all reliable/useful statistics (data␣
 ↪grouped by (A, Z)):')
mini_all_stat_df_AZ.drop(['A', 'Z'], axis = 1).corr('pearson')#.style.
 ↪background_gradient(cmap = rg_cmap, axis = None, vmin = 0, vmax = 1)
```

```
normalized correlation between all reliable/useful statistics (data grouped by
(A, Z)):
```

[118]:

	r	JS	KS
r	1.000000	0.773915	0.577101
JS	0.773915	1.000000	0.417217
KS	0.577101	0.417217	1.000000

The improvement in covariance is quite significant, especially considering how simplistic it is as a correlation metric. This is once again a good reason for sticking to the basic case (fixed slope fit).

In any case the above matrix confirms the intuitive (but nontrivial) result that the three metrics are consistent with each other, and can therefore be used in conjuction with no issues. In par-

ticular we remark the following: the fact that we can define some similarity scores using different approaches and that these different values are in good agreement gives us confidence in deciding e.g. which (A, Z) pairs produce distributions closer/further from the target heuristics. More generally: the agreement between different similarity scores gives a universal meaning to "significance in distribution deviations".

Group by *BST* The previous analysis is quite general, and can be easily be repeated e.g. by grouping by binary system type.

[119]:
```
KS_stat_BST_statistic_dict = {}

for bst in range(3):
    tmp = compute_KS_stat_all_statistic(df.loc[df.bin_system_type == bst,␣
 ↪'tmerg'])
    tmp['BST'] = bst_types[bst]
    KS_stat_BST_statistic_dict[bst] = tmp

KS_stat_BST_statistic_df = pd.DataFrame.from_dict(KS_stat_BST_statistic_dict,␣
 ↪'index')[['BST', 'original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable']]
print('KS similarity (data grouped by BST):')
KS_stat_BST_statistic_df#.style.background_gradient(cmap = rg_cmap, vmin = 0,␣
 ↪vmax = 1, subset = ['original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable'])
```

```
KS similarity (data grouped by BST):
```

[119]:

	BST	original_fixed	original_variable	loglog_fixed	loglog_variable
0	BH-NS	0.87500	0.875000	0.87500	0.875000
1	NS-NS	0.90625	0.812500	0.90625	0.812500
2	BH-BH	0.80000	0.914286	0.80000	0.914286

[120]:
```
print('KS similarity (data grouped by BST) (useless loglog transform removed):')
KS_stat_BST_statistic_df\
    .copy()\
    .drop(['loglog_fixed', 'loglog_variable'], axis = 1)\
    #.style.background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, subset =␣
 ↪['original_fixed', 'original_variable'])
```

```
KS similarity (data grouped by BST) (useless loglog transform removed):
```

[120]:

	BST	original_fixed	original_variable
0	BH-NS	0.87500	0.875000
1	NS-NS	0.90625	0.812500
2	BH-BH	0.80000	0.914286

When grouping by BST we find generally high KS similarity scores, all above our 0.7 threshold; indeed their p-values are all above 0.05, as can be easily verified:

[121]:
```
KS_stat_BST_pvalue_dict = {}

for bst in range(3):
    tmp = compute_KS_stat_all_pvalue(df.loc[df.bin_system_type == bst, 'tmerg'])
    tmp['BST'] = bst_types[bst]
    KS_stat_BST_pvalue_dict[bst] = tmp

KS_stat_BST_pvalue_df = pd.DataFrame.from_dict(KS_stat_BST_pvalue_dict,␣
 ↪'index')[['BST', 'original_fixed', 'original_variable', 'loglog_fixed',␣
 ↪'loglog_variable']]
print('KS p-values (data grouped by BST):')
KS_stat_BST_pvalue_df#.style.apply(lambda x: ['background: royalblue' if v < 0.
 ↪05 else '' for v in x], subset = ['original_fixed', 'original_variable']).
 ↪to_html()
```

```
KS p-values (data grouped by BST):
```

[121]:

	BST	original_fixed	original_variable	loglog_fixed	loglog_variable
0	BH-NS	0.968290	0.968290	0.968290	0.968290
1	NS-NS	0.999326	0.635139	0.999326	0.635139
2	BH-BH	0.491645	0.999703	0.491645	0.999703

[122]:
```
print('KS similarity < 0.7 (data grouped by BST):')
KS_stat_BST_statistic_df.loc[(KS_stat_BST_statistic_df.original_fixed < 0.7) |␣
 ↪(KS_stat_BST_statistic_df.original_variable < 0.7) |␣
 ↪(KS_stat_BST_statistic_df.loglog_fixed < 0.7) | (KS_stat_BST_statistic_df.
 ↪loglog_variable < 0.7)]
```

```
KS similarity < 0.7 (data grouped by BST):
```

[122]:
```
Empty DataFrame
Columns: [BST, original_fixed, original_variable, loglog_fixed, loglog_variable]
Index: []
```

[123]:
```
tmp1, tmp2 = KS_stat_BST_statistic_df.copy(), KS_stat_BST_pvalue_df.copy()
tmp1 = tmp1.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'KS_f', 'original_variable':'KS_v'}, axis = 1)
tmp2 = tmp2.drop(['loglog_fixed', 'loglog_variable'], axis = 1).
 ↪rename({'original_fixed':'p_f', 'original_variable':'p_v'}, axis = 1)
tmp1.merge(tmp2, on = 'BST')[['BST', 'KS_f', 'p_f', 'KS_v', 'p_v']]\
    # .style\
    # .apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in␣
 ↪x], subset = ['p_f', 'p_v'])\
    # .background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, subset =␣
 ↪['KS_f', 'KS_v'])
```

[123]:

	BST	KS_f	p_f	KS_v	p_v
0	BH-NS	0.87500	0.968290	0.875000	0.968290
1	NS-NS	0.90625	0.999326	0.812500	0.635139
2	BH-BH	0.80000	0.491645	0.914286	0.999703

By using the same "yes or no" discrete approach as before we find that all distribution deviations are insignificant according to the frequentist approach, which once again showcases that it's best to keep and interpret a numerical score instead of "compressing" it to a single yes/no discrete test result.

Still we remark that the above result is reasonable: we already found that mixing together different (A, Z) pairs has a smoothing, regularizing effect on the distribution - so it makes sense that both KS and the p-values will be larger than their respective thresholds.

6.3.6 KS similarity between different empirical distributions

Similarly to what we did with the JS similarity we can use the KS one to compare different empirical distributions, this time by directly comparing the underlying distributions instead of their information content.

Group by A

[126]:
```
M_ks_a = np.zeros([len(A_vec), len(A_vec)])
M_ks_a_pvalue = np.zeros([len(A_vec), len(A_vec)])

for i, A1 in enumerate(A_vec):
    for j, A2 in enumerate(A_vec):
        x1, y1 = compute_pdf(time_vec = df.loc[df.A == A1, 'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.A == A2, 'tmerg'])

        M_ks_a[i, j] = 1 - ks_2samp(y1, y2).statistic
        M_ks_a_pvalue[i, j] = ks_2samp(y1, y2).pvalue

print('KS similarity between different A values:')
display(pd.DataFrame(M_ks_a, index = A_vec, columns = A_vec))
# display(pd.DataFrame(M_ks_a, index = A_vec, columns = A_vec).style.
  ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None))
print('KS p-values between different A values:')
# print('KS p-values between different A values (p < 0.05 in blue):')
display(pd.DataFrame(M_ks_a_pvalue, index = A_vec, columns = A_vec))
# display(pd.DataFrame(M_ks_a_pvalue, index = A_vec, columns = A_vec).style.
  ↪apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in x]))
```

```
KS similarity between different A values:
```

	0.01	0.10	1.00	3.00	5.00	10.00
0.01	1.000000	0.742857	0.672269	0.789916	0.748918	0.371429
0.10	0.742857	1.000000	0.783193	0.734454	0.754113	0.628571
1.00	0.672269	0.783193	1.000000	0.882353	0.794118	0.441176
3.00	0.789916	0.734454	0.882353	1.000000	0.764706	0.529412

```
5.00   0.748918  0.754113  0.794118  0.764706  1.000000  0.484848
10.00  0.371429  0.628571  0.441176  0.529412  0.484848  1.000000
```

```
KS p-values between different A values:
```

	0.01	0.10	1.00	3.00	5.00	10.00
0.01	1.000000e+00	0.199054	0.040938	0.379442	0.188814	5.314657e-07
0.10	1.990539e-01	1.000000	0.330033	0.147623	0.209858	1.278262e-02
1.00	4.093791e-02	0.330033	1.000000	0.976192	0.380075	1.384365e-05
3.00	3.794424e-01	0.147623	0.976192	1.000000	0.236235	4.758006e-04
5.00	1.888136e-01	0.209858	0.380075	0.236235	1.000000	2.369774e-04
10.00	5.314657e-07	0.012783	0.000014	0.000476	0.000237	1.000000e+00

Group by Z

[127]:
```
M_ks_z = np.zeros([len(Z_vec), len(Z_vec)])
M_ks_z_pvalue = np.zeros([len(Z_vec), len(Z_vec)])

for i, Z1 in enumerate(Z_vec):
    for j, Z2 in enumerate(Z_vec):
        x1, y1 = compute_pdf(time_vec = df.loc[df.Z == Z1, 'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.Z == Z2, 'tmerg'])

        M_ks_z[i, j] = 1 - ks_2samp(y1, y2).statistic
        M_ks_z_pvalue[i, j] = ks_2samp(y1, y2).pvalue

print('KS similarity between different Z values:')
display(pd.DataFrame(M_ks_z, index = Z_vec, columns = Z_vec))
# display(pd.DataFrame(M_ks_z, index = Z_vec, columns = Z_vec).style.
  ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None))
print('KS p-values < 0.05 between different Z values:')
# print('KS p-values < 0.05 between different Z values (p < 0.05 in blue):')
display(pd.DataFrame(M_ks_z_pvalue, index = Z_vec, columns = Z_vec))
# display(pd.DataFrame(M_ks_z_pvalue, index = Z_vec, columns = Z_vec).style.
  ↪apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in x]))
```

```
KS similarity between different Z values:
```

	0.0002	0.0020
0.0002	1.000000	0.857143
0.0020	0.857143	1.000000

```
KS p-values < 0.05 between different Z values:
```

	0.0002	0.0020
0.0002	1.000000	0.874495
0.0020	0.874495	1.000000

Group by (A, Z)

[129]:
```
M_ks_az = np.zeros([len(A_vec) * len(Z_vec), len(A_vec) * len(Z_vec)])
M_ks_az_pvalue = np.zeros([len(A_vec) * len(Z_vec), len(A_vec) * len(Z_vec)])

for i, (A1, Z1) in enumerate(product(A_vec, Z_vec)):
    for j, (A2, Z2) in enumerate(product(A_vec, Z_vec)):
        x1, y1 = compute_pdf(time_vec = df.loc[(df.A == A1) & (df.Z == Z1),␣
 ↪'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[(df.A == A2) & (df.Z == Z2),␣
 ↪'tmerg'])

        M_ks_az[i, j] = 1 - ks_2samp(y1, y2).statistic
        M_ks_az_pvalue[i, j] = ks_2samp(y1, y2).pvalue

tmp = list(product(A_vec, Z_vec))
print('KS similarity between different (A, Z) values:')
display(pd.DataFrame(M_ks_az, index = tmp, columns = tmp))
# display(pd.DataFrame(M_ks_az, index = tmp, columns = tmp).style.
 ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None))
print('KS pvalues between different (A, Z) values:')
# print('KS pvalues between different (A, Z) values (p < 0.05 in blue):')
display(pd.DataFrame(M_ks_az_pvalue, index = tmp, columns = tmp))
# display(pd.DataFrame(M_ks_az_pvalue, index = tmp, columns = tmp).style.
 ↪apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in x]))
```

KS similarity between different (A, Z) values:

	(0.01, 0.0002)	(0.01, 0.002)	(0.1, 0.0002)	(0.1, 0.002)	\
(0.01, 0.0002)	1.000000	0.626786	0.806250	0.612132	
(0.01, 0.002)	0.626786	1.000000	0.800000	0.724370	
(0.1, 0.0002)	0.806250	0.800000	1.000000	0.747059	
(0.1, 0.002)	0.612132	0.724370	0.747059	1.000000	
(1.0, 0.0002)	0.665441	0.587395	0.730252	0.735294	
(1.0, 0.002)	0.639706	0.646218	0.763025	0.764706	
(3.0, 0.0002)	0.718750	0.653571	0.716071	0.748162	
(3.0, 0.002)	0.669118	0.755462	0.808403	0.823529	
(5.0, 0.0002)	0.664315	0.670046	0.805530	0.745731	
(5.0, 0.002)	0.708333	0.755844	0.814719	0.814617	
(10.0, 0.0002)	0.687500	0.493750	0.600000	0.705882	
(10.0, 0.002)	0.740530	0.551515	0.630303	0.705882	

	(1.0, 0.0002)	(1.0, 0.002)	(3.0, 0.0002)	(3.0, 0.002)	\
(0.01, 0.0002)	0.665441	0.639706	0.718750	0.669118	
(0.01, 0.002)	0.587395	0.646218	0.653571	0.755462	
(0.1, 0.0002)	0.730252	0.763025	0.716071	0.808403	
(0.1, 0.002)	0.735294	0.764706	0.748162	0.823529	
(1.0, 0.0002)	1.000000	0.852941	0.882353	0.794118	
(1.0, 0.002)	0.852941	1.000000	0.840074	0.823529	
(3.0, 0.0002)	0.882353	0.840074	1.000000	0.841912	

(3.0, 0.002)	0.794118	0.823529	0.841912	1.000000
(5.0, 0.0002)	0.764706	0.823529	0.718750	0.807400
(5.0, 0.002)	0.795009	0.854724	0.778409	0.853832
(10.0, 0.0002)	0.470588	0.500000	0.531250	0.617647
(10.0, 0.002)	0.500891	0.529412	0.560606	0.647950

	(5.0, 0.0002)	(5.0, 0.002)	(10.0, 0.0002)	(10.0, 0.002)
(0.01, 0.0002)	0.664315	0.708333	0.687500	0.740530
(0.01, 0.002)	0.670046	0.755844	0.493750	0.551515
(0.1, 0.0002)	0.805530	0.814719	0.600000	0.630303
(0.1, 0.002)	0.745731	0.814617	0.705882	0.705882
(1.0, 0.0002)	0.764706	0.795009	0.470588	0.500891
(1.0, 0.002)	0.823529	0.854724	0.500000	0.529412
(3.0, 0.0002)	0.718750	0.778409	0.531250	0.560606
(3.0, 0.002)	0.807400	0.853832	0.617647	0.647950
(5.0, 0.0002)	1.000000	0.845552	0.482863	0.512219
(5.0, 0.002)	0.845552	1.000000	0.606061	0.636364
(10.0, 0.0002)	0.482863	0.606061	1.000000	0.859848
(10.0, 0.002)	0.512219	0.636364	0.859848	1.000000

KS pvalues between different (A, Z) values:

	(0.01, 0.0002)	(0.01, 0.002)	(0.1, 0.0002)	(0.1, 0.002)	\
(0.01, 0.0002)	1.000000	0.013213	0.480993	0.009942	
(0.01, 0.002)	0.013213	1.000000	0.491645	0.115077	
(0.1, 0.0002)	0.480993	0.491645	1.000000	0.162751	
(0.1, 0.002)	0.009942	0.115077	0.162751	1.000000	
(1.0, 0.0002)	0.036738	0.004118	0.138901	0.185920	
(1.0, 0.002)	0.020129	0.020961	0.247935	0.306758	
(3.0, 0.0002)	0.160069	0.026065	0.104730	0.203590	
(3.0, 0.002)	0.040471	0.211830	0.465279	0.672684	
(5.0, 0.0002)	0.040472	0.041476	0.487858	0.197323	
(5.0, 0.002)	0.100149	0.216250	0.532282	0.547280	
(10.0, 0.0002)	0.087682	0.000179	0.006004	0.088859	
(10.0, 0.002)	0.185793	0.001133	0.013223	0.074917	

	(1.0, 0.0002)	(1.0, 0.002)	(3.0, 0.0002)	(3.0, 0.002)	\
(0.01, 0.0002)	0.036738	0.020129	0.160069	0.040471	
(0.01, 0.002)	0.004118	0.020961	0.026065	0.211830	
(0.1, 0.0002)	0.138901	0.247935	0.104730	0.465279	
(0.1, 0.002)	0.185920	0.306758	0.203590	0.672684	
(1.0, 0.0002)	1.000000	0.863227	0.942362	0.472802	
(1.0, 0.002)	0.863227	1.000000	0.726568	0.672684	
(3.0, 0.0002)	0.942362	0.726568	1.000000	0.738768	
(3.0, 0.002)	0.472802	0.672684	0.738768	1.000000	
(5.0, 0.0002)	0.270130	0.603258	0.115738	0.507927	
(5.0, 0.002)	0.384436	0.787539	0.351620	0.780321	
(10.0, 0.0002)	0.000077	0.000245	0.001505	0.011154	
(10.0, 0.002)	0.000161	0.000476	0.002739	0.018497	

	(5.0, 0.0002)	(5.0, 0.002)	(10.0, 0.0002)	(10.0, 0.002)
(0.01, 0.0002)	0.040472	0.100149	0.087682	0.185793
(0.01, 0.002)	0.041476	0.216250	0.000179	0.001133
(0.1, 0.0002)	0.487858	0.532282	0.006004	0.013223
(0.1, 0.002)	0.197323	0.547280	0.088859	0.074917
(1.0, 0.0002)	0.270130	0.384436	0.000077	0.000161
(1.0, 0.002)	0.603258	0.787539	0.000245	0.000476
(3.0, 0.0002)	0.115738	0.351620	0.001505	0.002739
(3.0, 0.002)	0.507927	0.780321	0.011154	0.018497
(5.0, 0.0002)	1.000000	0.758811	0.000266	0.000544
(5.0, 0.002)	0.758811	1.000000	0.006854	0.024674
(10.0, 0.0002)	0.000266	0.006854	1.000000	0.841573
(10.0, 0.002)	0.000544	0.024674	0.841573	1.000000

Once again this shows how easily one can construct a different similarity score that can allow one to compare any two empirical distributions.

Group by *BST*

[131]:
```
M_ks_bst = np.zeros([3, 3])
M_ks_bst_pvalue = np.zeros([3, 3])

for i in range(3):
    for j in range(3):
        x1, y1 = compute_pdf(time_vec = df.loc[df.bin_system_type == i,␣
 ↪'tmerg'])
        x2, y2 = compute_pdf(time_vec = df.loc[df.bin_system_type == j,␣
 ↪'tmerg'])

        M_ks_bst[i, j] = 1 - ks_2samp(y1, y2).statistic
        M_ks_bst_pvalue[i, j] = ks_2samp(y1, y2).pvalue

tmp = list(bst_types.values())
print('KS similarity between different BST values:')
display(pd.DataFrame(M_ks_bst, index = tmp, columns = tmp))
# display(pd.DataFrame(M_ks_bst, index = tmp, columns = tmp).style.
 ↪background_gradient(cmap = rg_cmap, vmin = 0, vmax = 1, axis = None))
print('KS pvalues between different BST values:')
# print('KS pvalues between different BST values (p < 0.05 in blue):')
display(pd.DataFrame(M_ks_bst_pvalue, index = tmp, columns = tmp))
# display(pd.DataFrame(M_ks_bst_pvalue, index = tmp, columns = tmp).style.
 ↪apply(lambda x: ['background: royalblue' if v < 0.05 else '' for v in x]))
```

KS similarity between different BST values:

	BH-NS	NS-NS	BH-BH
BH-NS	1.0000	0.875000	0.837500
NS-NS	0.8750	1.000000	0.804464

```
BH-BH  0.8375  0.804464  1.000000

KS pvalues between different BST values:

          BH-NS    NS-NS     BH-BH
BH-NS  1.000000  0.96829  0.693948
NS-NS  0.968290  1.00000  0.467240
BH-BH  0.693948  0.46724  1.000000
```

[78]:
```
print(f'Percent KS similarity between BH-NS and NS-NS: similarity =␣
  ↪{M_ks_bst[0, 1]:.3%}, p-value = {M_ks_bst_pvalue[0, 1]:.3%}')
print(f'Percent KS similarity between BH-NS and BH-BH: similarity =␣
  ↪{M_ks_bst[0, 2]:.3%}, p-value = {M_ks_bst_pvalue[0, 2]:.3%}')
print(f'Percent KS similarity between NS-NS and BH-BH: similarity =␣
  ↪{M_ks_bst[1, 2]:.3%}, p-value = {M_ks_bst_pvalue[1, 2]:.3%}')
```

```
Percent KS similarity between BH-NS and NS-NS: similarity = 87.500%, p-value =
96.829%
Percent KS similarity between BH-NS and BH-BH: similarity = 83.750%, p-value =
69.395%
Percent KS similarity between NS-NS and BH-BH: similarity = 80.446%, p-value =
46.724%
```

We obtain a result consistent with the JS case: the more compact object are in common, the higher the similarity score will be.

7 Fitting t_d with a "simple" formula

7.1 Regression problem introduction

We now turn to a completely different problem: instead of analyzing the distribution of t values (either irrespective of or as a function of a limited number of parameters) we now ask whether one can *predict* t as a function of *all* available parameters.

This is clearly a regression problem: we know that a given set of parameters (initial masses, initial orbital parameters, etc.), once fed to MOBSE, will return a value of t_d - and we wish to learn (at least approximately) this parameters $\mapsto t_d$ mapping. Notice that this mapping is not necessarily completely deterministic (hence why we aim to learn an approximation thereof); this mapping may also be relatively complex. For these reasons we would like to find a compromise between the t-estimator's simplicity and its accuracy in computing t_d given a set of initial conditions. Ideally we should be able to find a single formula (or at least a simple algorithm) capable of accurately predicting t_d.

7.2 Regression problem setup

Considering this is a relatively straightforward statistics/ML regression problem we can define a few ground rules useful to proceed.

- Dataset definition: we define a vector input dataset made of all available physical parameters (i.e. we exclude e.g. the `ID` column, which is clearly unphysical) and a scalar output target dataset consisting of the corresponding t values. One may try to simplify further by only

including *some* physical parameters, but this may substantially reduce prediction accuracy; also *a priori* it's not necessarily clear which variables are more important in determining t, so one may need to test potentially many different combinations. For these reason we choose the simplest possible approach of including all of them; we will revisit this point at the end of this work. Given this dataset we split it in $80-20\%$ proportions to define the training/test set.

- Loss definition: as is common practice in regression problems we optimize our algorithms by minimizing the standard euclidean $L2$ loss.
- Accuracy metric definition: once an algorithm has been trained we evaluate its performance by computing the following score on the test set, called the *coefficient of determination*:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \langle y \rangle)^2}$$

where y_i are the true values, $\hat{y}_i$ the predicted values, and $\langle y \rangle$ is the average of the true values. Clearly this metric compares the average $L2$ loss with the variance in the true y values; if the former is much smaller than the latter it means the prediction errors are small in the relevant y scale. This means that the closer R^2 is to to 1, the better, as 1 clearly is the best possible score; also notice that R^2 can be negative, because the model can be arbitrarily bad. We can recap the previous description by stating that R^2 is the $L2$ test error normalized w.r.t. $\mathrm{Var}(y)$ and subtracted from 1 so that instead of having 0 as the best possible score we have 1. This turns the $L2$ error into an easily interpretable quantity, because in its basic form one cannot easily tell whether its value is too large or small enough; R^2, instead, has a normalized max. value, which is enough for our purposes.

The previous rules can be used with a wide range of algorithms; in the following we will employ three of the most known and used ones, with the same dataset/loss/accuracy metric defined above.

[79]:
```
# define training/test dataset
input_dataset = df.copy()
input_dataset = input_dataset.drop(['ID', 'bin_system_type', 'tmerg'], axis = 1)
target = df.tmerg

X_train, X_test, y_train, y_test = train_test_split(input_dataset, target,␣
 ↪train_size = int(0.8 * df.shape[0]), shuffle = True, random_state = 1234)
print(f'training dataset shape: {X_train.shape}')
print(f'test dataset shape: {X_test.shape}')
```

```
training dataset shape: (215927, 15)
test dataset shape: (53982, 15)
```

7.3 Linear model fit

Probably the simplest possible regression model is the standard linear least squares regressor. It clearly satisfies our simplicity requirement, as it will allow us to compute t as a linear combination of the input features with the learned coefficients as weights; it also is very lightweight, allowing for a near instant training. The only remaining concern is how accurately it can predict t; we expect this model to be limited, as t is the result of complex physical processes that most likely require a nonlinear description.

[80]:
```
def print_model_scores(model, model_name, X_train = X_train, y_train = y_train,␣
↪X_test = X_test, y_test = y_test):
    L2_train_error_sqrt = np.sqrt(np.mean((model.predict(X_train) - y_train) **␣
↪2))
    y_train_std = y_train.std()
    L2_test_error_sqrt = np.sqrt(np.mean((model.predict(X_test) - y_test) ** 2))
    y_test_std = y_test.std()
    print(f'{model_name}:')
    print(f'R^2 train: {model.score(X_train, y_train):.4f}')
    print(f'R^2 test: {model.score(X_test, y_test):.4f}')
    print('\n')
    print(f'Average t error (Myr), training set: {L2_train_error_sqrt:.3e}, std␣
↪in y_train: {y_train_std:.3e}, ratio: {L2_train_error_sqrt/y_train_std:.3f}')
    print(f'Average t error (Myr), test set: {L2_test_error_sqrt:.3e}, std in␣
↪y_test: {y_test_std:.3e}, ratio: {L2_test_error_sqrt/y_test_std:.3f}')
```

[81]:
```
linear_model = LinearRegression().fit(X_train, y_train)
print_model_scores(linear_model, 'Linear model')
```

```
Linear model:
R^2 train: 0.2075
R^2 test: 0.2126


Average t error (Myr), training set: 3.272e+03, std in y_train: 3.676e+03,
ratio: 0.890
Average t error (Myr), test set: 3.266e+03, std in y_test: 3.681e+03, ratio:
0.887
```

As we can see the linear model - while achieving a decent score - is too simple to produce accurate results: the R^2 is about 0.2 due to the fact that the average error in predicting t is of about the same order of magnitude as the standard deviation of t itself. We also remark that the model does not suffer from overfitting, as the training and test scores are essentially equal; this makes the model trustworthy, if inaccurate.

If this error can be tolerated for the task at hand then we have obtained one of the simplest possible formulae for t that still guarantees decent accuracy:

[82]:
```
print('t = ' + ' + '.join([f'({coef:.2f} * {var})' for coef, var in␣
↪zip(linear_model.coef_, list(X_train.columns))]))
```

```
t = (438.40 * A) + (527775.37 * Z) + (-39.01 * min1) + (-67.54 * min2) + (29.61
* tform) + (1.48 * sepform) + (-399.63 * eccform) + (637.20 * k1form) + (107.56
* m1form) + (288.19 * k2form) + (119.49 * m2form) + (1646.63 * k1) + (12.52 *
m1) + (965.15 * k2) + (42.71 * m2)
```

Let us search now for more complex t formulae; in particular we want to understand how accurately can one predict t (as said above t may not be completely deterministic, which if true would make reaching $R^2 = 1$ impossible in practice).

7.4 Polynomial model fit

A trivial modification to the basic linear regressor is to include all possible products *with a fixed maximum degree* of the available variables. The final formula will still be linear in the learned coefficients (which is why we can use an unmodified least square optimization), while being more expressive and therefore powerful. The issue is that the computational complexity scales polynomially, which in practice means we can only reasonably use a small polynomial degree.

Let us see what happens with this approach.

[83]:
```
poly_features = PolynomialFeatures(degree = 2, include_bias = False)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

poly_model = LinearRegression().fit(X_train_poly, y_train)
print_model_scores(poly_model, 'Polynomial model (degree 2)', X_train =␣
↪X_train_poly, X_test = X_test_poly)
```

```
Polynomial model (degree 2):
R^2 train: 0.6355
R^2 test: 0.6323


Average t error (Myr), training set: 2.219e+03, std in y_train: 3.676e+03,
ratio: 0.604
Average t error (Myr), test set: 2.232e+03, std in y_test: 3.681e+03, ratio:
0.606
```

We notice that by moving the model to degree $= 2$ we were able to increase its performance without a significantly worse training time: R^2 has more than doubled, and the average t error has become less than $2/3$ of the standard deviation in t. We also notice that the training and test scores are still very close to each other, meaning the model does not overfit yet, as before.

This means we can try to further increase the max degree:

[84]:
```
poly_features = PolynomialFeatures(degree = 3, include_bias = False)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

poly_model = LinearRegression().fit(X_train_poly, y_train)
print_model_scores(poly_model, 'Polynomial model (degree 3)', X_train =␣
↪X_train_poly, X_test = X_test_poly)
```

```
Polynomial model (degree 3):
R^2 train: 0.8858
R^2 test: -0.1074


Average t error (Myr), training set: 1.242e+03, std in y_train: 3.676e+03,
ratio: 0.338
```

```
Average t error (Myr), test set: 3.873e+03, std in y_test: 3.681e+03, ratio:
1.052
```

We observe that increasing the degree to 3 has forced the model to overfit: even though R^2 and the average t error over the training set have significantly improved the model cannot generalize this good performance to the test set - as we can notice from the negative test R^2, which is a consequence of the fact that the average t error on the test set is larger than σ_t itself.

This means that we should either stop at the second degree polynomial model or consider a completely different algorithm; in any case we have reached the limits of this simple regressor.

7.5 Decision tree fit

We now consider a slightly more complex model, still restraining ourselves to interpretable models, i.e. "transparent" models whose predictions are computed in a human-readable fashion. This is an essential requirement if we want to be able to still write down a "simple" formula for t (of course "simple" is a relative term depending on the user's preferences).

In particular we are interested in finding out if we can reach a significantly better accuracy by increasing the model complexity, while still using "simple" algorithms. For this reason we consider *decision trees models*, that make predictions by progressively checking in which intervals the input features reside.

[85]:
```
tree_model = DecisionTreeRegressor().fit(X_train, y_train)
print_model_scores(tree_model, 'Decision tree model')
```

```
Decision tree model:
R^2 train: 1.0000
R^2 test: 0.9618
```

```
Average t error (Myr), training set: 1.044e-13, std in y_train: 3.676e+03,
ratio: 0.000
Average t error (Myr), test set: 7.191e+02, std in y_test: 3.681e+03, ratio:
0.195
```

Even though this model also is quite simple computationally (it still completes training in a few seconds) it achieves no overfitting and basically perfect performance: the training score reaches the maximum possible value, but the test score is only slightly lower. We notice that the error in predicting t values from the test set is on average an order of magnitude less than σ_t, which is why R^2 is so good.

Judging by performance alone this algorithm already yields the best performance we could ask for, and for this reason it makes no sense to consider more complex ones (such as neural networks, that would also require more work to train due to e.g. the need for hyperparameter optimization). Of course we also want a simple formula predicting t; while in this case the formula is much longer (too long to comfortably print here) it is still simple due to its intrinsic interpretability; for this reason we can declare this decision tree as one of the best possible compromises between simplicity and accuracy. Of course if more simplicity is required one can always revert to the previous linear/polynomial models, at the price of a reduced accuracy in predicting t.

We also remark this: the fact that we proved that one can reach $R^2 \approx 1$ also means that the mapping parameters $\mapsto t_d$ can be considered essentially deterministic.

7.6 "Mini" decision tree: how dominant are the initial conditions? What about A and Z?

Given how cheap it is to train a successful decision tree regressor we can retrain it on a simplified dataset to further investigate the deterministic relation between input parameters and t output. In particular we remark that some of the dataset features describe the system at an intermediate time, i.e. some time after the beginning but not yet at the end; physically we expect that these initial conditions should be enough to determine the delay time - indeed as we noted above the evolution of the system can be considered deterministic to excellent approximation. For this reason we now retrain the decision tree regressor using only informations about the initial state of the system.

[86]:
```
df0 = input_dataset.copy().drop(['k1', 'm1', 'k2', 'm2'], axis = 1)
X_train_mini, X_test_mini, y_train, y_test = train_test_split(df0, target,␣
 ↪train_size = int(0.8 * df.shape[0]), shuffle = True, random_state = 1234)

tree_model_mini = DecisionTreeRegressor().fit(X_train_mini, y_train)
print_model_scores(tree_model_mini, 'Mini decision tree (initial conditions␣
 ↪only)', X_train = X_train_mini, X_test = X_test_mini)
```

```
Mini decision tree (initial conditions only):
R^2 train: 1.0000
R^2 test: 0.9542


Average t error (Myr), training set: 1.037e-13, std in y_train: 3.676e+03,
ratio: 0.000
Average t error (Myr), test set: 7.875e+02, std in y_test: 3.681e+03, ratio:
0.214
```

We notice that the scores have remained approximately unchanged, consistent with the fact that the initial conditions should be enough to determine the evolution of the system. The score has actually slightly decreased (this makes sense: we removed some potentially useful information), but not significantly, which means that indeed the initial conditions dominate over the others.

On a similar note we may also try to further remove A and Z; depending on whether/how much the score changes we can infer their importance in determining t in a new, indirect way.

[87]:
```
df0 = input_dataset.copy().drop(['A', 'Z', 'k1', 'm1', 'k2', 'm2'], axis = 1)
X_train_mini, X_test_mini, y_train, y_test = train_test_split(df0, target,␣
 ↪train_size = int(0.8 * df.shape[0]), shuffle = True, random_state = 1234)

tree_model_mini = DecisionTreeRegressor().fit(X_train_mini, y_train)
print_model_scores(tree_model_mini, 'Mini decision tree (initial conditions␣
 ↪only, A and Z removed)', X_train = X_train_mini, X_test = X_test_mini)
```

```
Mini decision tree (initial conditions only, A and Z removed):
```

```
R^2 train: 1.0000
R^2 test: 0.9588


Average t error (Myr), training set: 7.742e-04, std in y_train: 3.676e+03,
ratio: 0.000
Average t error (Myr), test set: 7.471e+02, std in y_test: 3.681e+03, ratio:
0.203
```

We notice that by also removing A and Z the score is almost completely unmodified. This is consistent with the evidence accumulated above: the correlation between (A, Z) and t is very weak, because the other parameters matter a lot more in determining t's value. This statement is not as precise as when we computed the mutual information, but still consistent with it, which makes us more confident in the obtained results. We also observe that removing (A, Z) actually slightly improves the score; instead of taking this to mean that knowing (A, Z) is detrimental in estimating t the more likely conclusion is that the variation in R^2 is not significant, i.e. the score can be considered essentially unchanged. This follows from the fact that the variation is tiny (less than 1%).