Appendix

! please, if possible view the .html version of this file, as it is interactive !

This is the Jupyter notebook relevant to Emiel and Freek's 2020 Web Science final project report.
It will read a little like our exploratory programming journey. Each code block will be prefaced by a short explanation.

Section I: Building and simulating the model

Step -1: imports

Of course we start with imports necessary for the notebook to work.

In [19]:
```
#Imports
from __future__ import division

import random
import numpy as np
import plotly.graph_objects as go

from IPython.display import Markdown as md

print "done importing"
```

```
done importing
```

Step 0 : Setting variables

The next cell sets some starting variables to work with. They are small enough to be able to run interesting function against them without having to wait a long time, while still exhibitng behaviour that we want to explore later with larger numbers.

In [20]:
```
##aquire the data with the following global variables

N = 100              #people participate in the market
start_cash = 100.00  #the starting budget per person
rounds = 100         #investment round that the particpants will partake in
fig_no = 0           #to keep track of figure numbers
```

Step 1 : Generating the data

The programming starts with functions that are able to take the parameters above and turn them in to a workable dataset. The dataset contains a list of the monetary history of a person, starting at 100 and going from there. The model the history is based on can be found in the report. The last function, plots the generated data per round to get some starting insights in what we are working with.

In [21]:
```
if (N%2):
    print "please keep N even for smoother operations"

def func_fig_no():
    '''keeps track of figure numbers :)'''
    global fig_no
    fig_no += 1
    return "Fig. {}:  ".format(fig_no)

def func_bound_exp_p(mean, cap):
    '''returns a random number from an exponential distribution with average [mean]
capped at [cap] '''

    chance = cap+1
    while chance > cap:
        chance = np.random.exponential(mean)
    return chance


def func_redistr_wealth(people, d):
    '''takes a set of people as input and redistributes [d]% of each persons wealt
h, returns the new people'''

    budget = 0
    N = len(people)
    for i in range(N):
        take = people[i][-1] * d
        people[i][-1] -= take
        budget += take

    give = budget/N
    for i in range(N):
        people[i][-1] += give

    return people


def func_type_random(type_r, x):
    '''returns a random p up up to 1
    [type_r] = 'uni'formly with a max [x]
    [type_r] = 'exp'onentially with a cap [x]'''

    if type_r=='exp':
        return func_bound_exp_p(x, 1)
    else:
        return random.random() * x / 1


def func_inv_round(people, tax=0,
                   vola_type='uni', vola=0.5,
                   inve_type='uni', inve=1,
                   loss='mul'):
    '''
    a single round of investments applied on the people list, returns the modified
version
    '''
    N = len(people)
    sN = range(N) #range of indexes of the people participating
    random.shuffle(sN) #shuffled indexes so a random 50% win (first half of the shu
ffled indexes), the other 50% loses

    # percentage market gain/loss this round, up to 50% win or loss
    p = func_type_random(vola_type, vola)
```

Step 2 : plotting the data logarithmically

The first step towards exploring powerlaws is to get one time-unit of data visualised on a log-log plot, so that is exactly what we do. a function to sort a unit of t and a function to plot a unit of t (logarithmically).

In [22]:
```
def func_sorted_time(people, t):
    '''
    takes a time as input and returns the sorted list of wealth from that t
    '''

    N = len(people)
    money_at_t = (sorted([people[i][t] for i in range(N)]))[::-1] #sorts the wealth
of all people at t and reverses the sort (LtS)
    return money_at_t


def func_powerlaw_t(p_set, t, log=False):
    '''
    creates a (lin-lin or log-log) plot for supplied t
    '''

    people, title = p_set
    N = len(people)

    money_at_t = func_sorted_time(people, t)
    fpt_plot = go.Figure()
    fpt_plot.add_trace(go.Scatter(x = range(N), y = money_at_t))
    plot_type = "lin-lin"
    if log:
        fpt_plot.update_layout(xaxis_type="log", yaxis_type="log")
        plot_type = "log-log"

    fpt_plot.update_layout(
    title='{}individual wealth sorted at t={} on {}  ({})'.format(func_fig_no(), t,
plot_type, title),
    xaxis_title='nth person (n)',
    yaxis_title='individual wealth (€)')

    return fpt_plot



# plot the lin-lin and log-log plot for the last investment round
func_powerlaw_t(people_0, -1).show()
func_powerlaw_t(people_0, -1, log=True).show()
```

Step 3 : creating a powerlaw fit

Step 3.1: calculating the fit coefficients

For the data as displayed in the previous step we now want to be able to create a powerlaw fit. numpy's polyfit allows us to obtain the coefficients of such a fit. Running it on the last t gives us the results displayed below.

In [23]:
```
def func_polyfit_t(people, t):
    '''
    creates a np.polyfit for t and returns the [a,b] and [error]
    '''
    N = len(people)

    data = func_sorted_time(people, t)
    x = np.array(range(len(data))) + 1 # Array with same size as data, starting fro
m 1 to avoid pesky divide-by-zero errors
    polynomial_coefficients, residuals = np.polyfit(np.log(x),np.log(data), 1,full=
True)[:2] # Returns coefficients of a polynomial of degree 1 (just a linear relatio
n) with least square fit to data

    return polynomial_coefficients, residuals



#calculate the polyfit for the last investment round
polycoeffs, error = func_polyfit_t(people_0[0], -1)
print "set = {} @ t = {}".format(people_0[1], -1)
print "polynomial coefficients [a,b] : {}\n               residual/error : {}".form
at(polycoeffs, error)
```

```
set = practise set @ t = -1
polynomial coefficients [a,b] : [-2.27333717 10.46911892]
               residual/error : [163.51924746]
```

step 3.2: Putting the coefficients into the formula

This step is only to show the coefficients integrated into the formula. We used a trick to get it into a latex function, which is pretty dandy.

In [24]:
```
'''turn the previous polynomial coeffs into a nice latex formula using ipynb markdo
wn and python insertion'''

md("the best fitting power law for the log-log plot of the division of wealth is at
the last t is:  \n  \n$$y = e^{} + x^{}$$".format(("{"+str(round(polycoeffs[1],
3))+"}"), ("{"+str(round(polycoeffs[0],3))+"}")) )
```

Out[24]: the best fitting power law for the log-log plot of the division of wealth is at the last t is:

$$y = e^{10.469} + x^{-2.273}$$

Step 3.3: plotting the powerlaw fit

Pretty self-explanatory, we plot the data for the last t, with the fit calculated in step 3.1, translated into a line by using numpy's poly1d function.

In [25]:
```
def func_plots_with_fit(p_set, t, log = True):
    '''
    plot the data of round t with the best fitting power law
    '''
    people, title = p_set
    N = len(people)

    data = func_sorted_time(people, t)

    polycoeffs, error = func_polyfit_t(people, t)

    polynomial = np.poly1d(polycoeffs) #using poly1d to make a function from the va
riables gained previously

    x = np.array(range(len(data))) + 1 # Array with same size as data, starting fro
m 1 to avoid pesky divide-by-zero errors

    log_plot = go.Figure()
    log_plot.add_trace(go.Scatter(x = range(N), y = data, name = "data"))
    log_plot.add_trace(go.Scatter(x = range(N), y = np.exp(polynomial(np.log(x))),
name = "powerlaw fit"))

    plot_type = "lin-lin"
    if log == True:
        log_plot.update_layout(xaxis_type="log", yaxis_type="log")
        plot_type = "log-log"

    log_plot.update_layout(
    title='{}individual wealth sorted at t={} on {} with trendline  ({})'.format(fu
nc_fig_no(), t, plot_type, title),
    xaxis_title='nth person (n)',
    yaxis_title='individual wealth (€)')

    return log_plot


#plot the lin-lin and log-log plot for the last investment round
func_plots_with_fit(people_0, -1, log=True).show()
```

Step 4 : Exploring additional aspects

Step 4.1: a coefficient over time

The first aspect we want to explore is the power law over time. The most convenient way is to check the a coefficient of the powerlaw fit over time. Theoretically a powerlaw starts between 2 < a < 3, so instead of visualizing the actual data +fit of different timestamps in one or multiple graphs we just grab a for each t and have a look at that. Gather the a coefficients using functions of step 3.

In [26]:
```
def func_plot_a_vs_t(p_set):
    '''
    plots the a coefficient of each t's polyfit against t to visualize how powerlaw
-y the data is. a=2 consitutes a powerlaw
    '''
    people, title = p_set
    N = len(people)

    a_over_time = []

    for r in range(rounds):
        polycoeffs, error = func_polyfit_t(people, r)
        a_over_time.append(-1 * polycoeffs[0])

    alpha_plot = go.Figure()
    alpha_plot.add_trace(go.Scatter(x = range(r), y = a_over_time))

    alpha_plot.update_layout(
    title='{}a coefficient of the polyfit for t  ({})'.format(func_fig_no(), titl
e),
    xaxis_title='investment round (t)',
    yaxis_title='a coefficient of the trend')

    return alpha_plot



func_plot_a_vs_t(people_0).show()
```

Step 4.2: residuals over time

To verify how accurate our fit is, we also plot the residuals against t.

In [27]:
```
def func_plot_e_vs_t(p_set):
    '''
    plots the residuals of each t's polyfit against t to visualize the acuracy of o
ur fits
    '''
    people, title = p_set
    N = len(people)

    e_over_time = []

    for r in range(rounds):
        polycoeffs, error = func_polyfit_t(people, r)
        e_over_time.append(error[0])

    e_plot = go.Figure()
    e_plot.add_trace(go.Scatter(x = range(r), y = e_over_time))

    e_plot.update_layout(
    title='{}residuals/error of the polyfit for t  ({})'.format(func_fig_no(), titl
e),
    xaxis_title='investment round (t)',
    yaxis_title='resisuals of the trend')

    return e_plot



func_plot_e_vs_t(people_0).show()
```

Step 4.3: Who owns 50% of the wealth

The sorted (lin-lin) plot of any t possesses interesting information, you can read it as, n people posses at least j €, alternatively: n people posses j% of the wealth. However, as in the previous situation we are not interested in a single t. We, instead, want to quantify this data across time. So we settled for a j% of 50 and rand this over time.

The functions calculate how many people (the minimum) posses about 50% of all the wealth for a given t, and plot this agains the range t. You can also choose to print the statement for a single t with a little more info, but we do not use this currently.

In [28]:
```
def func_halfwaycash(people, t, info = False):
    '''
    given a t it returns the nth person so that everyone including them own closest
to half of the wealth at that t
    using info prints the person in text with a bit of explanation
    '''

    N = len(people)

    ls_in = func_sorted_time(people, t)
    half_total_cash = sum(ls_in)/2 #half of the cummulative wealth

    track_cash = 0
    wealthy = 0
    for idx, cash in enumerate(ls_in):
        if (abs(half_total_cash - track_cash) >   #if the new itteration is closer
to half of the money
            abs(half_total_cash - (track_cash+cash))): #than the previous itteratio
n
            track_cash += cash #continue
        else: #if this is not the case, we have found the best halfway split as the
results will not get closer
            if info:
                print("\n\nthere are {} rich that closest split the graph 50/50 lef
t and right with €{} combined".format(idx, round(track_cash,2)))
                print("the nr. {} most wealthy person has €{}".format(idx, round(ls
_in[idx-1],2)))
            return (round(idx/N *100,2))
    return


def func_plot_5050_vs_t(p_set):
    '''
    plots the amount least amount of people owning closest to half the wealth of ea
ch t against t to visualize the acuracy of our fits
    '''

    people, title = p_set
    N = len(people)

    p50_over_time = []

    for r in range(rounds):
        p50_over_time.append(func_halfwaycash(people, r))

    p50_plot = go.Figure()
    p50_plot.add_trace(go.Scatter(x = range(r), y = p50_over_time))

    p50_plot.update_layout(
    title='{}percentage of people that possess closest to half of the cumulative we
alth  ({})'.format(func_fig_no(), title),
    xaxis_title='investments rounds (t)',
    yaxis_title='% of people')

    return p50_plot




#halfwaycash(func_powerlaw_t(10), info = True)
func_plot_5050_vs_t(people_0)
```

Step 4.4: average wealth over time

Originally we expected the average wealth to remain around €100, when looking at the first graph however we felt like this might not be the case at all. To find out we plot the average wealth of t agains the range t.

In [29]:
```
def func_list_average(lst):
    '''returns the average value of a list of values'''

    t_avg = sum(lst) / len(lst)
    return t_avg


def func_plot_avg(p_set):
    '''returns a plot for with a wealth average for each t'''

    people, title = p_set
    rounds = len(people[0])

    market = []
    market_alt = []
    for t in range(rounds):
        market.append(func_list_average(func_sorted_time(people, t)))

    mkt_plot = go.Figure()
    mkt_plot.add_trace(go.Scatter(x = range(rounds), y = market))
    mkt_plot.add_trace(go.Scatter(x = range(rounds), y = market_alt))

    mkt_plot.update_layout(
    title='{}average wealth per round  ({})'.format(func_fig_no(), title),
    xaxis_title='investment round (t)',
    yaxis_title='average wealth')

    return mkt_plot

func_plot_avg(people_0)
```

Section II: Extended Model and Simulation

Now that we can create and visualize/explore data it is time to use larger numbers in our simulations. The slowest graph to plot by far is the *individual wealth vs t graph* of step 1 as it has to process a lot of datapoints for a lot of people, with an increasing N this only gets worse. We believe section I step 1 visualised the core of the data/model wel enough so to save our computers from drawing difficult graphs we will leave *individual wealth vs t graphs* out of it from now on and will only explore features of the data sets.

Graphs that can take more than one set

In order to show multiple sets of data with different starting variables, we need to adapt the graphing functions a little. We do so in the following code cell. The names are the smae except that *plot* is switched to *plots* and the input is a list of the lists we were using before.

In [30]:
```
def func_plots_a_vs_t(p_sets):
    '''multi people set version of func_plot_a_vs_t
    input: set of people sets'''

    alpha_plot = go.Figure()

    for p_set in p_sets:
        people, title = p_set
        N = len(people)

        a_over_time = []

        for r in range(rounds):
            polycoeffs, error = func_polyfit_t(people, r)
            a_over_time.append(-1 * polycoeffs[0])

        alpha_plot.add_trace(go.Scatter(x = range(r), y = a_over_time, name=title))

    alpha_plot.update_layout(
    title='{}a coefficient of the polyfit for t'.format(func_fig_no()),
    xaxis_title='investment round (t)',
    yaxis_title='a coefficient of the trend')

    return alpha_plot


def func_plots_avg(p_sets):
    '''multi people set version of func_plot_avg
    input: set of people sets'''

    mkt_plot = go.Figure()

    for p_set in p_sets:
        people, title = p_set
        rounds = len(people[0])

        market = []
        for t in range(rounds):
            market.append(func_list_average(func_sorted_time(people, t)))

        mkt_plot.add_trace(go.Scatter(x = range(rounds), y = market, name=title))

    mkt_plot.update_layout(
    title='{}average wealth per round'.format(func_fig_no()),
    xaxis_title='investment round (t)',
    yaxis_title='average wealth')

    return mkt_plot


def func_plots_5050_vs_t(p_sets):
    '''multi people set version of func_plot_5050_vs_t
    input: set of people sets'''

    p50_plot = go.Figure()

    for p_set in p_sets:
        people, title = p_set
        N = len(people)

        p50_over_time = []

        for r in range(rounds):
            p50_over_time.append(func_halfwaycash(people, r))
```

More data with different attributes

Next to explore the effects of different starting attributes and environments, we want data sets with way more people to even out the extreme randomness involved in the model. We switch from a 100 people for easy graphing to 10'000 people for more stable results.
Then we create the data sets with new attributes.

p_0 *default state* : Uses the variables of our interpretation of the standard model.

The other data sets change one feature of this "default"

p_1 *smarter % ivnesting* : People tend to invest less of their total money with an exponential distribution and mean of 20% instead of a uniform 50%
p_2 *global tax* : After each round, 1% of every indivuals wealth is taken and distributed over everyone
p_3 *multiplicative loss* : Instead of calculating loss by 1-p we calculate it as 1/(1+p), making the absolute impact of losing and winning equal
p_4 *extreme volatility* : The market p% (profit/loss) fluctuates between 0 and 100 instead of 0 and 40

In [34]:
```
# people_1 = func_investments(rounds, tax = 0.00003) #tax = 0.00003 so 0.003% this
stabilizes the a coeff at 2ish, not fun fact

rounds = 400
N = 10000

p_0 = func_investments(rounds, title="default state",
                                   loss='add', tax=0,
                                   vola_type='uni', vola=0.4,
                                   inve_type='uni', inve=1)

p_1 = func_investments(rounds, title="smarter % investing",
                                   loss='add', tax=0,
                                   vola_type='uni', vola=0.4,
                                   inve_type='exp', inve=0.2)

p_2 = func_investments(rounds, title="global tax",
                                   loss='add', tax=0.01,
                                   vola_type='uni', vola=0.4,
                                   inve_type='uni', inve=1)

p_3 = func_investments(rounds, title="multiplicative loss",
                                   loss='mul', tax=0,
                                   vola_type='uni', vola=0.4,
                                   inve_type='uni', inve=1)

p_4 = func_investments(rounds, title="extreme volatility",
                                   loss='add', tax=0,
                                   vola_type='uni', vola=1,
                                   inve_type='uni', inve=1)

p = [p_0,p_1,p_2,p_3,p_4]
```

Plotting aspects of the different models

And now we plot using in the previously creating function.

In [35]:
```
def func_plot_multi_p_compressed(p):
    '''plots the average, 50/50 & a-coeff graphs with all sets in one graph '''

    func_plots_avg(p).show()
    func_plots_5050_vs_t(p).show()
    func_plots_a_vs_t(p).show()

func_plot_multi_p_compressed(p)
```

Plots with individual sets

Now we visualise the sets again, but thist time every data set get's its own graphs in it's own section. For refference purposes

In [33]:
```
def func_plot_multi_p(p):
    '''plots all graphs for each set of people supplied'''
    for p_x in p:
        print "{}\n".format(p_x[1])
        #func_plot_wealth_vs_t(p_x).show() #PLEASE DONT DO THIS WITH HIGH N
        func_plot_5050_vs_t(p_x).show()
        func_plots_with_fit(p_x, -1, log=True).show()
        func_plot_a_vs_t(p_x).show()
        func_plot_avg(p_x).show()
        print "\n\n"

func_plot_multi_p(p)
```

```
default state
```

```
smarter % investing
```

```
global tax
```

```
multiplicative loss
```

```
extreme volatility
```