Markov Chain Monte Carlo Project

Abstract

Python libraries for random distribution generation and for Markov Chain Monte Carlo algorithm are built. The algorithm is applied to one randomly generated and one supplied dataset. The results and the properties of the algorithm are explored.

1 Problem description

The general problem is defined as creating a universal Markov Chain Monte Carlo algorithm that can fit a given distribution with the specified function – by finding the optimal values of the function parameters. In this particular work, the constructed version of MCMC is tested on the following data sets:

1. Generated points on a $y = Ax + B$ line, with added Gaussian noise to each point along both axes.
2. Supplied data set points, to fit to the $y = A\sin(Bx) + Cx$ line.

2 Methodology

(Full code is available in the Appendix.)

Two Python libraries were built (and thoroughly tested) in order to provide the tools for easy writing of the script solving the stated problems. The libraries fully support multi-dimensionality, including functions with multiple coordinate inputs, multiple parameters and multi-dimensional output. For this particular project, multi-dimensionality is not required, but it can be useful in the future projects.

The first library, **lib_distrib.py**, provides tools for random distribution generation; of particular importance are the functions *draw_rand()* and *gen_distrib()*, generating one or multiple points correspondently for a given distribution density function. They work by building a Monte Carlo box within the given limits and generating random points within the box (with the specified border locations), returning the points which lie "below" the function value.

The secont library, **MCMC.py**, features the MCMC procedure (*MCMC* function). It utilizes Metropolis-Hastings algorithm, which at each step changes each of the guessed parameters by the specified value; in this implementation, the change of each parameter is defined by a roll according to Gaussian distribution with the specified STD. After each step, **likelihood** value is calculated, maximization of logarithm of which corresponds to the minimum squares plus constant:

$$\log(L) = -\sum_i (y_i - f_i)^2 + c,$$

where L is the likelihood, y_i is the data function value for the i-th point, f_i is the guess function value for the i-th point, and c is the constant (discarded in the algorithm). Then comparison between the likelihood value for the new guessed parameter set, L_{new}, and the previous value, L_{old}, is made. If either $L_{new} > L_{old}$, or if $R < e^{L_{new}-L_{old}}$, where R is a random value between 0 and 1 – then the current parameter set changes to the guessed one. Otherwise, the current parameter set remains the same.

The algorithm works in three consequitive modes. The first, "**coarse**" mode, uses the initial value for STD of Gaussian parameter guess changes in order to approach the location of the optimal (i.e. maximum likelihood) parameter values. The second, "**fine**" mode, decreases the STD at each step, in order to close in on the optimal parameter values and narrow down the region they lay in further.

The last, "**final**" mode, uses the resulting decreased STD to sample the parameter values within that narrowed region. At each step in each mode three quantities are saved: the step number, current guessed parameter values, and the likelihood value.

The script file **Run.py** uses these two libraries in order to apply the MCMC algorithm to the problems stated in the previous section. It set the relevant parameters for both parts of the problem, as well as the general reporting and plotting parameters. For each part, initial distribution is generated and plotted, and the MCMC algorithm is performed. The optimal parameter values, acceptance rates in each mode and the parameter value ranges at different confidence levels are calculated. The posterior distributions for individual parameters and cross-distributions for each parameter pair are received. The cumulative distributions with confidence intervals are plotted, and the result of the final fit for the second part is displayed. The further description of the script and the results are provided in the next section.

3 Procedure and results

3.1 Part 1: y=Ax+B

We generate the distribution of 200 (x, y) points for the linear function $y = Ax + B$ with values $A = 7$, $B = -21$ in the range from $x = -20$ to $x = 20$. We add Gaussian noise to each point with STD of 1 along the x axis and 8 along the y axis, with the x-size of both Monte Carlo boxes of (-20,20). The resulting distribution is shown on fig. 1.

Then we perform the MCMC procedure. We use the following parameters: initial guess $(A, B) = (0, 0)$, initial STD $(dA, dB) = (0.01, 0.01)$, the decay factor in the "fine" mode (the reduction of the Gaussian STD at each step relative to the current value) of (0.0002,0.0002), and the number of steps in each mode of $15,000$ ($45,000$ total). It is worth noting that at the N-th step the value of the STD for a given parameter $dP_N = dP_{N-1} \times (1 - dP_{decay})$, so after N_{fine} steps in the fine mode, the new parameter value equals:

$$dP_{final} = dP_{initial} \times (1 - dP_{decay})^{\frac{1}{dP_{decay}} N_{fine} \times dP_{decay}}$$

Knowing the relation $\lim_{x\to\infty} (1 + \frac{1}{x})^x = e$, we can see that for the number of steps large enough, approximately:

$$dP_{final} \approx dP_{initial} \times e^{-N_{fine} \times dP_{decay}},$$

which lets us treat $N_{fine} \times dP_{decay}$ as a scaling factor, describing how much the parameter region is narrowed down by the "fine" mode in the MCMC procedure. In this case this scaling factor is $15,000 \times 0.0002 = 3$, so the region is narrowed down by a factor of $\sim e^3 \approx 20$.

Throughout the run, the MCMC procedure prints out logging information, which helps keep track of the procedure and the time it takes for it to complete, printing lines such as this:

```
628 s: Coarse mode, 14250 out of 15000 steps done.
```

By using this information, we see that the whole $45,000$ steps run takes $1967s$, or approximately 33 minutes. Note that with proper parameter values, the convergence to the values close to the true ones turns out to be possible within ~ 200 steps in the "coarse" mode, which takes approximately $10s$.

Next, the algorithm finds the median for each parameter value in the final mode and prints them out as optimal values. In our case the output is:

```
Optimal linear parameter values: A =  6.92561209477 , B =  −21.3078919281
```

, which is very close to the true values of $(A, B) = (7, -21)$.

Calculation of acceptance rates of each of the procedure modes is done by looping over all the likelihood values and comparing each two consequitive ones for equality. The output is:

```
Acceptance rate for the Coarse mode:  99.99 %.
Acceptance rate for the Fine mode:  99.99 %.
Acceptance rate for the Final mode:  100.0 %.
```

The acceptance rates are so high, because the parameter changes dA, dB are small, and the Metropolis-Hastings condition is almost always satisfied. They are much lower in case of larger parameter changes, such as $(dA, dB) = (0.5, 0.5)$.

The posterior distributions for A and B, and the cross distribution between A and B, for the relevant part of the algorithm (the "final" mode) are shown on fig. 2, fig. 3 and fig. 4. For parameter B, there are two distinct local maxima of likelihood, and from the cross-distribution plot we see that A and B are covariant with each other. It is also clear from the fact that, to fit the static distribution of points, change in one parameter expects change in another parameter to make up for the possible likelihood value decrease.

Finally, we form the cumulative sum graphs (normalized to the maximum value of 1) for each of the posterior distributions. On the cumulative sum graph, we find the point closest to the value of 0.5. Then, for each confidence value V (such as 0.68), we find the points closest to the values of $\frac{1\pm V}{2}$. The confidence interval is the parameter value range between those two points. The script outputs the following intervals for the confidence values of 68% and 95% (note that the exact confidence values are slightly different, since the discretization of the cumulative sum makes it impossible to find the precise value points):

```
Parameter A , confidence level  68.0066666667 %.  A  = [ 6.92107734064 ,
    6.930555961 ].
Parameter A , confidence level  94.9866666667 %.  A  = [ 6.91757461775 ,
    6.93640575291 ].
Parameter B , confidence level  68.0 %.  B  = [ −21.3251462982 ,
   −21.2516698036 ].
Parameter B , confidence level  95.0133333333 %.  B  = [ −21.3513841122 ,
     −21.2298197023 ].
```

These lay outside the actual values of $(A, B) = (7, -21)$, which is expected due to the noise injected in the data. The values of the uncertainties do not seem to be corresponding to the amount of injected noise (comparable ratios of the interval sizes, as opposed to 1/8) and are likely determined by the combination of the amount of injected noise, and the STD of the parameter guess change per step in the "final" mode.

The cumulative distributions and the confidence intervals for A, B are shown on fig. 5 and fig. 6.

3.2 Part 2: y=Asin(Bx)+Cx

We use the data from the file **Data_part_2.py** for (x,y) points. The initial data is shown on fig. 7.

The MCMC procedure is the same as in Part 1 (and again we start at the zero parameter values $(A, B, C) = (0, 0, 0)$); however, in this case, given the nature of the function (sinusoidal), as well as the number of parameters (3 instead of 2), the algorithm performs much slower, and there is a danger of it being stuck in the local minimum given the parameter change per step small enough. This is exactly what occurred on the first long run, consisting of $8,000$ steps per each mode, $24,000$ steps total, and the parameter change STDs $(dA, dB, dC) = (0.02, 0.02, 0.02)$, with the decay factor of 0.0003 for each parameter. Taking as long as $43338s$ ($\approx 12hrs$) to run, the resulting parameter values were

```
Optimal sinusoidal−linear parameter values: A =  1.7825095286 , B =
   0.118261796478 , C =  −0.431355024449
```

, and the resulting fit obviously did not match the expectations (fig. 8), despite a very large number of samples.

To avoid the algorithm getting stuck in the (wrong) local minimum, we increase the initial parameter change STD values to $(dA, dB, dC) = (0.4, 0.4, 0.1)$, take $1,500$ steps per mode ($4,500$ steps total), and select the decay factor of 0.001 for each STD (resulting in the narrowing down of the final parameter region by a factor of $\sim e^{1.5}$). The full run took $8451s$, or roughly $2.3hrs$. Further testing reveals that with certain parameter tweaks, the algorithm can go from $(A, B, C) = (0, 0, 0)$ to roughly the right values (meaning good-looking fit) within a total of ~ 2000 steps, significantly more than for the linear case.

The optimal parameters were found to be:

```
Optimal sinusoidal-linear parameter values: A =  4.21202609644  , B =
    0.880385540266  , C =  -0.29835255097
```

Acceptance rates were significantly lower than for the linear case, mostly because of the relatively high parameter guess change STDs, leading to more rejections, especially during the "fine" mode, where relatively quick change of the parameter STDs led to the point sometimes wandering beyond the decreasing region.

```
Acceptance rate for the Coarse mode:  99.8 %.
Acceptance rate for the Fine mode:  92.13 %.
Acceptance rate for the Final mode:  99.07 %.
```

The posterior distributions for A,B,C, and the posterior cross distributions of each two of them, are shown on fig. 9, fig. 10, fig. 11, fig. 12, fig. 13, fig. 14. Due to the significantly smaller number of steps than in the linear case, the histograms aren't filled densely, but nevertheless demonstrate the algorithm closing up on the optimal solution.

The confidence intervals turn out to be

```
Parameter A , confidence level  66.6666666667 %.  A  = [  3.9911380634  ,
    4.40446057456  ].
Parameter A , confidence level  94.6666666667 %.  A  = [  3.882712237  ,
    4.48495112539  ].
Parameter B , confidence level  69.8 %.  B  = [  0.874465093586  ,
    0.884383962936  ].
Parameter B , confidence level  95.1333333333 %.  B  = [  0.868025886674  ,
      0.885942485428  ].
Parameter C , confidence level  68.2 %.  C  = [  -0.314372485797  ,
    -0.288286422059  ].
Parameter C , confidence level  95.6 %.  C  = [  -0.321567106731  ,
    -0.280231413467  ].
```

, with very significant uncertainties for A. The cumulative distributions and confidence interals are displayed on fig. 15, 16, 17.

Nevertheless, despite these uncertainties, the fit turns out to be very precise, as seen on fig. 18.

4 Appendix: Code

4.1 Run.py

```
from __future__ import division
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
```

```
from scipy.interpolate import griddata
import lib_distrib
import MCMC
import Data_part_2


# General parameters.
N_logging = 25
dL_critical = 0.0000001 # Used when calculating acceptance rate; change
    of likelihood below this is considered null.
grid_density = 100 # How many cells per 1 dimension to use when plotting
    cross-distribution for 2 parameters.
cumul_points = 15000 # How many points the cumulative distribution graph
    to calculate for (essentially its resolution).


# Parameters for part 1.
N_points_1 = 200 # Number of generated data points.
x_min_1, x_max_1 = -20, 20 # The range to generate data points in.
A_1, B_1 = 7, -21 # y = Ax + B
dx_1, dy_1 = 1, 8 # Amount of noise.
dx_factor_1, dy_factor_1 = 20, 20 # The noise will be generated from -
    dxy_factor_1*dxy_1 to dxy_factor_1*dxy_1.
dp_1 = (0.01, 0.01)
dp_decay_1 = (0.0002, 0.0002)
N_coarse_1 = 15000
N_fine_1 = 15000
N_final_1 = 15000


# Parameters for part 2.
dp_2 = (0.4, 0.4, 0.1)
dp_decay_2 = (0.001, 0.001, 0.001)
N_coarse_2 = 1500
N_fine_2 = 1500
N_final_2 = 1500


# Linear function from part 1.
def linear (x, p):
    return p[0]*x + p[1]


# Sinusoidal+linear function from part 2.
def sinusoidal (x, p):
    return p[0]*np.sin(p[1]*x) + p[2]*x


# Plot the noisy linear data generated in part 1.
def plot_initial_data_1 (data_x, data_y):
    plt.clf()
    plt.title("Linear data with noise", size=24)
    plt.xlabel('x', size=24)
    plt.ylabel('y', size=24)
    plt.tick_params(labelsize=18)
    recs = [mpatches.Rectangle((0,0),1,1,fc='blue'), mpatches.Rectangle
        ((0,0),1,1,fc='red')]
    legend_labels = ["Noisy data", "Original line"]
    plt.legend(recs, legend_labels, loc=4, fontsize=18)
    plt.scatter(data_x, data_y, color='blue', s=25, marker='o')
    plt.xlim(min(data_x), max(data_x))
    plt.ylim(min(data_y), max(data_y))
    plt.plot([x_min_1, x_max_1], [A_1*x_min_1+B_1, A_1*x_max_1+B_1],
        color='red', linewidth=3, linestyle='--')
    plt.gcf().set_size_inches(25.6, 14.4)
    plt.gcf().savefig('InitialData_1.png', dpi=100)
    plt.close()


# Plot the data for part 2.
def plot_initial_data_2 (data_x, data_y):
    plt.clf()
    plt.title("Sinusoidal+linear data", size=24)
    plt.xlabel('x', size=24)
    plt.ylabel('y', size=24)
    plt.tick_params(labelsize=18)
    plt.scatter(data_x, data_y, color='blue', s=25, marker='o')
    plt.xlim(min(data_x), max(data_x))
    plt.ylim(min(data_y), max(data_y))
    plt.gcf().set_size_inches(25.6, 14.4)
    plt.gcf().savefig('InitialData_2.png', dpi=100)
    plt.close()


# Calculates acceptance rates for three modes. The arguments are arrays
    of the likelihood values for each mode.
def calc_acceptance_rates (L_coarse, L_fine, L_final):
    L_list = [L_coarse, L_fine, L_final]
    val_list = [[],[],[]] # Lists containing 1 if L changed and 0 if not
        for each step.
    for i in range(0,3):
        L_old = L_list[i][0]
        for L_new in L_list[i]:
            if abs(L_new-L_old)<dL_critical:
                val_list[i].append(0)
            else:
                val_list[i].append(1)
        val_list[i]=val_list[i][1:] # Remove the first element (1) as
            redundant, since the count starts with the 1st step, not 0th.
    return np.array([np.mean(val_list[0]), np.mean(val_list[1]), np.mean(
        val_list[2])])
```

```
# Plots the posterior distribution for all individual and dual
    combinations of the supplied list of arrays of parameter values.
# par_names is the list of the names of the parameters to put the correct
     labels on the plots.
# part: for this project, either "1" or "2"
def plot_distr (data_list, par_names, part):
    data_list = np.array(data_list)
    # Plot individual distributions.
    for data, par_name in zip(data_list, par_names):
        plt.clf()
        if part==1:
            plt.title("y = A*x + B", size=24)
        else:
            plt.title("y = A*sin(B*x) + C*x", size=24)
        plt.xlabel(par_name, size=24)
        plt.ylabel("Occurrence", size=24)
        plt.tick_params(labelsize=18)
        plt.hist(data, bins=50, color='black', fill=False, linewidth=2,
            histtype='step', density=True)
        plt.gcf().set_size_inches(25.6, 14.4)
        plt.gcf().savefig('Parameter_'+str(par_name)+'_'+str(part)+'.png'
            )
        plt.close()
    # Plot dual distributions.
    for i in range (0, len(par_names)):
        for j in range (i+1, len(par_names)):
            plt.clf()
            if part==1:
                plt.title("y = A*x + B", size=24)
            else:
                plt.title("y = A*sin(B*x) + C*x", size=24)
            plt.xlabel(par_names[i], size=24)
            plt.ylabel(par_names[j], size=24)
            plt.tick_params(labelsize=18)
            extent=[min(data_list[i]), max(data_list[i]), min(data_list[j
                ]), max(data_list[j])]
            x_grid, y_grid = np.meshgrid(np.linspace(extent[0],extent[1],
                grid_density+1), np.linspace(extent[2],extent[3],
                grid_density+1))
            # griddata ignores duplicates, so they need to be summed up.
            data_temp_x, data_temp_y, data_temp_N = [], [], []
            for x, y in zip (data_list[i], data_list[j]):
                if x in data_temp_x:
                    data_temp_N[data_temp_x.index(x)]+=1
                else:
                    data_temp_x.append(x)
                    data_temp_y.append(y)
                    data_temp_N.append(1)
            data_temp_x, data_temp_y, data_temp_N = np.array(data_temp_x)
                , np.array(data_temp_y), np.array(data_temp_N)
            # Now, use griddata.
            grid_data = griddata((data_temp_x, data_temp_y), data_temp_N,
                 (x_grid, y_grid), method='cubic', fill_value=0)
            grid_data[grid_data<0] = 0
            plt.imshow(grid_data, origin='lower', cmap=plt.get_cmap('hot'
                ), extent=extent, aspect='auto', interpolation='bilinear')
            cbar = plt.colorbar()
            cbar.ax.tick_params(labelsize=18)
            cbar.set_label('Frequency', size=24)
            plt.gcf().set_size_inches(25.6, 14.4)
            plt.gcf().savefig('Parameters_'+str(par_names[i])+'_'+str(
                par_names[j])+'_'+str(part)+'.png')
            plt.close()


# Calculates cumulative distribution for the parameters, plots it and
    calculates confidence intervals.
# data_list, par_names, part - same as in plot_distr.
# conf_int - set of confidence values, for example (0.68, 0.95, 0.997).
def plot_cumul(data_list, par_names, part, conf_int):
    for data, par_name in zip (data_list, par_names):
        x_points = np.linspace(min(data), max(data), cumul_points+1)
        cumul_data = list(np.array([sum(i <= num for i in data) for num
            in x_points])/len(data))
        # Find the index for closest point to the 50% mark.
        opt_y = min(cumul_data, key=lambda x:abs(x-0.5))
        opt_i = cumul_data.index(opt_y)
        opt_x = x_points[opt_i]
        # Perform the confidence interval search.
        conf_int_x = []
        for i in range (0, len(conf_int)):
            # Find the closest points to the confidence values and their
                indeces.
            bot_y = min(cumul_data, key=lambda x:abs(x-(1-conf_int[i])/2)
                )
            top_y = min(cumul_data, key=lambda x:abs(x-(1+conf_int[i])/2)
                )
            bot_i, top_i = cumul_data.index(bot_y), cumul_data.index(
                top_y)
            # Calculate the parameter values at these points and the
                exact confidence interval.
            left_x = x_points[bot_i]
            right_x = x_points[top_i]
            conf = top_y-bot_y
            conf_int_x.append([left_x, right_x, bot_y, top_y, conf])
        # Plot the distribution and the intervals.
        plt.clf()
        if part==1:
            plt.title("y␣=␣A*x␣+␣B␣-␣cumulative␣distribution", size=24)
        else:
            plt.title("y␣=␣A*sin(B*x)␣+␣C*x␣-␣cumulative␣distribution",
                size=24)
        plt.xlabel(par_name, size=24)
        plt.ylabel("Cumulative fraction", size=24)
        plt.xlim(min(x_points), max(x_points))
        plt.ylim(0, 1)
        plt.tick_params(labelsize=18)
        plt.plot(x_points, cumul_data, color='blue', linewidth=3)
        plt.plot([opt_x, opt_x], [0, 1], color='red', linewidth=2)
        for i in range (0, len(conf_int)):
            plt.plot([conf_int_x[i][0], conf_int_x[i][0]], [0, 1], color=
                'black', linewidth=1, linestyle='--')
            plt.plot([conf_int_x[i][1], conf_int_x[i][1]], [0, 1], color=
                'black', linewidth=1, linestyle='--')
            plt.plot([x_points[0], x_points[-1]], [conf_int_x[i][2],
                conf_int_x[i][2]], color='black', linewidth=1, linestyle='
                --')
            plt.plot([x_points[0], x_points[-1]], [conf_int_x[i][3],
                conf_int_x[i][3]], color='black', linewidth=1, linestyle='
                --')
        plt.gcf().set_size_inches(25.6, 14.4)
        plt.gcf().savefig('Cumulative_'+str(par_name)+'_'+str(part)+'.png
            ')
        plt.close()
        # Print confidence interval data.
        for i in range (0, len(conf_int)):
            print "Parameter", par_name, ", confidence level ",
                conf_int_x[i][4]*100, "%. ", par_name, " = [", conf_int_x[
                i][0], ", ", conf_int_x[i][1], "]."


# Plots the data set and the function predicted by the MCMC procedure.
def plot_part_2_result (data_x, data_y, func, p):
    plt.clf()
    plt.title("Sinusoidal+linear fit", size=24)
    plt.xlabel('x', size=24)
    plt.ylabel('y', size=24)
    plt.tick_params(labelsize=18)
    plt.scatter(data_x, data_y, color='blue', s=25, marker='o')
    plt.xlim(min(data_x), max(data_x))
    plt.ylim(min(data_y), max(data_y))
    # Generate the fit line.
    x_grid = np.linspace(min(data_x), max(data_x), 5000+1)
    y = func (x_grid, p)
    # Plot the fit line.
    plt.plot(x_grid, y, color='red', linewidth=3)
    plt.gcf().set_size_inches(25.6, 14.4)
    plt.gcf().savefig('Result_2.png', dpi=100)
    plt.close()


def PART_1():
    # Generate clean data.
    data_x_1 = np.random.uniform(x_min_1, x_max_1, N_points_1)
    data_y_1 = A_1*data_x_1 + B_1
    # Add noise.
    data_x_1 += lib_distrib.gen_distrib (lib_distrib.Gauss, (0, dx_1), -
        dx_factor_1*dx_1, dx_factor_1*dx_1, 0, lib_distrib.Gauss(0, (0,
        dx_1)), N_points_1)
    data_y_1 += lib_distrib.gen_distrib (lib_distrib.Gauss, (0, dy_1), -
        dy_factor_1*dy_1, dy_factor_1*dy_1, 0, lib_distrib.Gauss(0, (0,
        dy_1)), N_points_1)
    # Plot the generated data.
    plot_initial_data_1 (data_x_1, data_y_1)
    # Perform the MCMC procedure.
    data_1 = MCMC.MCMC (linear, (0,0), data_x_1, data_y_1, dp_1,
        dp_decay_1, N_coarse_1, N_fine_1, N_final_1, N_logging)
    # Retrieve the relevant data (the last N_final_1 batches):
    data_final_1 = data_1[N_coarse_1+N_fine_1+1:N_coarse_1+N_fine_1+
        N_final_1+1]
    # Calculate and output the optimal A, B values:
    A_1_opt, B_1_opt = np.median([d[1][0] for d in data_final_1]), np.
        median([d[1][1] for d in data_final_1])
    print "Optimal linear parameter values: A = ", A_1_opt, ", B = ",
        B_1_opt
    # Calculate the acceptance rates for all 3 modes. Print them out.
    acc_rates = calc_acceptance_rates ([d[2] for d in data_1[1:N_coarse_1
        +1]], [d[2] for d in data_1[N_coarse_1+1:N_coarse_1+N_fine_1+1]],
        [d[2] for d in data_final_1])
    print "Acceptance rate for the Coarse mode: ", round(acc_rates
        [0]*100, 2), "%."
    print "Acceptance rate for the Fine mode: ", round(acc_rates[1]*100,
        2), "%."
    print "Acceptance rate for the Final mode: ", round(acc_rates[2]*100,
         2), "%."
    # Plot the posterior distribution for A, B and both. Only for the
        Final mode.
    plot_distr ([[d[1][0] for d in data_final_1], [d[1][1] for d in
        data_final_1]], ['A','B'], 1)
    # Plot the cumulative sum graph and calculate 68% and 95% confidence
        intervals.
    plot_cumul ([[d[1][0] for d in data_final_1],[d[1][1] for d in
        data_final_1]], ['A','B'], 1, [0.68, 0.95])


def PART_2():
    # Retrieve the data set.
    data_x_2 = Data_part_2.x
    data_y_2 = Data_part_2.y
    # Plot the retrieved data.
    plot_initial_data_2 (data_x_2, data_y_2)
    # Perform the MCMC procedure.
    data_2 = MCMC.MCMC (sinusoidal, (0,0,0), data_x_2, data_y_2, dp_2,
        dp_decay_2, N_coarse_2, N_fine_2, N_final_2, N_logging)
    # Retrieve the relevant data (the last N_final_1 batches):
    data_final_2 = data_2[N_coarse_2+N_fine_2+1:N_coarse_2+N_fine_2+
        N_final_2+1]
    # Calculate and output the optimal A, B, C values:
    A_2_opt, B_2_opt, C_2_opt = np.median([d[1][0] for d in data_final_2
        ]), np.median([d[1][1] for d in data_final_2]), np.median([d[1][2]
         for d in data_final_2])
    print "Optimal sinusoidal-linear parameter values: A = ", A_2_opt, ",
        B = ", B_2_opt, ", C = ", C_2_opt
    # Calculate the acceptance rates for all 3 modes. Print them out.
    acc_rates = calc_acceptance_rates ([d[2] for d in data_2[1:N_coarse_2
        +1]], [d[2] for d in data_2[N_coarse_2+1:N_coarse_2+N_fine_2+1]],
        [d[2] for d in data_final_2])
    print "Acceptance rate for the Coarse mode: ", round(acc_rates
        [0]*100, 2), "%."
    print "Acceptance rate for the Fine mode: ", round(acc_rates[1]*100,
        2), "%."
    print "Acceptance rate for the Final mode: ", round(acc_rates[2]*100,
         2), "%."
    # Plot the posterior distribution for A, B and both. Only for the
        Final mode.
    plot_distr ([[d[1][0] for d in data_final_2], [d[1][1] for d in
        data_final_2], [d[1][2] for d in data_final_2]], ['A','B','C'], 2)
    # Plot the cumulative sum graph and calculate 68% and 95% confidence
        intervals.
    plot_cumul ([[d[1][0] for d in data_final_2], [d[1][1] for d in
        data_final_2], [d[1][2] for d in data_final_2]], ['A','B','C'], 2,
         [0.68, 0.95])
    # Plot the predicted function over the initial dataset.
    plot_part_2_result (data_x_2, data_y_2, sinusoidal, (A_2_opt, B_2_opt
        , C_2_opt))


if __name__ == '__main__':
    PART_1()
    PART_2()
```

4.2 lib_distrib.py

```
from __future__ import division
import numpy as np
import random


# AUTHOR: Dmitrii Gudin, University of Notre Dame, dgudin@nd.edu


# DESCRIPTION:
# ******************************************
# The module provides a set of tools for quick and easy generation of
    distributions characterized by probability density function. It
    supports multidimensionality, including functions with multiple
    arguments, multiple parameters and multi-dimensional function values.
    A typical multidimensional function will look like this:
#     def func (x, p):
#         x, p = np.array([x]).flatten(), np.array([p]).flatten()
#         {some code calculating the function value f}
#         return f
# , where x is the input coordinate vector (single number, list, tuple or
     numpy array), p is the set of parameters (single number, list, tuple
    or numpy array), and f is the function value (single number, list,
    tuple or numpy array). The exact type does not matter, as in the
    function body all inputs are first converted into numpy arrays, so
    they become iterable.
# *********************************************


# Multidimensional Gaussian density distribution function.
# ---------
# [float vector] x: coordinate.
# [float vector] p: list of parameters in the following order: (mu_1,
    sigma_1, mu_2, sigma_2, ..., mu_n, sigma_n) for coordinate elements
    x_1, x_2, ..., x_n.
# ---------
# Output:
# [float] G: the Gaussian function value.
# ---------
# Examples of usage:
#     >>> import lib_distrib as l
#     >>> l.Gauss (2, (0, 1))
#     0.05399096651318063
#     >>> l.Gauss ((1, 3), (0, 1, 2, 5))
#     0.01892417679583174 5
def Gauss (x, p):
    x, p = np.array([x]).flatten(), np.array([p]).flatten()
    # Define the 1-dimensional Gaussian function.
    def Gauss_1d (x_1d, mu_1d, sigma_1d):
        return 1/sigma_1d/np.sqrt(2*np.pi)*np.exp(-(x_1d-mu_1d)**2/2/
            sigma_1d**2)
    # Multiply all 1-dimensional Gaussian values to obtain the
        multidimensional Gaussian value.
    G = np.prod([Gauss_1d(x_1d, mu_1d, sigma_1d) for x_1d, mu_1d,
        sigma_1d in zip (x, p[::2], p[1::2])])
    # Return the result as a number.
    return G


# Generates a point for a given distribution and returns its coordinate.
# ---------
# [float vector function] func: density distribution function name. The
```

```
    function should be of the type func (x, p) as explained above.
# [float vector] p: list of function parameters.
# [float vector] left_lim: list of minimum coordinate components of the
    box for Monte-Carlo sampling.
# [float vector] right_lim: list of maximum coordinate components of the
    box for Monte-Carlo sampling.
# [float vector] low_lim: list of minimum function value components of
    the box for Monte-Carlo sampling.
# [float vector] up_lim: list of maximum function value components of the
     box for Monte-Carlo sampling.
#———
# Output:
# [numpy array] or [float] x: the point coordinate vector.
#———
# Examples of usage:
#     >>> import lib_distrib as l
#     >>> l.draw_rand (l.Gauss, (0,3), -20, 20, 0, (l.Gauss(0,(0,3))))
#     3.0757019662666885
#     >>> p = [0,3,2,5]
#     >>> l.draw_rand (l.Gauss, p, (-50,-50), (50,50), (0,0), (l.Gauss(0,
    p[:2]), l.Gauss(0,p[2:])))
#      array([ 4.78994773, -3.22910379])
def draw_rand (func, p, left_lim, right_lim, low_lim, up_lim):
    p = np.array([p]).flatten()
    left_lim = np.array([left_lim]).flatten()
    right_lim = np.array([right_lim]).flatten()
    low_lim = np.array([low_lim]).flatten()
    up_lim = np.array([up_lim]).flatten()
    random.seed()
    # Do a loop of rolls, until one satisfies the Monte-Carlo conditions.
    while(True):
        # Generate a random coordinate vector.
        Roll = np.array([random.uniform(l_l,r_l) for l_l, r_l in zip(
            left_lim, right_lim)]).flatten()
        # Generate a random function-space vector.
        Roll_value = np.array([random.uniform(l_l, u_l) for l_l, u_l in
            zip(low_lim, up_lim)]).flatten()
        # The roll is only accepted if all the elements of the function-
            space vector are less than those of the corresponding function
             values with the coordinate vector as an argument.
        for i in range (0, len(Roll_value)):
            if Roll_value[i]>np.array([func(Roll,p)]).flatten()[i]:
                break
            if len(Roll)==1: return Roll[0]
            return Roll


# Generates a list of coordinates for a given distribution.
#——
# [float vector function] func: density distribution function name. The
    function should be of the type func (x, p) as explained above.
```

```
# [float vector] p: list of function parameters.
# [float vector] left_lim: list of minimum coordinate components of the
    box for Monte-Carlo sampling.
# [float vector] right_lim: list of maximum coordinate components of the
    box for Monte-Carlo sampling.
# [float vector] low_lim: list of minimum function value components of
    the box for Monte-Carlo sampling.
# [float vector] up_lim: list of maximum function value components of the
     box for Monte-Carlo sampling.
# [int] N: number of points to generate.
# ------
# Output:
# [list of [numpy array]] output_list: list of the generated coordinate
    vectors.
# ------
# Example of usage:
#     >>> import lib_distrib as l
#     >>> p = [0, 3, 1, 5]
#     >>> l.gen_distrib (l.Gauss, (0,3), -20, 20, 0, (l.Gauss(0,(0,3)))
    ,7)
#      [5.8502994077335586, 1.6415874772056007, -8.2615979908369717,
    0.42729648568183265, 4.0388538740797451, -0.71863002513151386,
    0.37606182298244661]
#     >>> l.gen_distrib (l.Gauss, p, (-50,-50), (50,50), (0,0), (l.Gauss
    (0,p[:2]), l.Gauss(0,p[2:])), 5)
#      [array([-2.22807008,  1.20392057]), array([-5.68009949,
    5.41191758]), array([-0.29557321, -8.68410193]), array([-2.94120285,
    9.88142477]), array([-0.46809802, -6.34584931])]
def gen_distrib (func, p, left_lim, right_lim, low_lim, up_lim, N=1000):
    p = np.array([p]).flatten()
    left_lim = np.array([left_lim]).flatten()
    right_lim = np.array([right_lim]).flatten()
    low_lim = np.array([low_lim]).flatten()
    up_lim = np.array([up_lim]).flatten()
    output_list = []
    for i in range(0, N):
        # Use the point generation function N times and form a list out
            of results.
        output_list.append(draw_rand(func, p, left_lim, right_lim,
            low_lim, up_lim))
    return output_list
```

4.3 MCMC.py

```
from __future__ import division
import numpy as np
import lib_distrib
import random
import time
```

```
# AUTHOR: Dmitrii Gudin, University of Notre Dame, dgudin@nd.edu


# DESCRIPTION:
# ********************************************
# The module provides a Markov chain Monte-Carlo tool for estimating fit
    function parameters for multivariate multidimensional functions.
# ********************************************


# NOTES:
# ********************************************
# See the notes in lib_distrib.py, explaining the details of
    implementation of multidimensionality. They explain what shape a
    supplied function must have and provide terminology convention.
# ********************************************


# Calculates the likelihood for the hypothesized function with regards to
     the data. Maximum likelihood corresponds to the least squares for
    Gaussian distribution.
# ---
# [list of [float vector]] guess_f: list of values of the hypothesized
    function over a set of data coordinate points.
# [list of [float vector]] data_f: list of data function values over the
    set of the same coordinate points.
# ---
# Output:
#  [float] The likelihood value.
# ---
# Example of usage:
#      >>> import MCMC as m
#      >>> x = [1, 2, 3, 4, 5]
#      >>> data_f = [2, 4, 6, 8, 10]
#      >>> def guess_func (t):
#      ...      return 1.9*np.array(t) + 0.3
#      >>> guess_f = guess_func(x)
#      >>> print m.L (guess_f, data_f)
#      -0.1
#      >>> x = [(1,11), (2,22), (3,33), (4,44), (5,55)]
#      >>> data_f = [(2,2,1), (4,4,2), (6,6,3), (8,8,4), (10,10,5)]
#      >>> guess_f = guess_func(x)
#      >>> print m.L (guess_f, data_f)
#      -21043.4
def L (guess_f, data_f):
    guess_f = np.array([guess_f]).flatten()
    data_f = np.array([data_f]).flatten()
    return -sum([np.linalg.norm(g_f - d_f)**2 for g_f, d_f in zip(guess_f
        , data_f)])
```

```
# Performs MCMC approximation of parameters of the supplied fit function.
    Works in three sequential modes:
# 1. "Coarse" mode: large range of parameter change per step. The change
   is calculated from a gaussian distribution with the specified STD.
# 2. "Fine" mode: the range of parameter change per step decays over time
   .
# 3. "Final" mode: the final range of parameter change per step from "
   Fine" mode is used.
# ———
# [float vector function] func: fit function name (should be defined in
   advance). The function should be of the type func (x, p) as explained
   in lib_distrib.py.
# NOTE: if you want only some of the parameters varied and others fixed,
   then redefine the function before supplying to the algorithm, like so:
#     def func_new (x, p_var):
#         return func (x, (p[0]=1, p[1]=5, p_var[2:]*))
# [float vector] p_init: initial guess for the function parameters.
# [list of [float vector]] data_x: data coordinate vectors.
# [list of [float vector]] data_f: data function value vectors.
# [float vector] dp: vector of STDs for parameter changes in the "Coarse"
    mode.
# [float vector] dp_decay: vector of parameter change decay factors,
   specifying by what fraction each of the dp elements decreases per step
    in the "Fine" mode.
# [int] N_coarse: Number of steps in the "Coarse" mode.
# [int] N_fine: Number of steps in the "Fine" mode.
# [int] N_final: Number of steps in the "Fine" mode.
# [int] N_logging: How often (in terms of number of steps) to print the
   log messages. Non-positive means no logging.
# ———
# Output:
# [list of [list]] output_list: list of lists (one for each algorithm
   step) consisting of the following elements (in order):
#    - [int] output_step_num: step number.
#    - [tuple] output_p: list of parameter values guessed.
#    - [float] output_L: value of the likelihood function

def MCMC (func, p_init, data_x, data_f, dp, dp_decay, N_coarse=2000,
   N_fine=500, N_final=2000, N_logging=0):
    p_init = np.array([p_init]).flatten()
    data_x = [np.array([d]).flatten() for d in data_x]
    data_f = [np.array([d]).flatten() for d in data_f]
    dp = np.array([dp]).flatten()
    dp_decay = np.array([dp_decay]).flatten()
    # Start time count for logging.
    time_begin = time.time()
    # Calculate the initial likelihood value and initialize the output
        list.
    L_old = L_guess = L([func(x,p_init) for x in data_x], data_f)
    p_old = p_guess = p_init
    if len(p_init)==1:
        output_list = [[0, p_old[0], L_old]]
    else:
        output_list = [[0, p_old, L_old]]

    # "Coarse" mode.
    for i in range (1, N_coarse+1):
        # Guess the parameters by Gaussian displacement of the previous
            set.
        dp_gauss_param = (np.array([[0,dp_i] for dp_i in dp]).flatten())
        p_guess = p_old + lib_distrib.draw_rand (lib_distrib.Gauss,
            dp_gauss_param, -50*np.array(dp), 50*np.array(dp), [0]*len(dp)
            , lib_distrib.Gauss([0]*len(dp),dp_gauss_param))
        # Calculate the new likelihood.
        L_guess = L([func(x,p_guess) for x in data_x], data_f)
        # Perform Metropolis-Hastings check to decide whether to accept
            the new parameter values.
        random.seed()
        if L_guess > L_old or random.uniform(0,1) < np.exp(L_guess-L_old)
            :
            p_old = p_guess
            L_old = L_guess
        # Record the data.
        if len(p_old)==1:
            output_list.append([i, p_old[0], L_old])
        else:
            output_list.append([i, p_old, L_old])
        # Log message.
        if N_logging>0:
            if i%N_logging==0:
                print str(int(time.time()-time_begin)), "s: Coarse mode,"
                    , str(i), "out of", str(N_coarse), "steps done."

    # "Fine" mode.
    for i in range (N_coarse+1, N_coarse+N_fine+1):
        # Calculate the new parameter change.
        dp = np.multiply(dp,1-dp_decay)
        # Guess the parameters by Gaussian displacement of the previous
            set.
        dp_gauss_param = (np.array([[0,dp_i] for dp_i in dp]).flatten())
        p_guess = p_old + lib_distrib.draw_rand (lib_distrib.Gauss,
            dp_gauss_param, -50*np.array(dp), 50*np.array(dp), [0]*len(dp)
            , lib_distrib.Gauss([0]*len(dp),dp_gauss_param))
        # Calculate the new likelihood.
        L_guess = L([func(x,p_guess) for x in data_x], data_f)
        # Perform Metropolis-Hastings check to decide whether to accept
            the new parameter values.
        random.seed()
        if L_guess > L_old or random.uniform(0,1) < np.exp(L_guess-L_old)
            :
            p_old = p_guess
            L_old = L_guess
        # Record the data.
        if len(p_old)==1:
            output_list.append([i, p_old[0], L_old])
        else:
            output_list.append([i, p_old, L_old])
        # Log message.
        if N_logging>0:
            if (i-N_coarse)%N_logging==0:
                print str(int(time.time()-time_begin)), "s: Fine mode,",
                    str(i-N_coarse), "out of", str(N_fine), "steps done."

    # "Final" mode.
    for i in range (N_coarse+N_fine+1, N_coarse+N_fine+N_final+1):
        # Guess the parameters by Gaussian displacement of the previous
            set.
        dp_gauss_param = (np.array([[0,dp_i] for dp_i in dp]).flatten())
        p_guess = p_old + lib_distrib.draw_rand (lib_distrib.Gauss,
            dp_gauss_param, -50*np.array(dp), 50*np.array(dp), [0]*len(dp)
            , lib_distrib.Gauss([0]*len(dp),dp_gauss_param))
        # Calculate the new likelihood.
        L_guess = L([func(x,p_guess) for x in data_x], data_f)
        # Perform Metropolis-Hastings check to decide whether to accept
            the new parameter values.
        random.seed()
        if L_guess > L_old or random.uniform(0,1) < np.exp(L_guess-L_old)
            :
            p_old = p_guess
            L_old = L_guess
        # Record the data.
        if len(p_old)==1:
            output_list.append([i, p_old[0], L_old])
        else:
            output_list.append([i, p_old, L_old])
        # Log message.
        if N_logging>0:
            if (i-N_coarse-N_fine)%N_logging==0:
                print str(int(time.time()-time_begin)), "s: Final mode,",
                     str(i-N_coarse-N_fine), "out of", str(N_final), "
                    steps done."

    # Return the result as list of lists.
    return output_list
```

4.4 Data_part_2.py

```
from __future__ import division
import numpy as np

x = np.array
```

```
    ([1.0815,2.0906,3.0127,4.0913,5.0632,6.0098,7.0278,8.0547,9.0958,10.0965,

     11.0158,12.0971,13.0957,14.0485,15.0800,16.0142,17.0422,18.0916,19.0792,20.0959])


y = np.array
    ([3.0951,3.4227,1.0783,−3.0820,−5.5794,−5.3297,−2.5325,0.6026,1.4259,−0.8572,

     −4.4055,−7.5680,−7.5651,−5.0845,−1.8224,−0.6096,−2.3531,−6.2875,−9.4180,−9.8996])
```

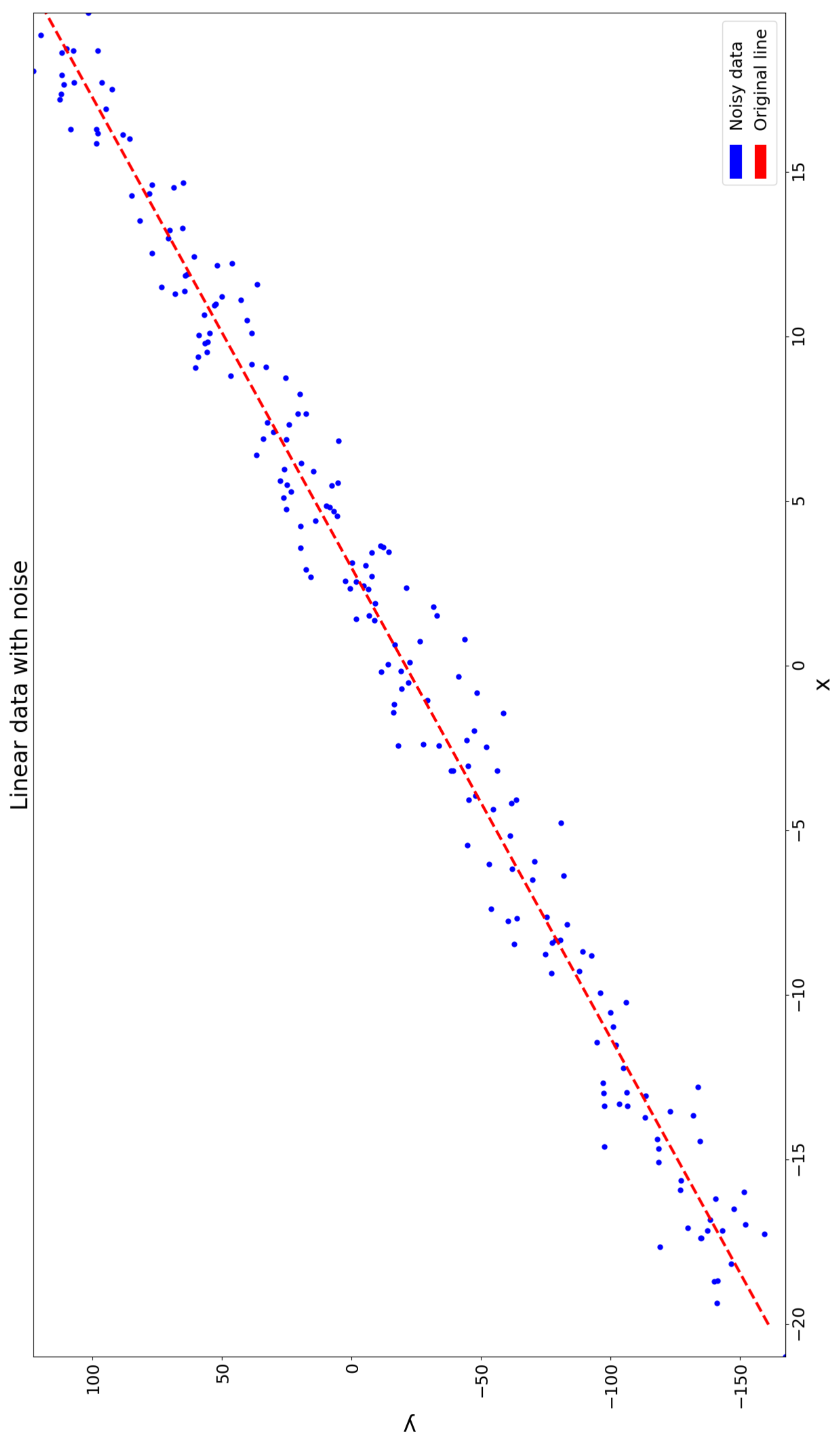

Figure 1: Initial distribution for the noisy linear data.

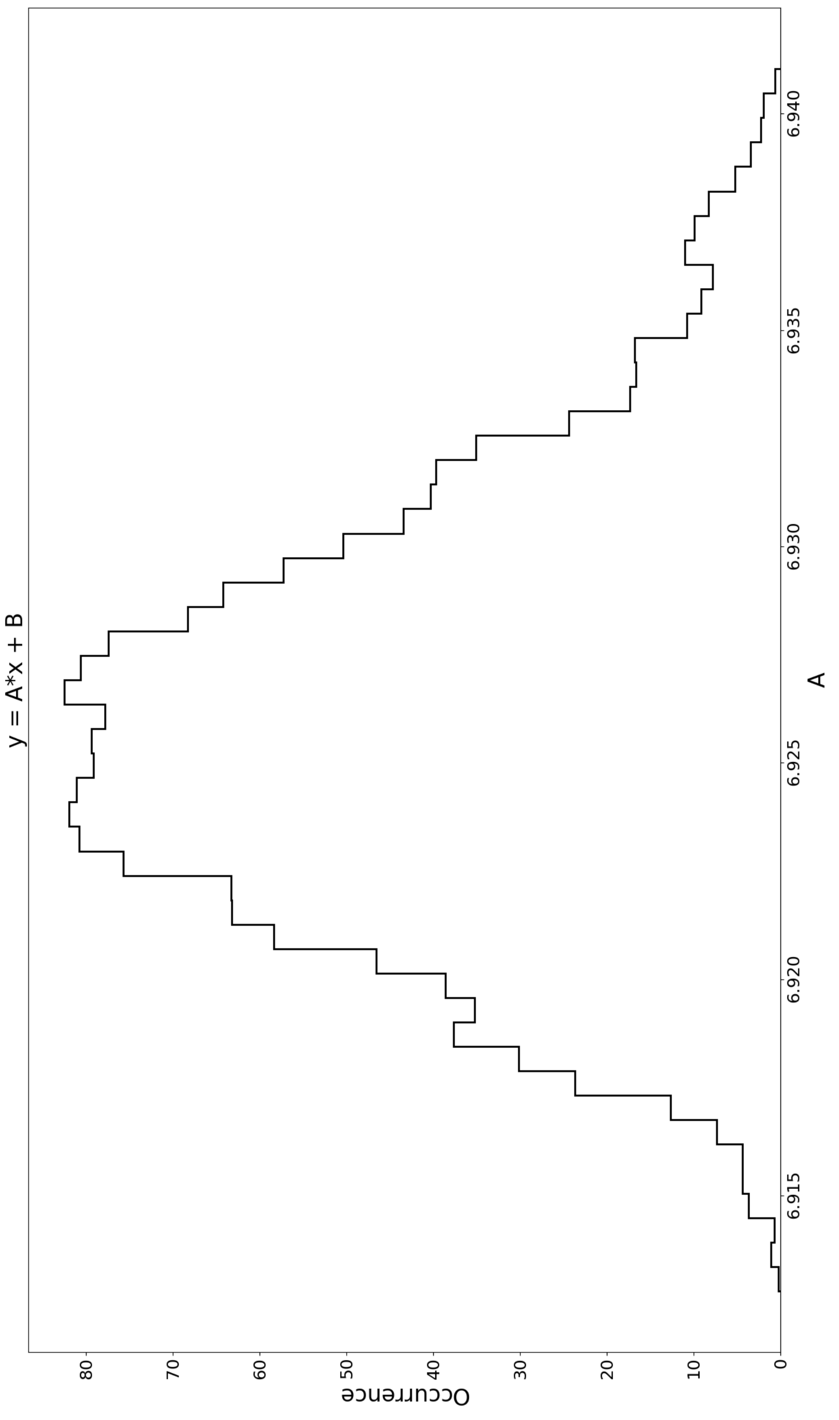

Figure 2: Posterior distribution for A for the linear fit.

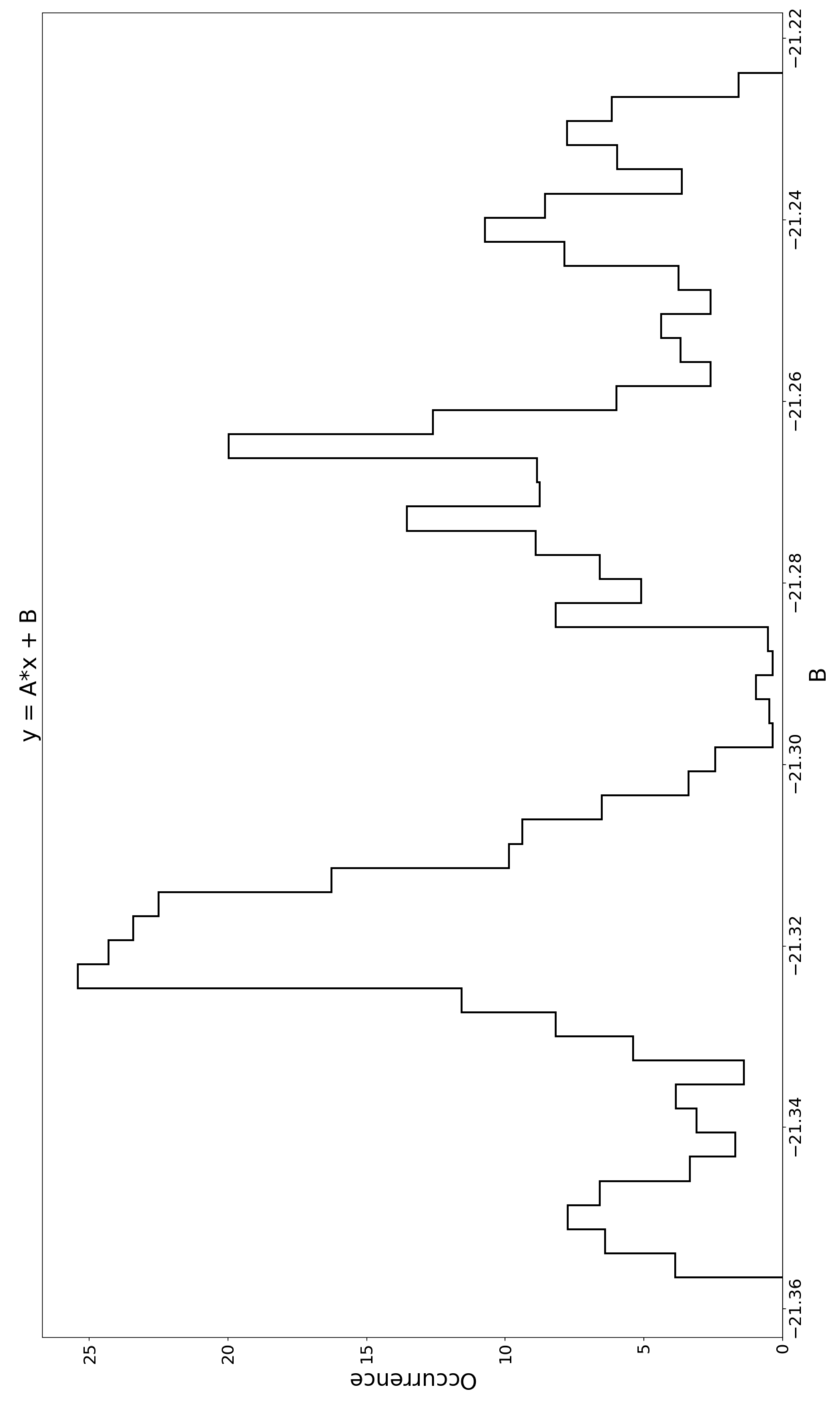

Figure 3: Posterior distribution for B for the linear fit.

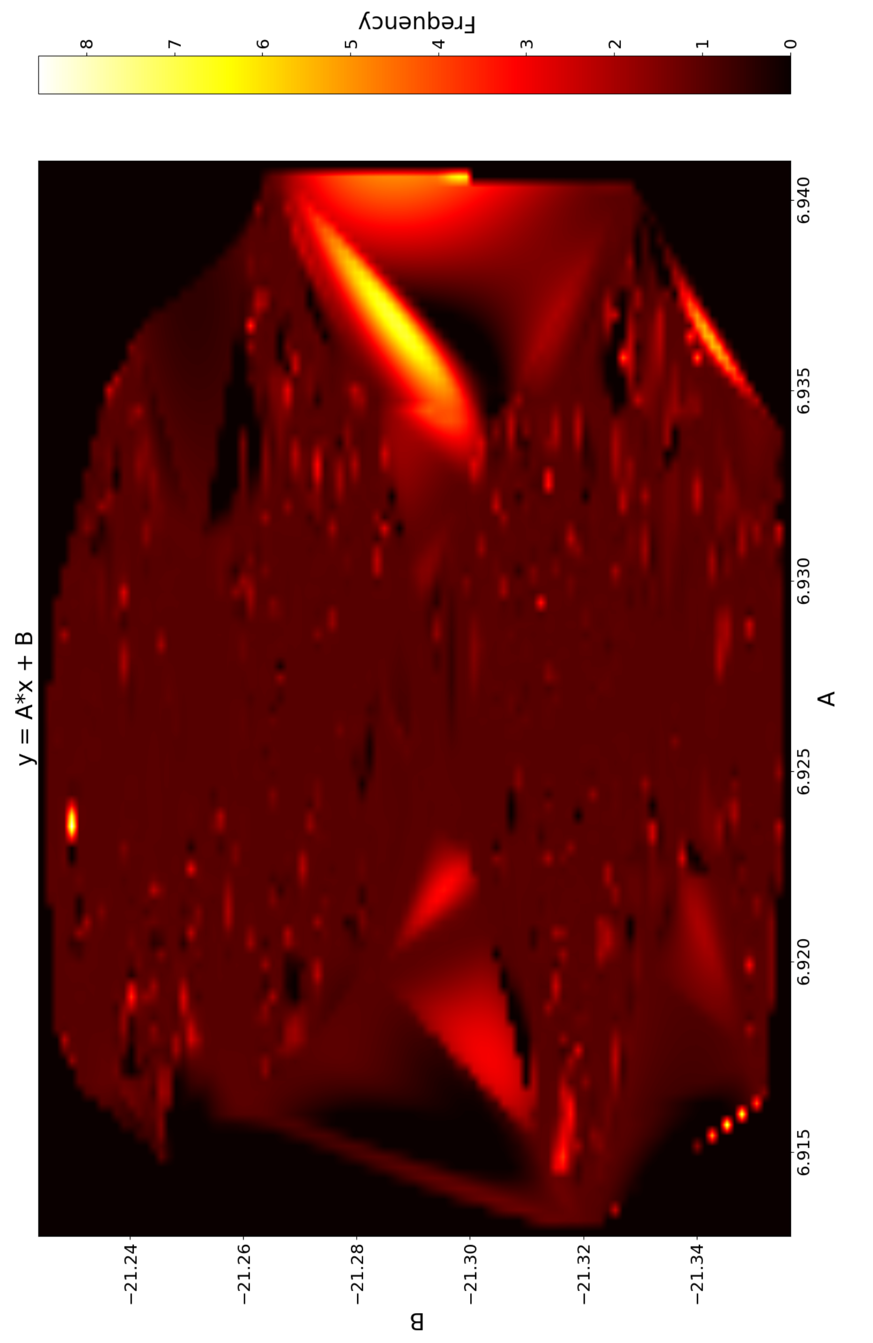

Figure 4: Posterior cross-distribution for A and B for the linear fit.

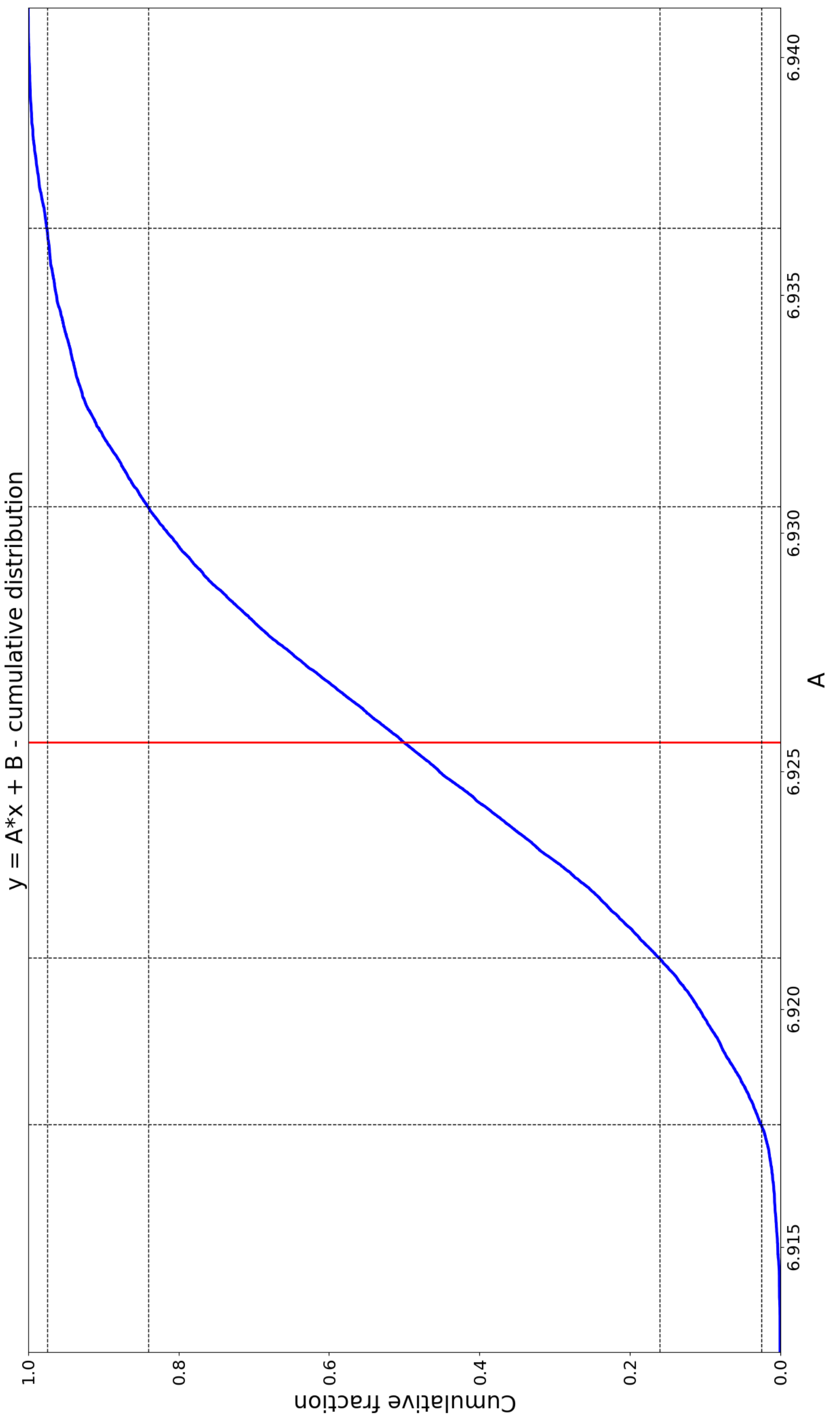

Figure 5: Cumulative distribution and 68%, 95% confidence intervals for A.

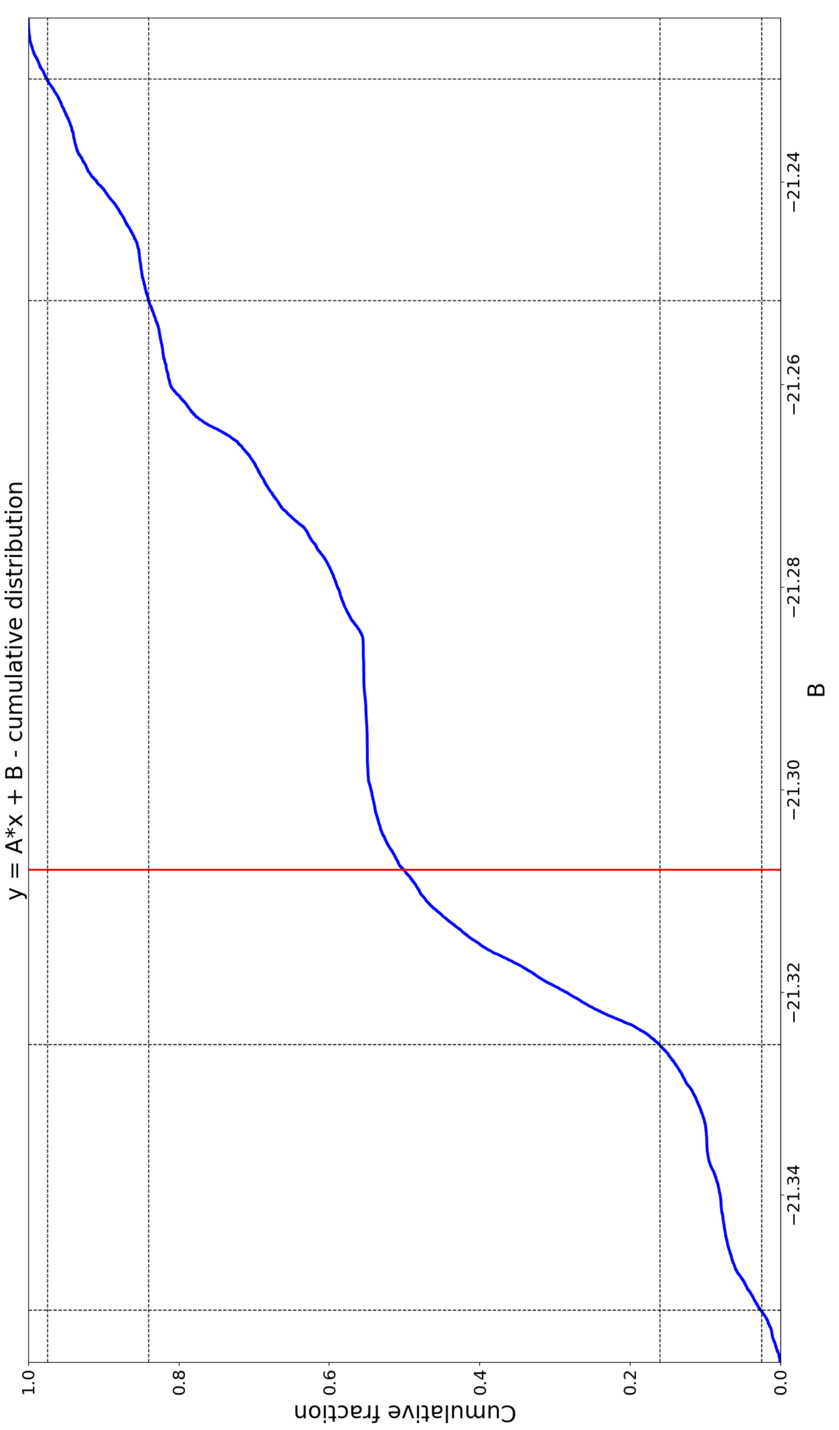

Figure 6: Cumulative distribution and 68%, 95% confidence intervals for B.

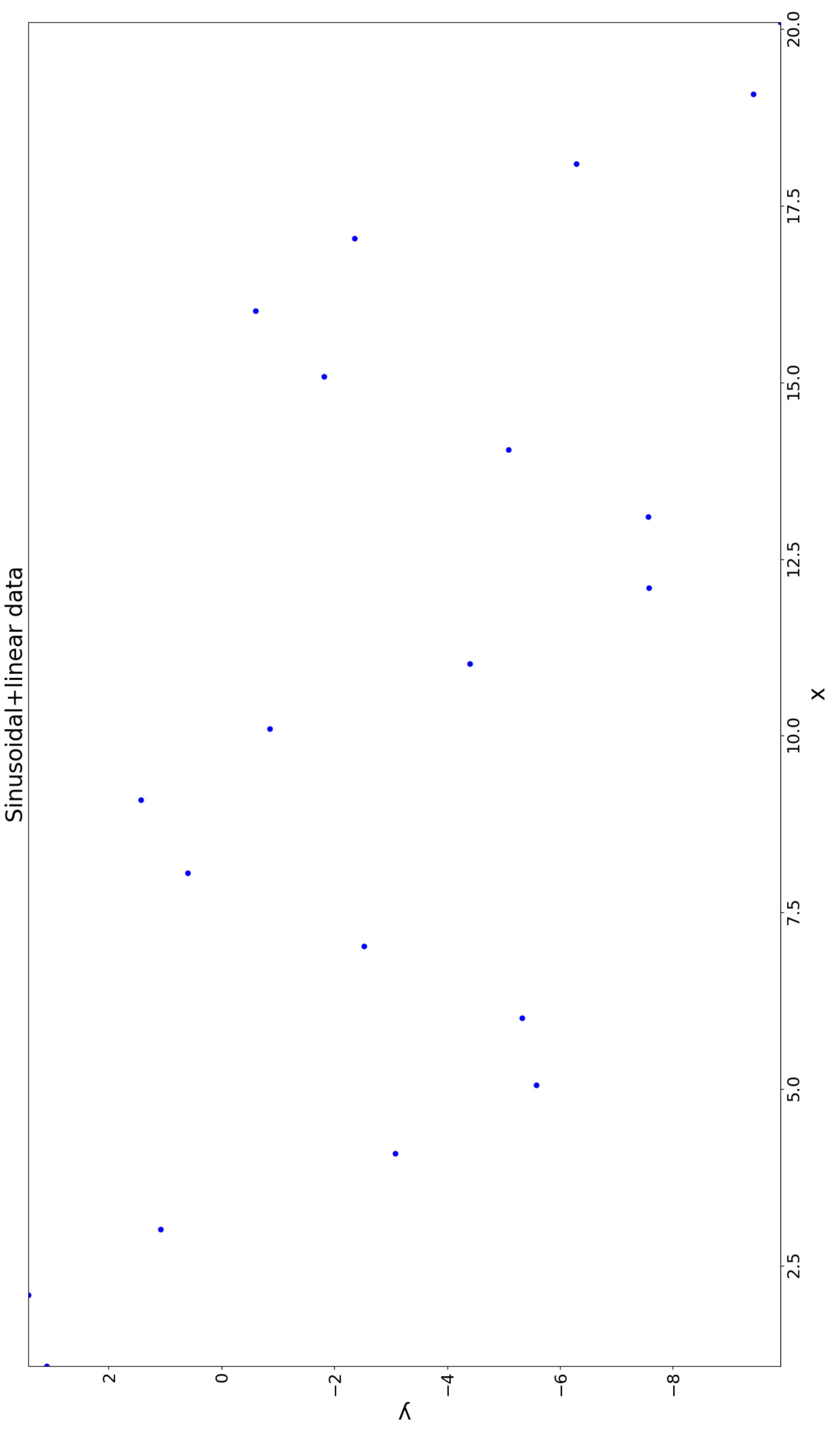

Figure 7: Initial data for the sinusoidal-linear function fit.

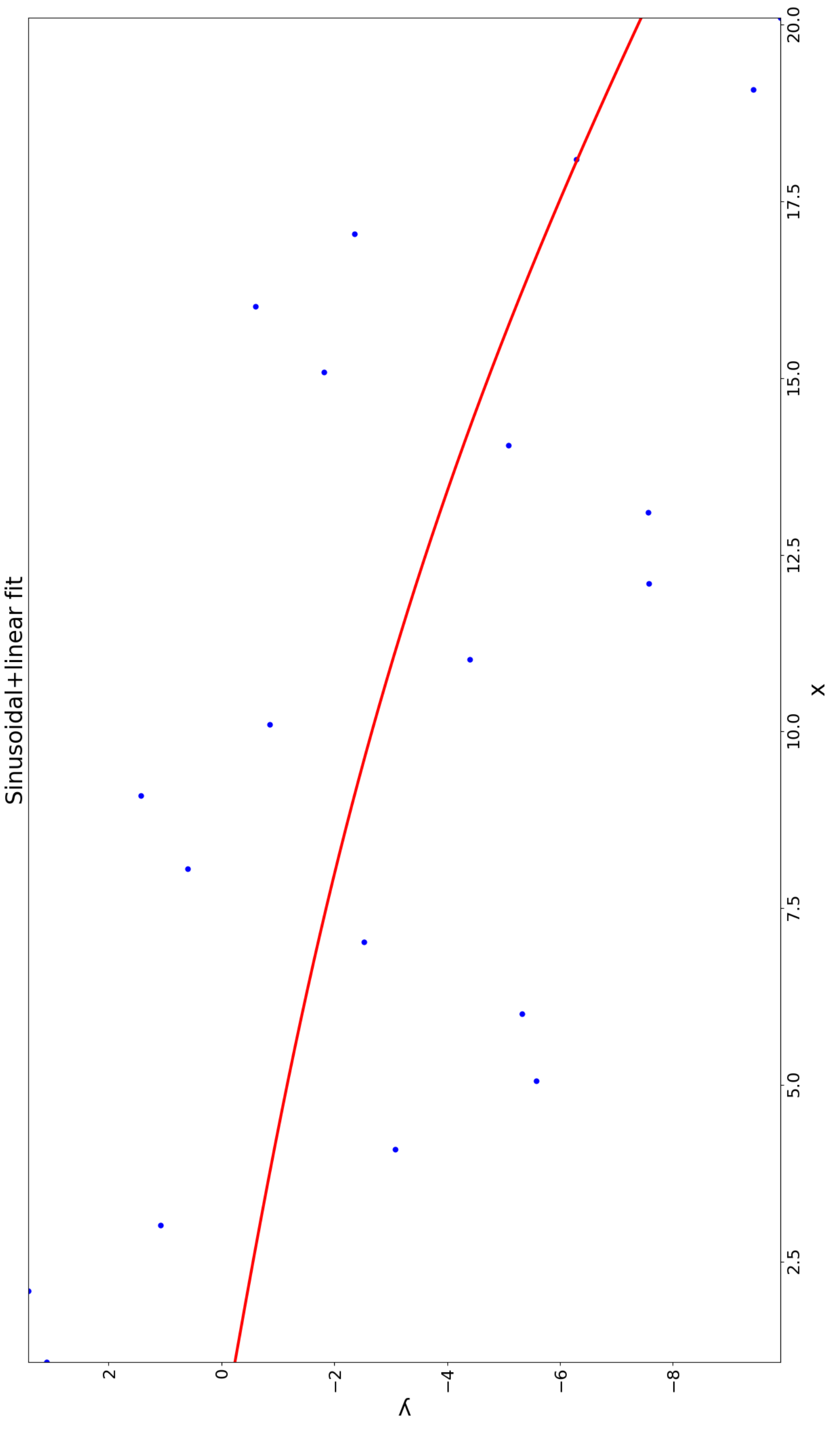

Figure 8: First attempt at fitting the function; local, instead of global, minimum found.

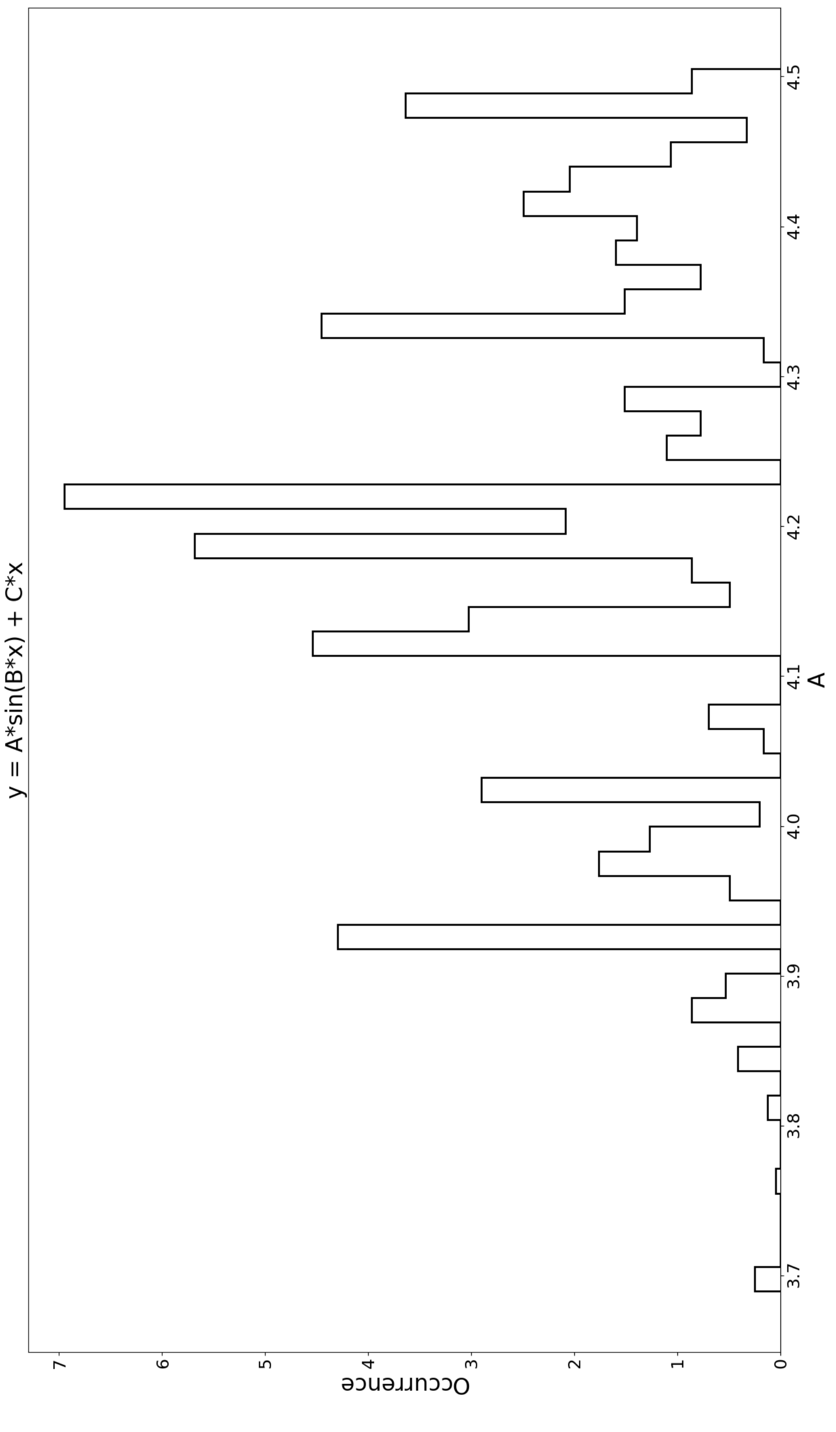

Figure 9: Posterior distribution for A for the sinusoidal-linear fit.

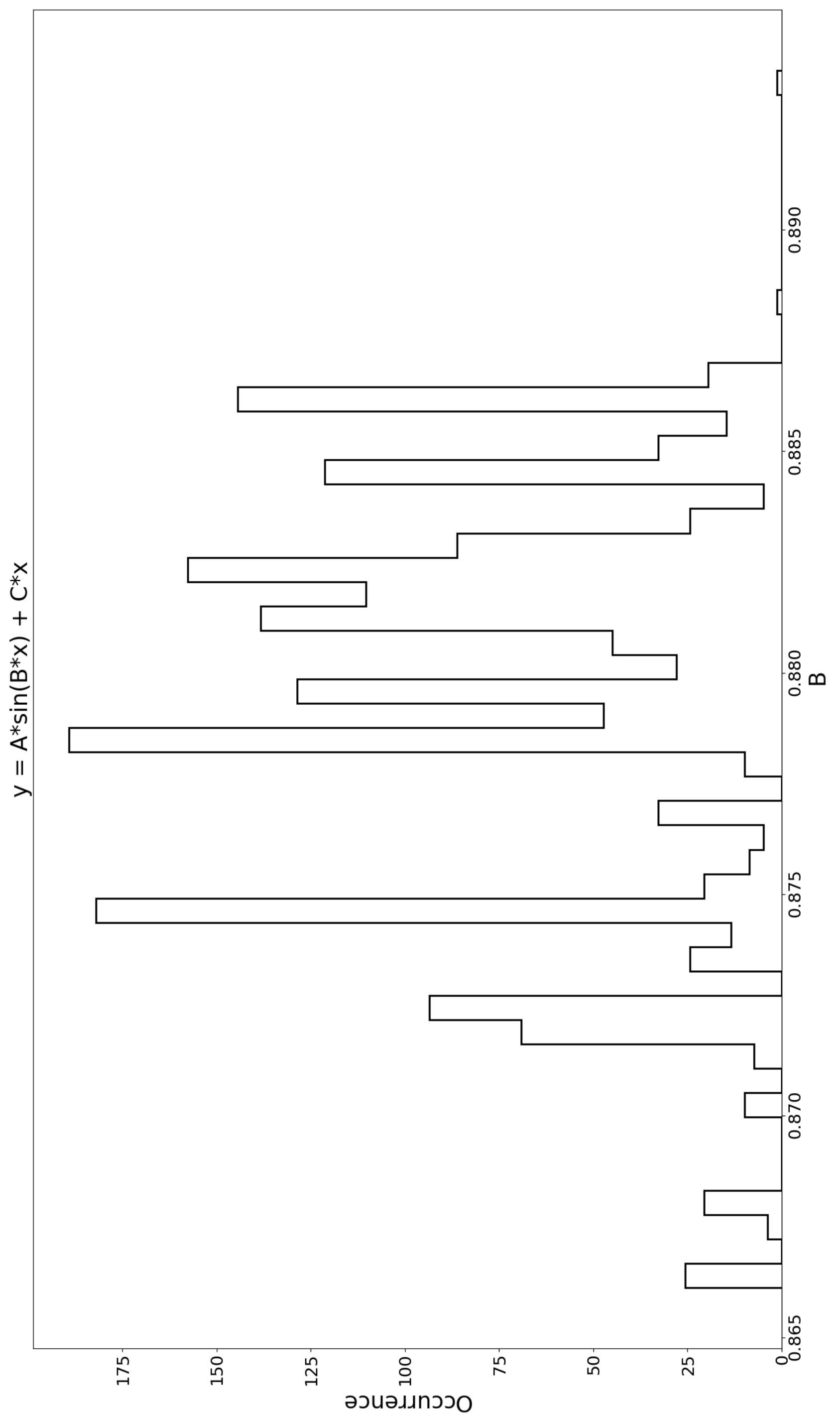

Figure 10: Posterior distribution for B for the sinusoidal-linear fit.

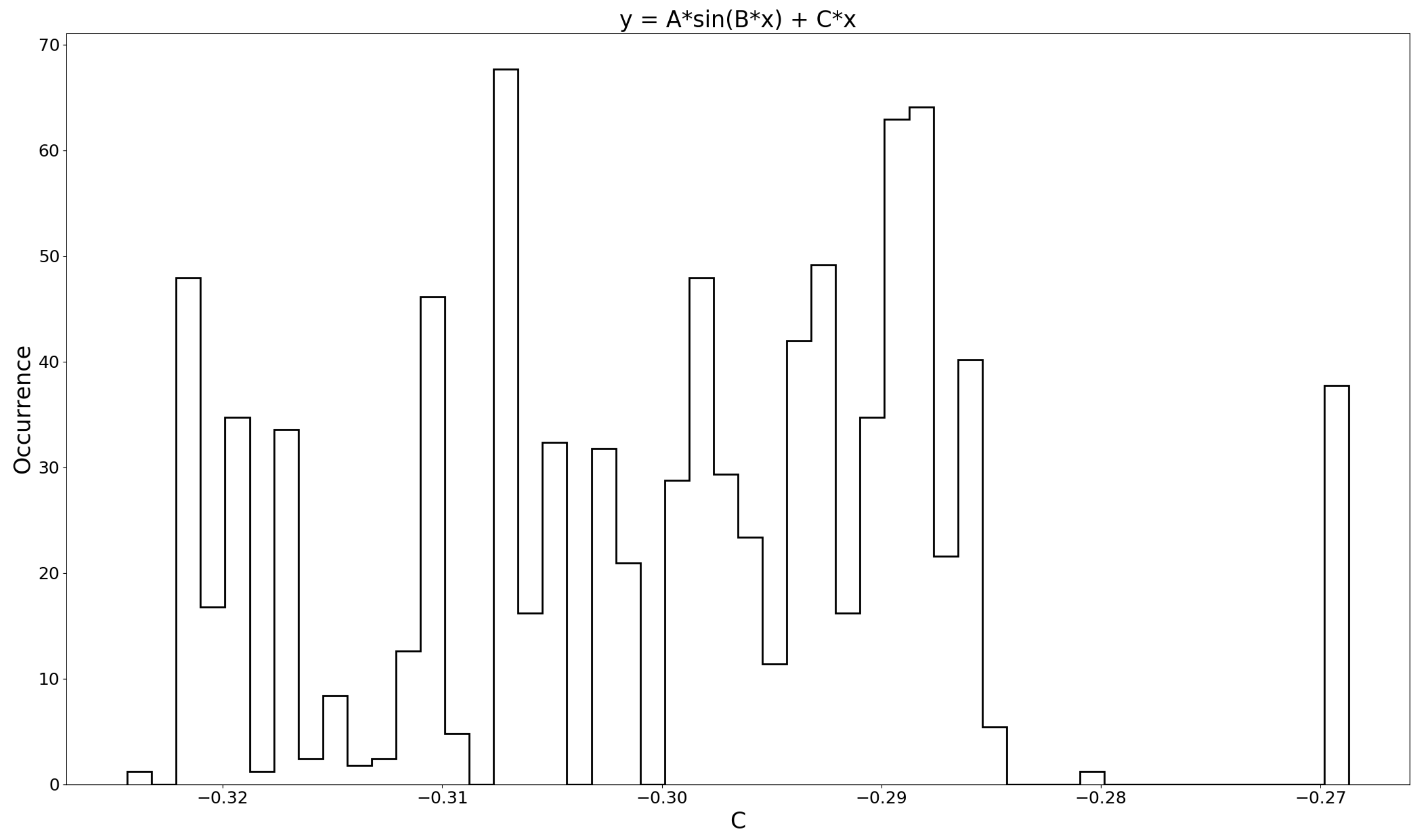

Figure 11: Posterior distribution for C for the sinusoidal-linear fit.

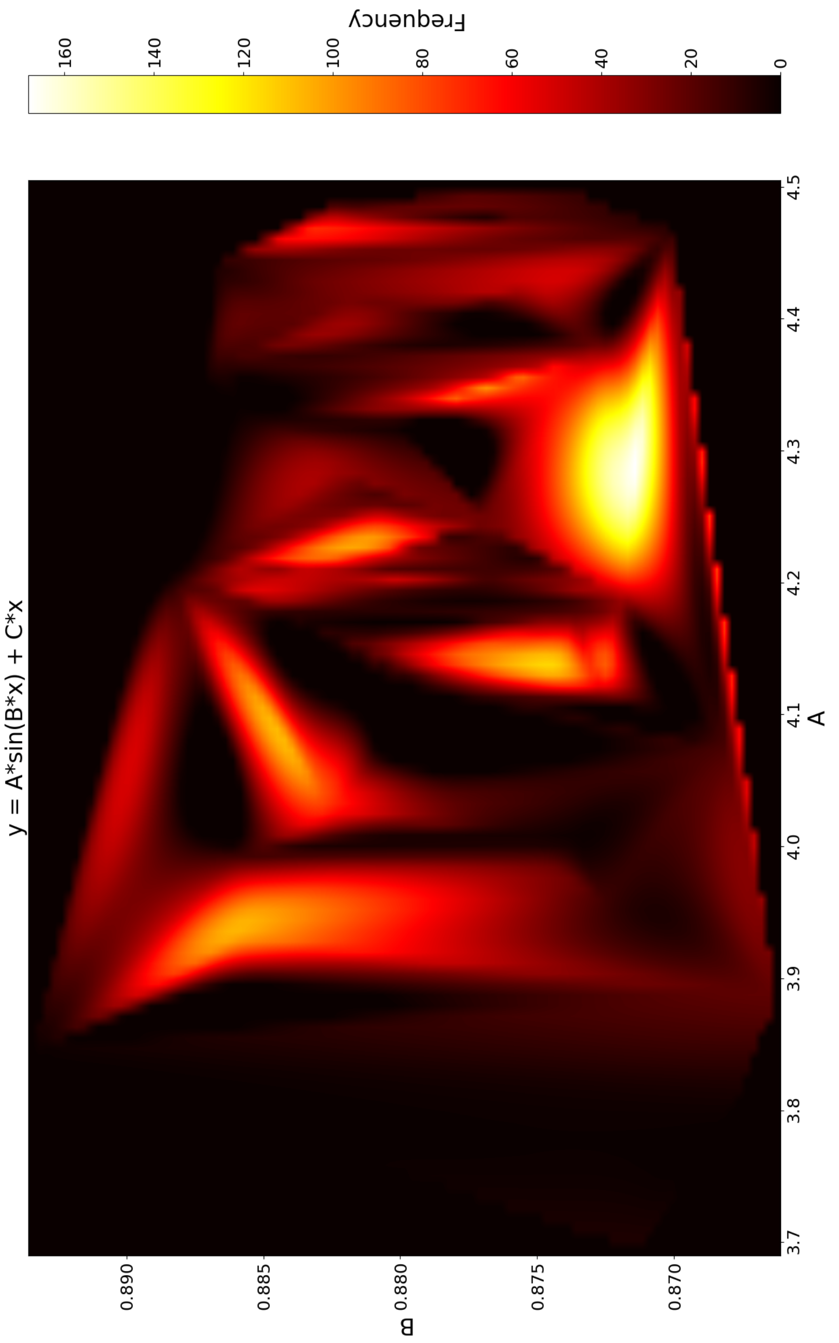

Figure 12: Posterior cross-distribution for A and B for the sinusoidal-linear fit.

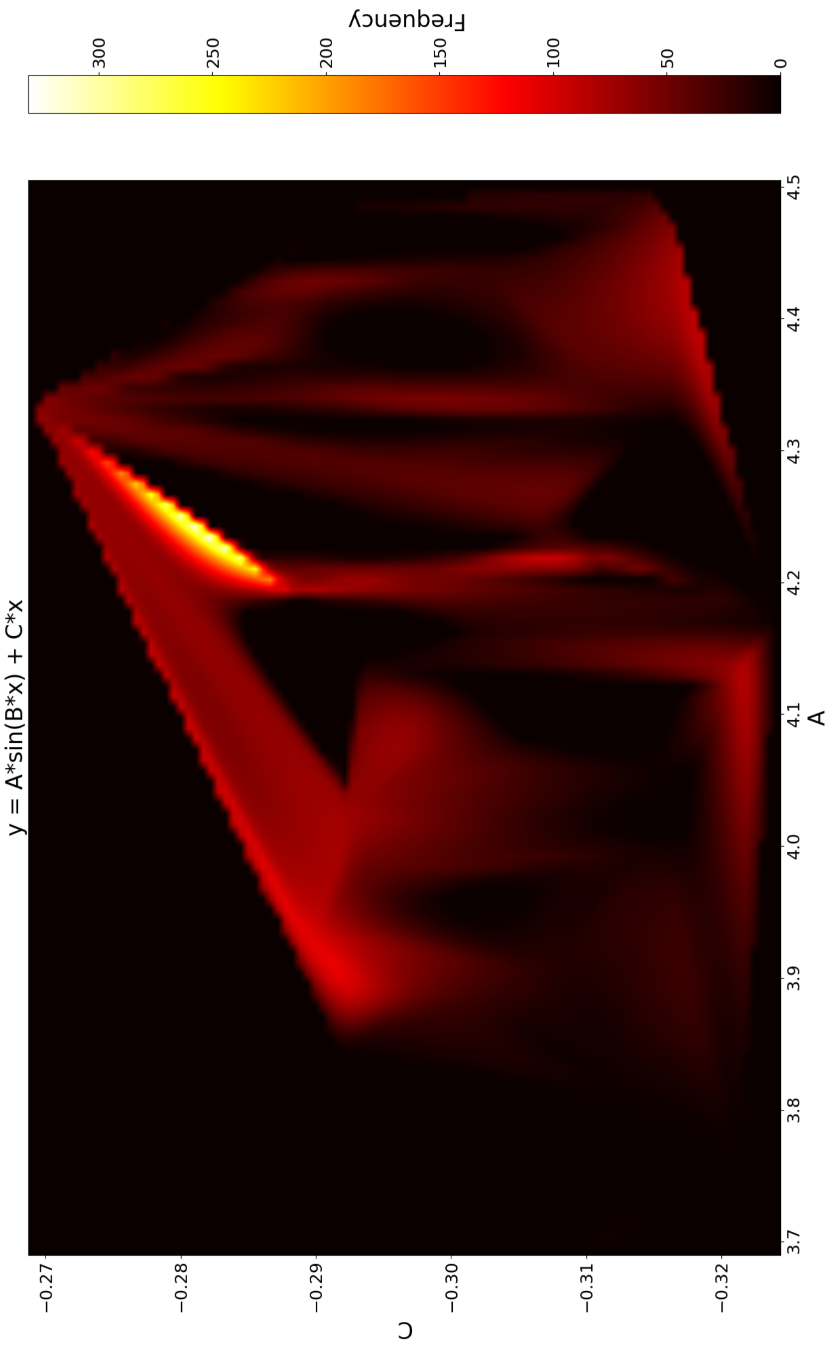

Figure 13: Posterior cross-distribution for A and C for the sinusoidal-linear fit.

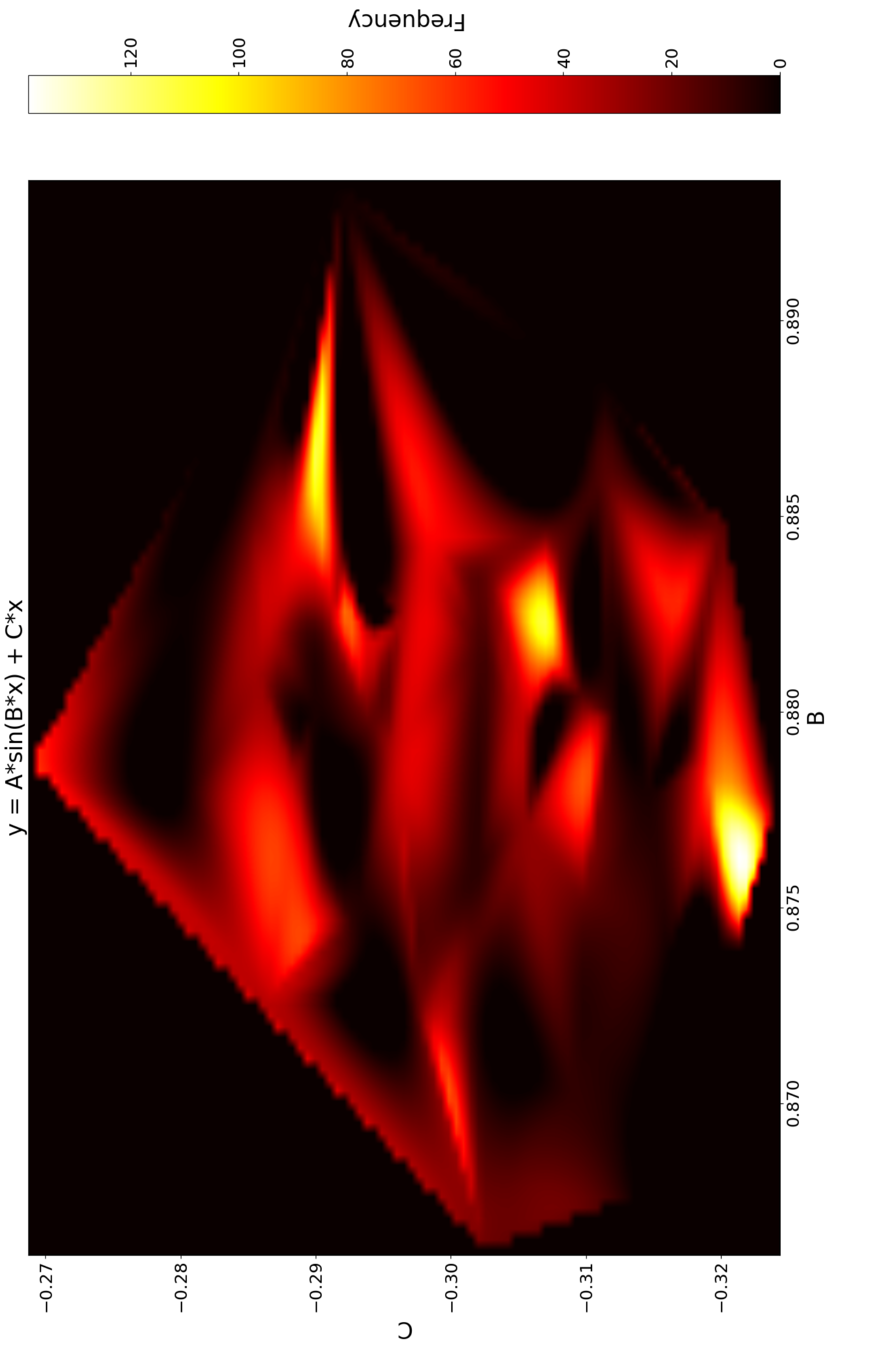

Figure 14: Posterior cross-distribution for B and C for the sinusoidal-linear fit.

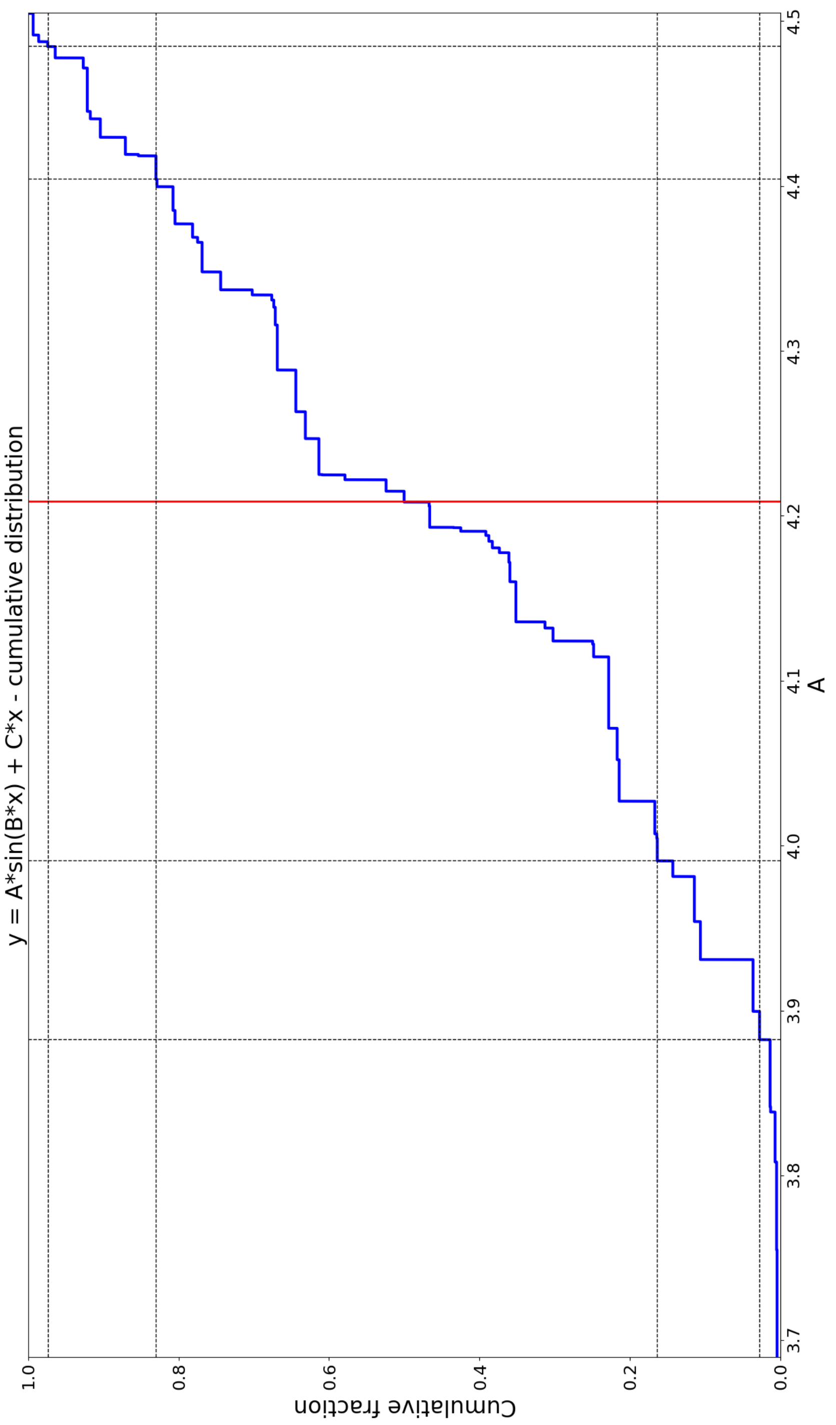

Figure 15: Cumulative distribution and 68%, 95% confidence intervals for A.

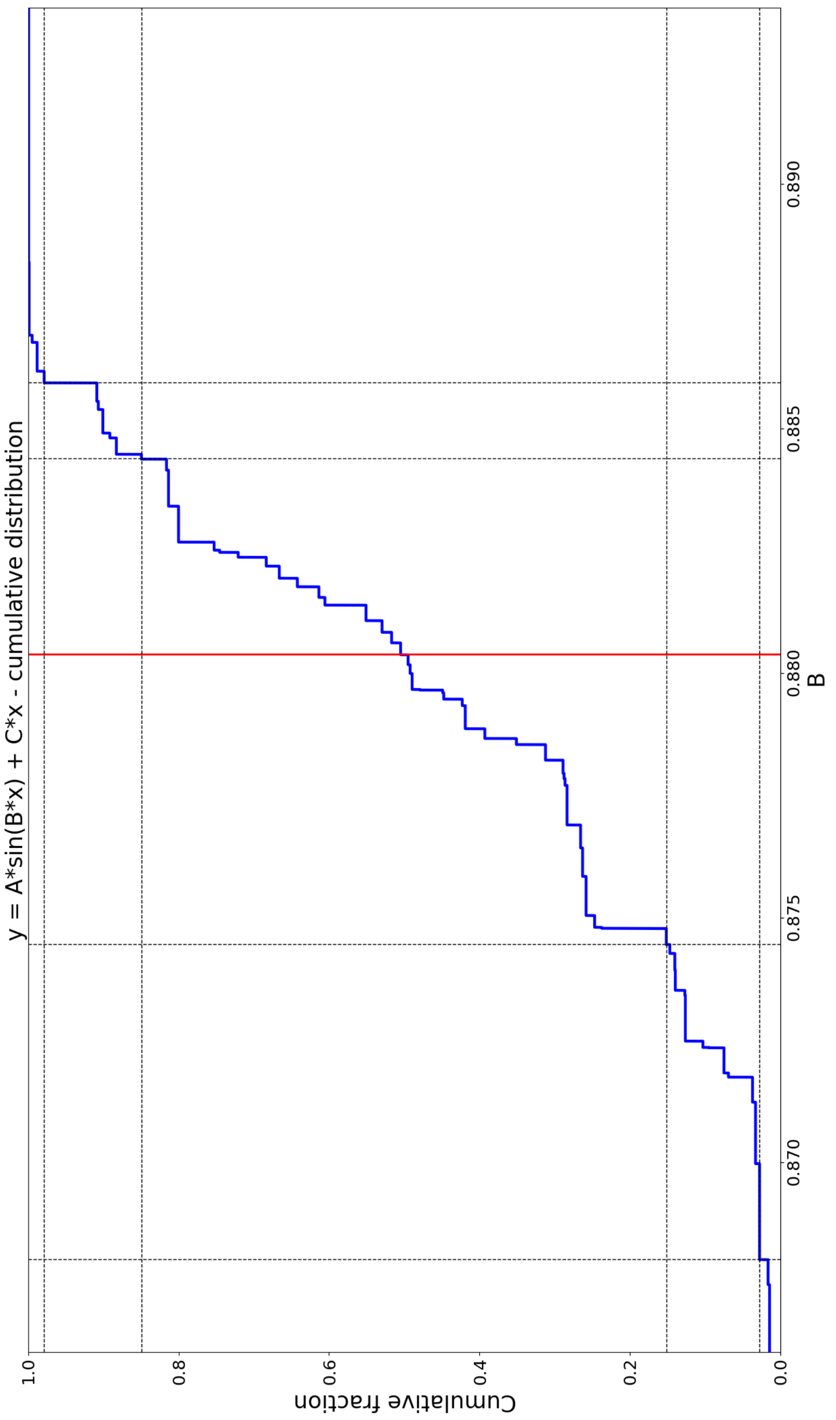

Figure 16: Cumulative distribution and 68%, 95% confidence intervals for B.

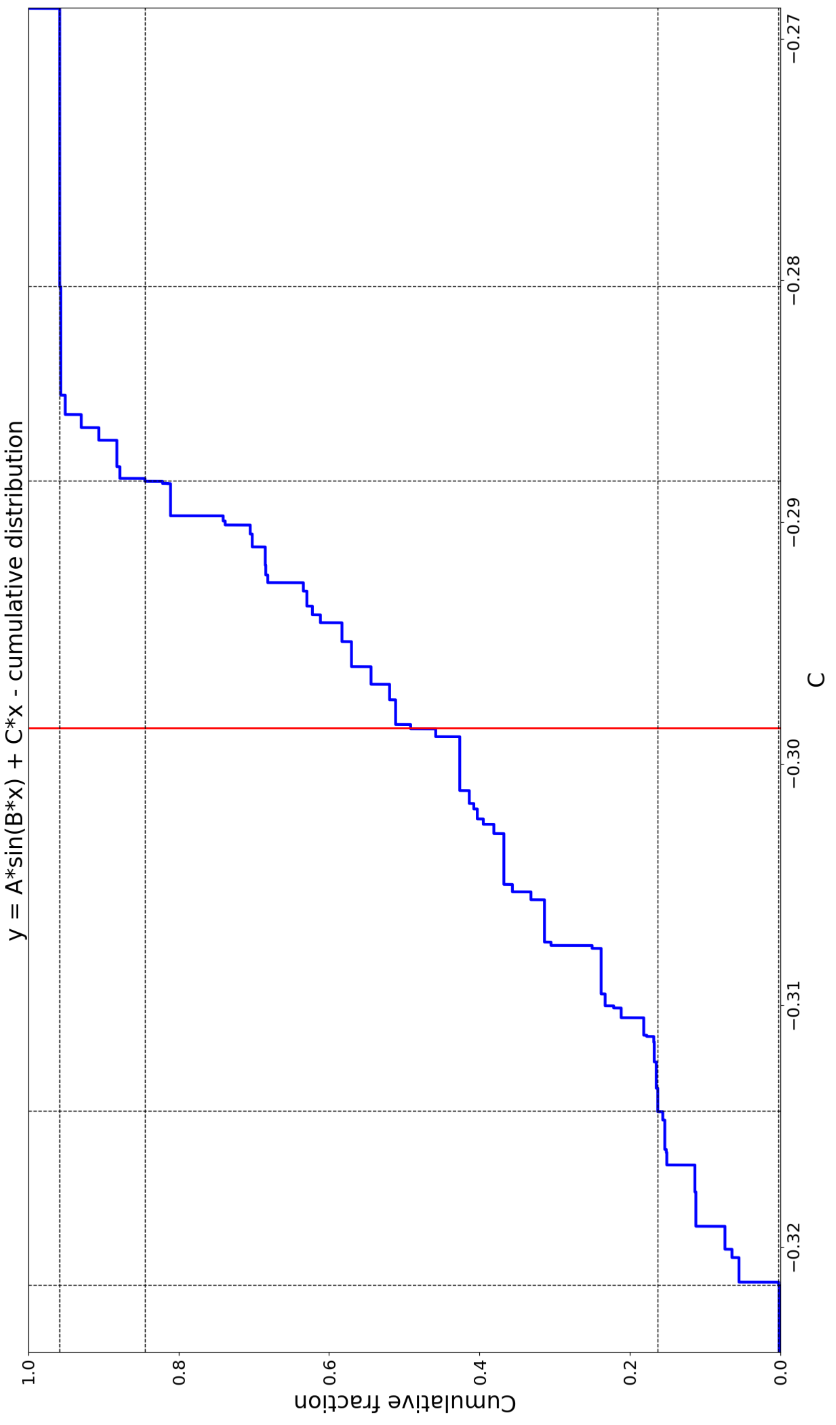

Figure 17: Cumulative distribution and 68%, 95% confidence intervals for C.

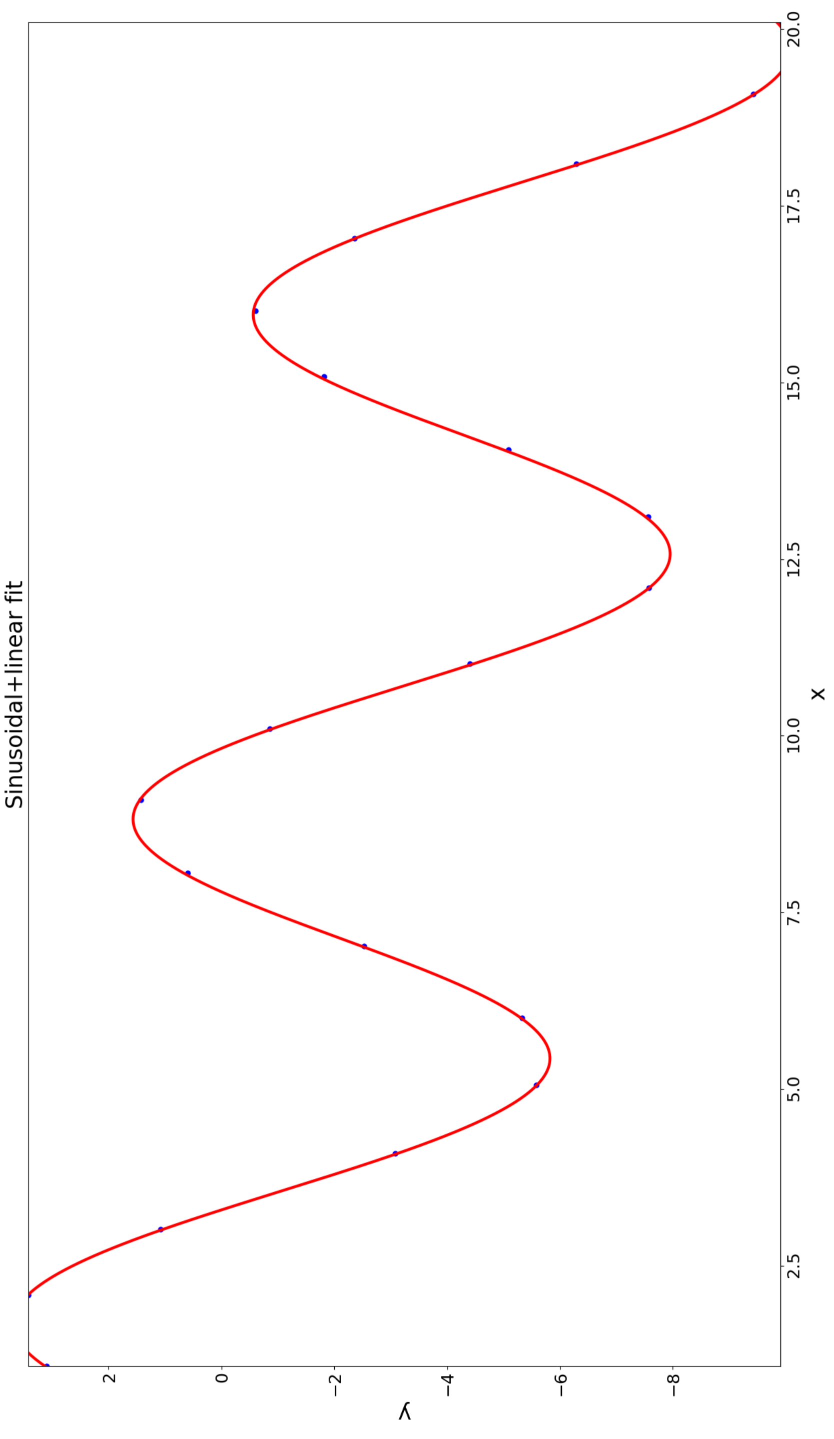

Figure 18: The resulting fit for the sinusoidal-linear function.