PHY407 (Lab 4: Solving Linear and Non-linear Equations)

Utkarsh Mali[1] and Aslesha Pokhrel[1,2]

[1]Department of Physics, University of Toronto
[2]Department of Computer Science, University of Toronto

October 7, 2020

Contents

Work Allocation:
Aslesha: Question 1, Question 3
Utkarsh: Question 2, Question 3

Question 1

(a) **In this part, we modified the module `SolveLinear.py` to incorporate the partial pivoting by adding `PartialPivot(A, v)` function.** This function was tested by comparing the result of (6.16) to Equation (6.2) and it produced the correct result. Nothing is required to be submitted for this part.

(b) **Here we test the performance of the Gaussian elimination, partial pivoting, and LU decomposition approaches (which are all mathematically equivalent) by comparing the accuracy and run-time of each of these methods.** The Gaussian elimination function was provided and we use for partial pivoting we use the function implemented in part **(a)**. Finally, `numpy.solve` was used for LU decomposition.

First, we measure the run-time for each of these methods to solve a system of random matrix A and random array `v` of size N where N ranges from 5 to 500. The same A and `v` is used for each of the three methods for the same value of N. The result is shown in the log-log plot (fig. 1). We can see that, as expected, the timing increases for all three methods as the matrix size increases. Similarly, LU decomposition is the fastest among all three methods. This is because of the optimization done by `numpy` library while implementing this method. Finally, partial pivoting takes longer time compared to the Gaussian elimination. This difference is widely visible for the smaller values of N but the difference becomes less distinct as the value of N increases. The partial pivoting is bound to have a slightly longer run-time due to the added step of finding the largest value in the column and swapping the respective rows, however, as we see from fig. 1, the cost becomes minimal for the larger values of N.

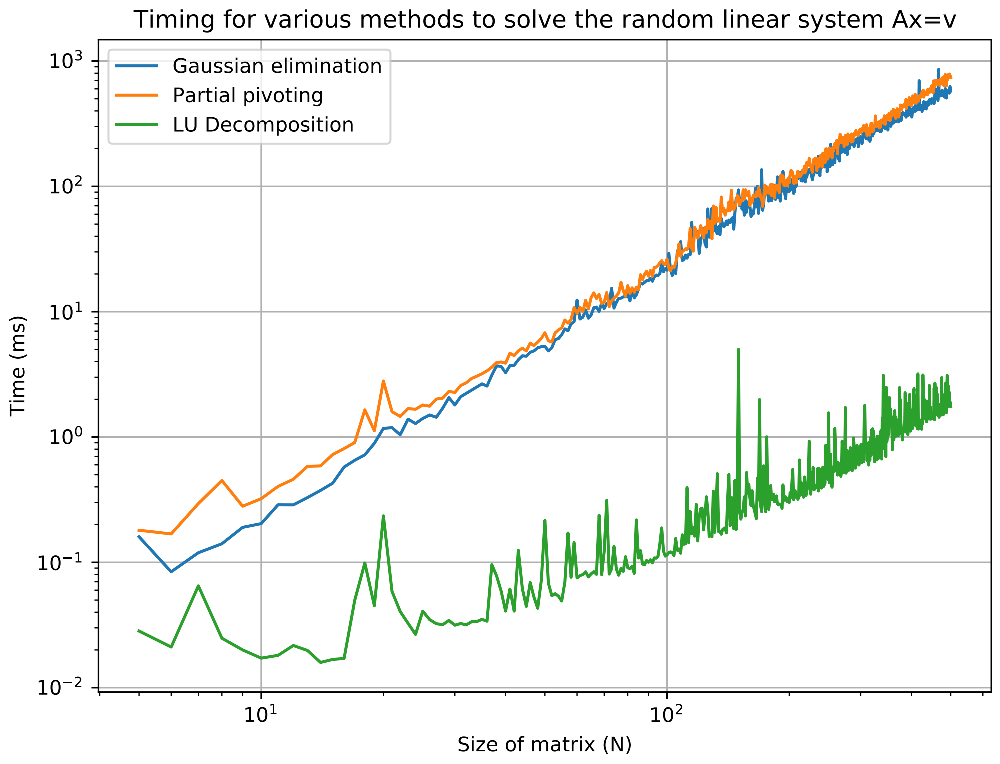

Figure 1: A log-log plot showing the run-time for three different methods to solve a random linear system of different sizes.

Next, we measure the error in the solutions of solving the same system described above by comparing `v_sol = numpy.dot(A, x)` to the original input array `v`. The error is is calculated by taking the mean of the absolute value of the differences between the arrays, i.e. `err = mean(abs(v-v_sol))`. The result is shown in the log-log plot in fig. 2. For the smaller values of N, the error is fairly small, closer to the machine epsilon. The error grows for all three methods as N increases. This is as expected since the larger matrices can be ill-conditioned and the rounding error also accumulates for the larger values of N. Similarly, the error from the Gaussian elimination is the highest among all three methods. This is most likely because of the presence of very small number in the diagonals as we need to divide by this number which can introduce big errors as seen in the lecture. Finally, the error for Partial pivoting and LU decomposition seems to be very close to each other. This shows that `numpy` most likely uses pivoting in the implementation of LU decomposition.

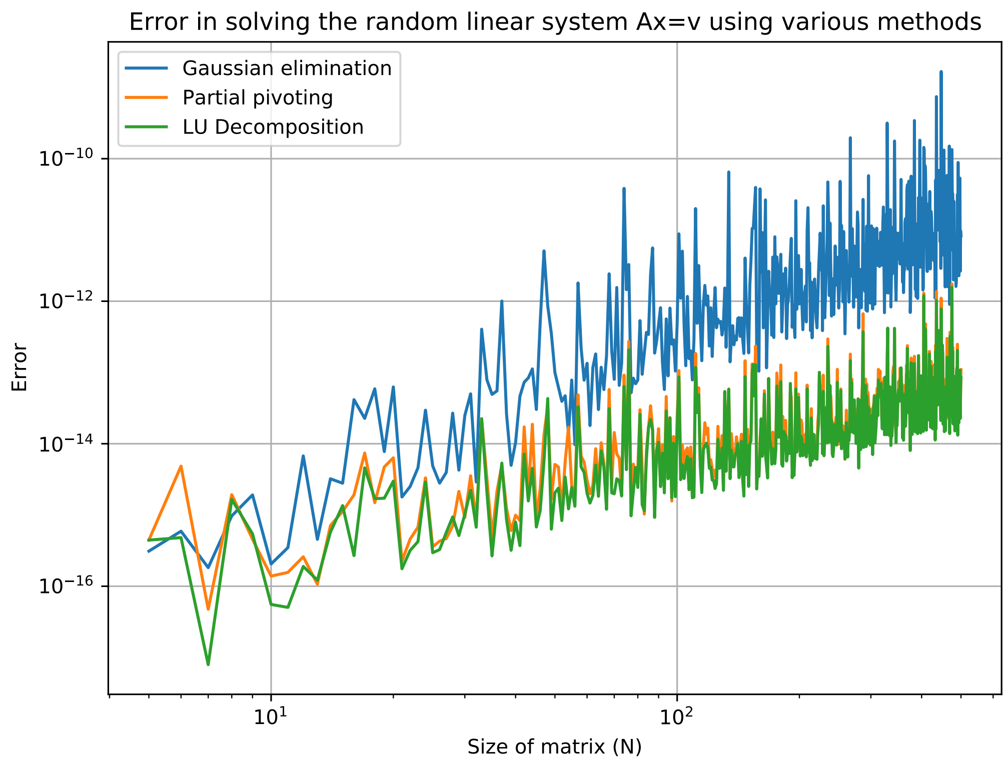

Figure 2: A log-log plot showing the error in solving a random linear system of different sizes using three different methods.

(c) In this question, we solve the linear system for the given electric circuit using the partial pivoting implemented in Q1a-b. We first solve for the given RC circuit and find the amplitudes and phases pf all three voltages. Next, we plot the (real) voltages as a function of time for two periods of the oscillations. The code output and the plot is given below:

```
Code Output:

     ### For RC circuit ###
The amplitudes of the three voltages at t=0 are given below:
   |V1| = 1.7014390658777336
   |V2| = 1.4806053465364062
   |V3| = 1.8607693200562132
Their phases in degrees at t=0 is:
   phase of x1 = -5.469094970111944
   phase of x2 = 11.583418604687065
   phase of x3 = -4.164672651865924
```

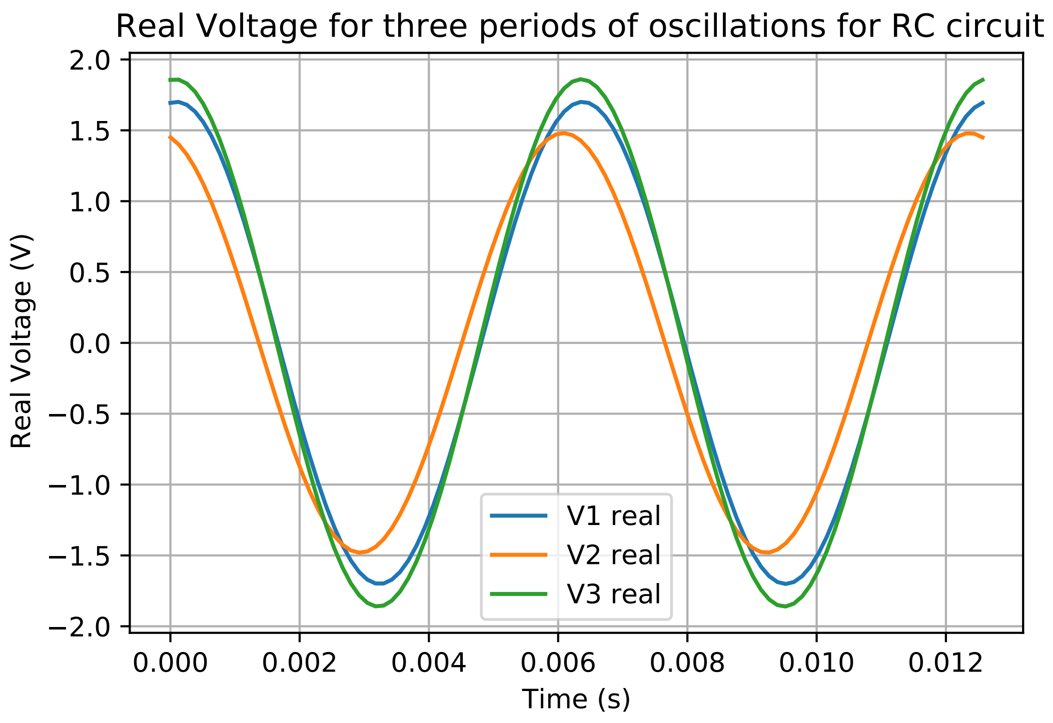

Figure 3: The plot showing the (real) voltages V_1, V_2 and V_3 for RC circuit for two periods of the oscillations.

Then, we replace the resistor R_6 with an inductor L. We do so by replacing all R_6 in the system of equation from before by $i\omega L$ where $L = 2H$ and $\omega = 1000 rad.s^{-1}$. Again, the the amplitudes and phases of V_1, V_2 and V_3, and plot V_1, V_2 and V_3 as a function of time is given below:

```
Code Output:

     ### For RLC circuit ###
The amplitudes of the three voltages at t=0 are given below:
   |V1| = 1.5621181940219633
   |V2| = 1.4994286802306562
   |V3| = 2.8112763903392537
Their phases in degrees at t=0 is:
   phase of x1 = -4.025908819603362
   phase of x2 = 21.63928264257023
   phase of x3 = 14.352479528588603
```

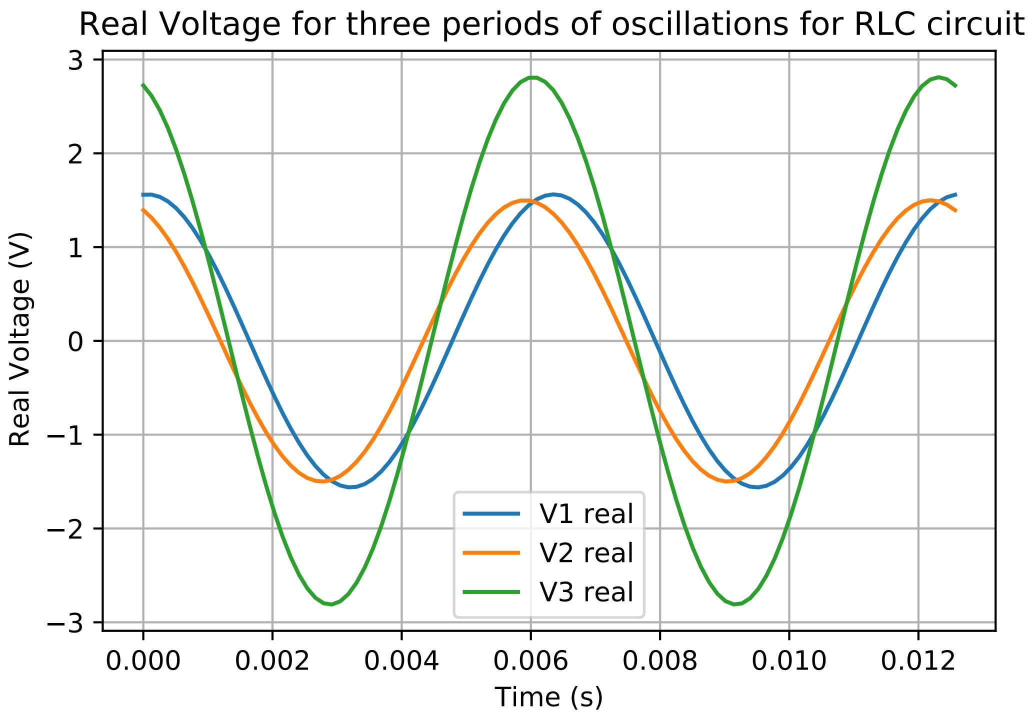

Figure 4: The plot showing the (real) voltages V_1, V_2 and V_3 for RLC circuit for two periods of the oscillations.

The impact of replacing the resistor R_6 by the imaginary impedance of the same magnitude, $i\omega L = iR_6$ as $L = R_6/\omega$ is that the part of the circuit where the inductor is placed is less resistive and therefore less dissipative. Therefore, the voltage obtained is of higher amplitude as seen in fig. 4 compared to fig. 3 since the circuit has less dissipative effect on the current.

Question 2

(a) **This part requires us to solve the equation given in the text.** The solution to this section has been provided in the handout for the Lab.

(b) **This part also requires us to solve the equation given in the text.** The solution to this section has also been provided in the handout for the Lab. The few short lines of code used in writing the program can be found in the python file attachment in which $H(m, n)$ is defined.

(c) **In this section, we are tasked with creating a matrix which we then use to solve the eigenvalue equation. We can also use the recommended format for completing the H Matrix.** Having set out our initial $n \times n$ 10-dimensional matrix we compute each term using the guide featured in the handout. Since we know this matrix is hermitian, we can use `np.linalg.eigh()` for a much faster compute time. The computer value showed in the text has also been displayed.

(d) **This part requires us to modify some parameters in order to accommodate a larger matrix size. We are to then look at the difference in values and explain why there might be such a discrepancy** I simply changed the values of m and n to accommodate for the larger size and let the code run as normal. **There is a discrepancy between the two set of values due to the compute computer prevision error** that was done during calculation. Since there is supposed to be an infinite amount of n, truncating it will truncate the reliability of our results.

(e) **Here we are asked to modify the program in order to calculate the wave-function which we are supposed to plot with a normalization constant.** We start by defining $\Psi(x)$ and $V(x)$ as required for computing the first 3 wave-functions. In our definition we iterate over all n (within the definition) to calculate the value of Ψ for that particular state. We access the general eigenvector for that state during the iteration and select the according n state as required. That is then multiplies by $\sin\left(\frac{n\pi x}{L}\right)$ in order to finalize the sum. After this we initialize an array of x values and the probability density function which is then used to calculate the the values of each of the probabilities for a given values of x. We then use `gaussxw(N)` to integrate using Gaussian quadrature, it was chosen since it was the best type of integration typically giving the most reliable results. The square root was returned as specified in the lab handout. Finally the three wave-functions were plotted as a function of x and the axis's were clearly labeled.

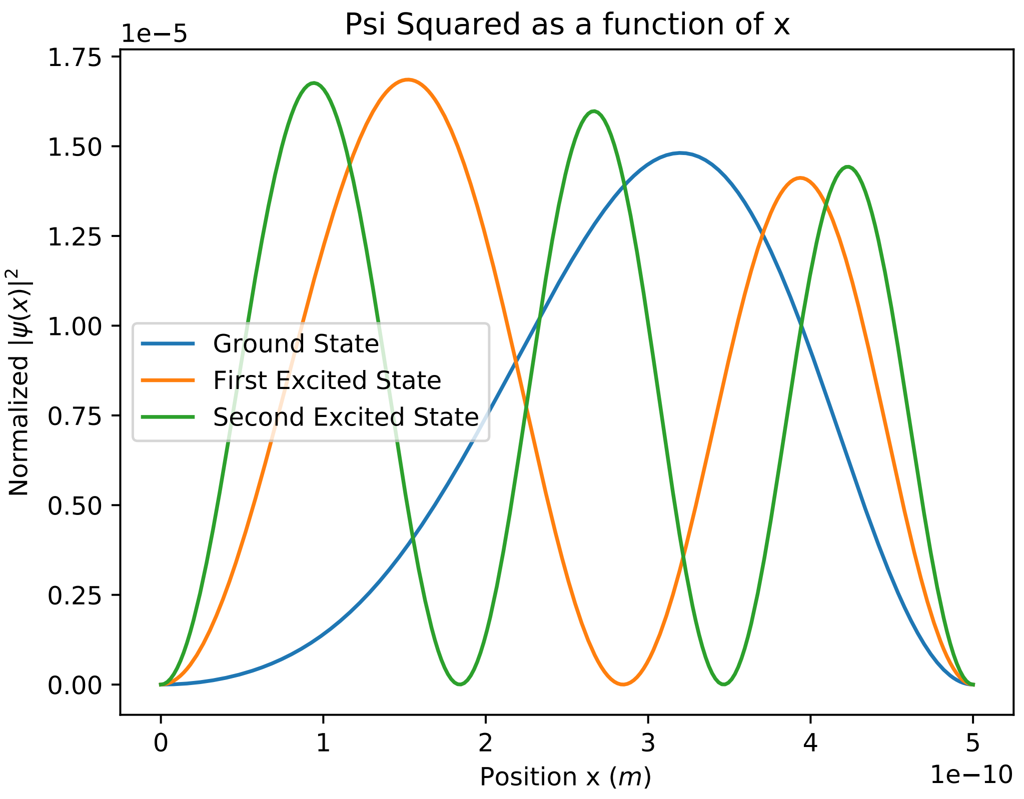

Figure 5: Above are the ground state and first 2 excited states as solutions of the Schrodinger Equation from the varying potential well given in Exercise 6.9 of the Newman computational physics book. It incorporates the solved solutions and includes normalization constants in order to make sure that the values of the sum of all functions sums to one to maintain the orthonormality of the Hilbert space. All values were computed in electron volts.

Code Output:

```
Ground State Energy: 5.8364 eV

Comparison of eigenvalues at first 10 points:
 [  5.8363769  11.18109291 18.66289158 29.14419776 42.65507485
  59.18525782 78.72936019 101.28548383 126.85138575 155.55532885]
 [  5.8363765  11.18109158 18.66288971 29.14418896 42.65506573
  59.18520524 78.72930836 101.2848529 126.85055342 155.42570639]
```

Question 3

(a) **Here we are asked to write a program that solves the given non-linear equation $x = 1 - e^{-cx}$ using the relaxation method.** We first set-up a generic solver and then specified values of c, such as the one used in our plot.

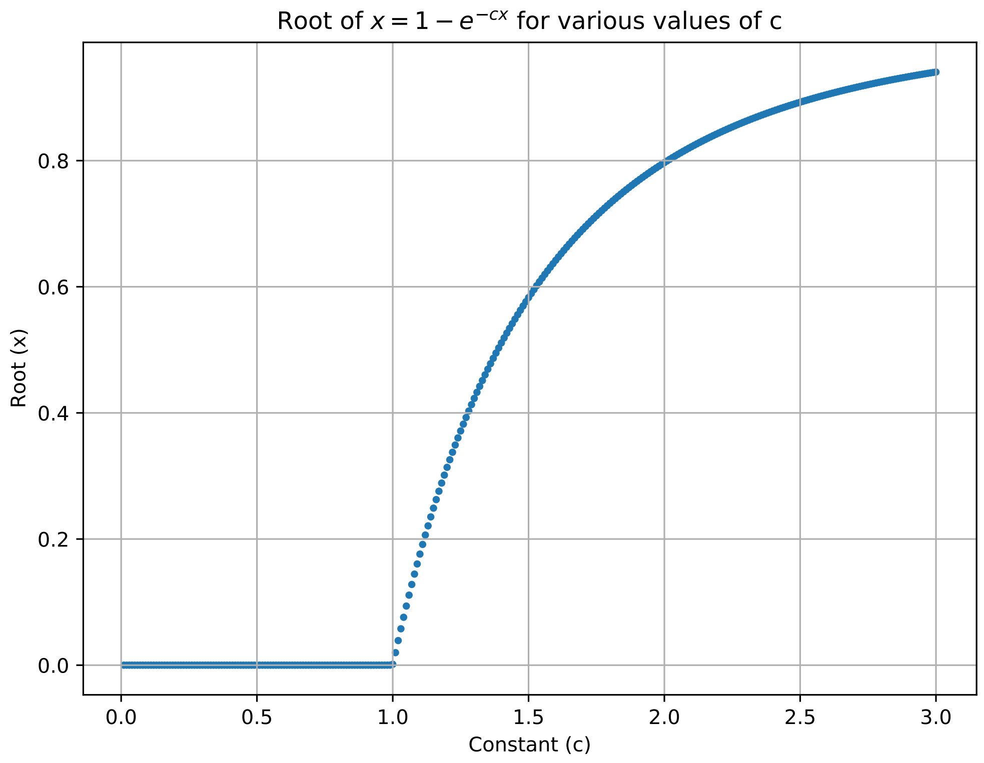

Figure 6: The plot shows the root (value of x) obtained for difference constants (c) using the relaxation method for the equation $x = 1 - e^{-cx}$.

We can see a clear transition from regime in which $x = 0$ to a regime of nonzero x in fig. 6 as expected.

(b) **In this part we continue with the equation $x = 1 - e^{-cx}$ for $c = 2$ and solve exercise 6.11 parts b, c and d where we implement the overrelaxation method. We also compare the number of iterations required to converge using the overrelaxation method to the number of iterations using the relaxation method.**

We first measure the number of iterations required to converge to a solution accurate to 10^{-6}. The output from the code is given below:

```
Code Output:

Using the relaxation method:
   It takes 14 iterations to converge to a solution accurate to 1e-6
   It converges to x = 0.7968126311118457
```

Now, we write program to solve for x for the same equation with the same value of c and same accuracy. After experimenting with a few values of ω, $\omega = 0.5$ was found to perform the best as it takes the lowest number of iteration to converge to a solution. The output from the code is given below:

```
For overrelaxation method, omega = 0.5 gives the lowest number of iterations to
    converge to a solution accurate to 1e-6.
    In this case, the number of required iteration is 5
    And, the value of x is 0.7968121566399141
```

Thus, from the output shown above, the overrelaxation method clearly outperforms the relaxation method as the number of required iterations for the overrelaxation method is less than half of the relaxation method. Hence, the overerelaxation method is at least twice as fast as the relaxation method.

Next, we consider the circumstances where using the value $\omega < 0$ would help us find a solution faster than the ordinary relaxation method. Indeed, when we are solving for the root of a function with very steep slope, using $\omega < 0$ can help since the relaxation method by itself might overshoot in these situations and fail to find the root or take longer.

(c) **This section requires us to use the binary search method for the given equation and then use the solution to approximately estimate the surface temperature of the Sun.** We started off by defining an arbitrary `binary_search` method which takes in any function and returns the root values between the point estimated points which were taken to be $x_1 = 1$ and $x_2 = 100$ respectively. This method was then compared against the newton and relaxation methods.For the binary search method, the value was computed as instructed in the book and additional helper functions were defined to improve readability. Subsequently, the newton method was defined, again, using the textbook definition of computing the Newton method. Since the solution was known, a rough estimate near the solution was given in order to enable the Newton method to operate as expected. A new derivative function was required to be defined in order to do this. Finally, the relaxation method was implemented using a newly defined function as specified in the course textbook. **All the values differ slightly due to the computational precision error.** Finally we used the equation and constants specified in the course in order to calculate the surface temperature of the Sun, as required.

Code Output:

```
Relaxation Method: 4.965114221697042
Newton's Method 4.965114231752603
Binary Search Method: 4.965114112943411
Surface Temperature of the Sun: 5772.454232112551
```