UNIVERSITY OF OSLO

COMPUTATIONAL PHYSICS

Project 1

Authers:

Birgitte Madsen

Soumya Chalakkal

Autumn 2015

UiO : University of Oslo

Department of Physics
University of Oslo
Sem Sælands vei 24
0371 Oslo, Norway
+47 22 85 64 28
http://www.mn.uio.no/fysikk/english/

Course:

Computational Physics

Project number:

1

Link to GitHub folder:

Link to Github folder

Hand-in deadline:

Monday, September 14, 2015

Project Members:

Birgitte Madsen
Soumya Chalakkal

Copies: 1
Page count: 17
Appendices: 0
Completed: Friday, September 11, 2015

TABLE OF CONTENTS

Chapter 1 Introduction **1**

Chapter 2 Method **3**

2.1 Nature of the problem 3

2.2 Description of the Algorithm 4

Chapter 3 Results **7**

3.1 Interpretation of Results 7

3.2 Comparing the Algorithm to the LU-decomposition Method 10

Chapter 4 Conclusion **11**

Bibliography **13**

Appendix A MatLab code for plotting numerical and closed-form solution **15**

Appendix B MatLab code for plotting the relative error **17**

CHAPTER 1

INTRODUCTION

The problem of solving differential equations appear in various places in nature. It is, however, not always possible to solve these differential equations analytically, and one might instead need solve the problem numerically.

This project aims to solve the problem of finding a solution for the general one-dimensional Poisson equation with the double differentiated function approximated by the three point formula using less floating point operations (flops) than needed in the ordinary Gaussian elimination and the LU decomposition method. The number of flops is reduced by identifying that the Poisson equation can be numerically solved by solving a matrix equation in which the matrix is tridiagonal. This ultimately ends out in a number of flops that goes as $\mathcal{O}(n)$ for this specific problem which is evidently a great reduction from the number of flops needed in the other two methods.

Futhermore, another objective of the project is to consider how a decrease of step length of the variable x influences the precision of the numerical solution compared to the closed-form solution. It is seen that for the step lengths chosen, a decrease in step length causes an increase in accuracy of the solution.

CHAPTER

2 METHOD

This chapter provides argumentation for why the Poisson equation can be solved numerically using a linear set of equations. Thereafter, the algorithm for solving the problem is discussed, and the number of flops in this algorithm is compared to the number of flops in the ordinary Gaussian elimination and LU decomposition method.

The source code itself can be found in the GitHub folder [1]

2.1 Nature of the problem

The problem is to solve the one-dimensional Poisson equation with Dirichlet boundary conditions, given as

$$-u''(x) = p(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0 \tag{2.1}$$

To solve (2.1) numerically, the second derivative of u is approximated by the three point formula, which gives the following reformulation of the problem.

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = p_i \qquad \text{for } i = 1, \dots, n \tag{2.2}$$

This equation can be rewritten as a linear set of equations of the form

$$\mathbf{Av} = \mathbf{f} \tag{2.3}$$

with A beeing a $n \times n$ tridiagonal matrix given by

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix} \tag{2.4}$$

[1] FiXme Note: link to GitHub folder

To prove this first consider the matrix equation

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ 0 & a_3 & b_3 & c_3 & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & \dots & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}. \tag{2.5}$$

Solving the above matrix equations gives a set of linear equations as

$$\begin{aligned} 2v_1 - v_2 &= f_1 \\ -v_1 + 2v_2 - v_3 &= f_2 \\ -v_2 + 2v_3 - v_4 &= f_3 \\ &\vdots \\ -v_{n-1} + 2v_n - v_{n+1} &= f_n \end{aligned}$$

In general we can write it as a

$$-v_{i-1} + 2v_i - v_{i+1} = f_i \qquad \text{for } i = 1, \dots, n \tag{2.6}$$

If we substitute $f_i = p_i h^2$ then (2.6) becomes

$$-\frac{-v_{i+1} + 2v_i - v_{i-1}}{h^2} = p_i \qquad \text{for } i = 1, \dots, n \tag{2.7}$$

(2.7) is equal to (2.2) for second derivative of u. Thus proving that the equation for second derivative of u can be rewritten as a set of linear equations of the form

$$\mathbf{A}\mathbf{v} = \mathbf{f} \tag{2.8}$$

2.2 Description of the Algorithm

The algorithm written to solve the problem of computing vector **v** in (2.7), written out as the matrix equation

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}.$$

uses the Gaussian elimination method. However, since the linear problem includes a special matrix, namely a tridiagonal matrix, the number of floating points operation needed to solve this specific problem can be reduced by modifying the Gaussian elimination method.

The algorithm used to solve this problem, still, consists of the same two steps as in the Gaussian elimination: forward substitution and back substitution.

Let us first address the forward substitution. The aim of the forward substitution is essentially to make the matrix **A** into an upper triangular matrix by suitable subtractions of multiple of the first row from the other rows in the matrix. This gives rise to a change in the matrix elements. However, the elements in the first row of the matrix will not be changed, and hence we have that

$$\tilde{b}_1 = b_1 \tag{2.9}$$

in which $\tilde{b}_1$ is the first element in the first row of the computed triangular matrix. By writing out the computed matrix elements after subtracting multiple of the first row from the other rows to create zeros below the diagonal, it is seen that the elements in the diagonal for $i > 1$, named b_i in the tridagonal matrix **A** and $\tilde{b}_i$ in the computed triangular matrix, get the value

$$\tilde{b}_i = b_i - \frac{1}{b_{i-1}} \qquad \text{for } i = 2, \ldots, n \tag{2.10}$$

(2.10) has that form since all elements right above and below the diagonal of **A** are equal to -1, the ones in the diagonal of **A** are b_i, whilst the remaining elements of the matrix are 0. Likewise, the elements in the vector **f** are changed to

$$\tilde{f}_i = f_i + \frac{f_{i-1}}{b_{i-1}} \qquad \text{for } i = 2, \ldots, n \tag{2.11}$$

whilst the elements named a_i in **A** become equal to zero, and the elements c_i are unchanged.

This gives rise to the following code for the forward substitution.

```
// Forward substitution

   double abtemp[n];
   double btemp = b[0];

   for (int i=1 ; i<n ; i++)
   {
      abtemp[i] = - 1/btemp;
      btemp = b[i] + abtemp[i];
      f[i] = f[i] - f[i-1]*abtemp[i];
      b[i] = btemp;
   }
```

Notice that in the above lines of code, the first element of a vector is $i = 0$.

For every time the for loop runs, there are 4 flops. We have chosen to calculate $1/b_{i-1}$, which is used in both (2.10) and (2.11), as the very first operation in the for loop to reduce the number of flops by 1 for every time the loop is run. Since the loop runs for $i = 2$ to $i = n$, if $i = 1$ is the first element of a vector (remember, the c++ code above has $i = 0$ for $i = 1$), the loop runs $n - 1$ times, which gives a total number of flops for the forward substitution of

$$\#flops = 4(n-1) \tag{2.12}$$

In the back substitution, the values of the entrances of vector **v** in (2.8) are computed. Since the result from the forward substitution is an upper triangular matrix, it is evident that

$$v_n = \frac{\tilde{f}_n}{\tilde{b}_n} \tag{2.13}$$

in which $\tilde{f}_n$ and $\tilde{b}_n$ are elements of **f** and **b** after the forward substitution. From the determined value of v_n, the values of the rest of the v_i's can be determined using the fact, that all elements in the upper triangular matrix are zero apart from the elements in the diagonal and the elements just above the diagonal. Since the elements just above the diagonal are still the c_i's, they have the value -1, which yields that the value of v_i can be determined by

$$v_i = \frac{\tilde{f}_i + v_{i+1}}{\tilde{b}_i} \qquad \text{for } i = 1, \ldots, n-1 \tag{2.14}$$

yielding a source code:

```
// Back substitution

   v[n-1] = f[n-1]/b[n-1];

   for(int i=n-1 ; i>= 0; i--)
   {
      v[i] = (f[i]+v[i+1])/b[i];
   }
```

Once again, the first element of a vector in the source code above has the index $i = 0$. Hence, $v[n-1]$ is the same as v_n.
For each time the loop runs, there are 2 flops. Like in the forward substitution, the loop runs $n-1$ times, yielding

$$\#flops = 2(n-1)+1 \tag{2.15}$$

The number $+1$ is included in (2.15) due to the initial calculation of v_n just before the for loop. Hence, the total number of flops for both the forward substitution and back substitution is

$$\#flops_{total} = 4(n-1)+2(n-1)+1 = 6n-6 \tag{2.16}$$

which gives that the number of flops goes as $6n$ or $\mathcal{O}(n)$.

When comparing the number of flops using the above described algorithm to solve (2.7) in which **A** is a tridiagonal matrix and **v** is approximated by the three point formula (2.2) with the number of flops in the ordinary Gaussian elimination or LU decomposition, it is evident that this algorithm is much more efficient to solving this specific case, since the number of flops for solving a linear set of equations using the LU decomposition scales as $\mathcal{O}(n^2)$, whilst the LU decomposition itself scales as $\mathcal{O}(n^3)$, and the number of flops required in the Gaussian elimination is $2n^3/3 + \mathcal{O}(n^2)$. [1, 173]

CHAPTER 3

RESULTS

When running the code presented in Chap. 2 with different number of grid points n's, corresponding to different step lengths h, the value of the numerical solution for an arbitrary x_i will be different, as well. This is due to the fact, that a decrement or increment of the step length will change the precision of the result. This chapter strives to consider what happens to the precision of the numerical solution as the number of grid points is changed from $n = 10$ to $n = 100$ and $n = 1{,}000$.

In the final part of the following section, the relative error is plotted as a function of $\log(h)$, and in this part the number of grid points is increased to $n = 10{,}000$ and $n = 100{,}000$.

The results from running the code with 10, 100, 1,000, 10,000, and 100,000 grid points can be found in the GitHub folder [1].

3.1 Interpretation of Results

It can be shown by inserting into (2.1) that

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{3.1}$$

is a solution to the differential equation for $p(x) = 100e^{-10x}$ and hence $f(x) = 100e^{-10x}h^2$. The expression in (3.1) is the closed-form solution to the one-dimensional Poisson equation.

In the figure below the closed-form solution given in (3.1) is plotted as a function of x together with the numerical solution gained from the algorithm described in Sec. 2.2 for three different number of grid points n, namely 10, 100 and 1,000. The plots are made in MatLab and the closed-form solution is plotted with 1000 grid points. The source code for the MatLab plot can be seen in App. A.

[1] FiXme Note: linkt to Github-address

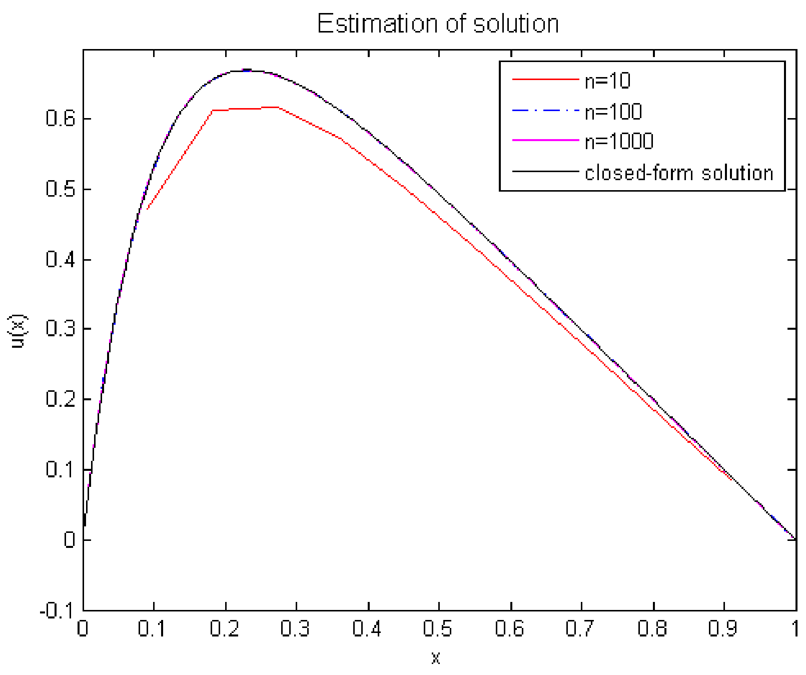

Figure 3.1. Plot of the closed-form solution together with the numerical solution for different number of grid points. It is evident that the accuracy of the numerical solution gets better when number of grid points is increased from $n = 10$ to $n = 100$ and $n = 1000$.

From Fig. 3.1 it is easily seen that by increasing the number of grid points from $n = 10$ to $n = 100$ the precision of the solution is better. By zooming in on the figure, it can be seen that an increment of n from 100 to 1000 actually gives a further improvement to the numerical solution, as can be seen in Fig. 3.2.

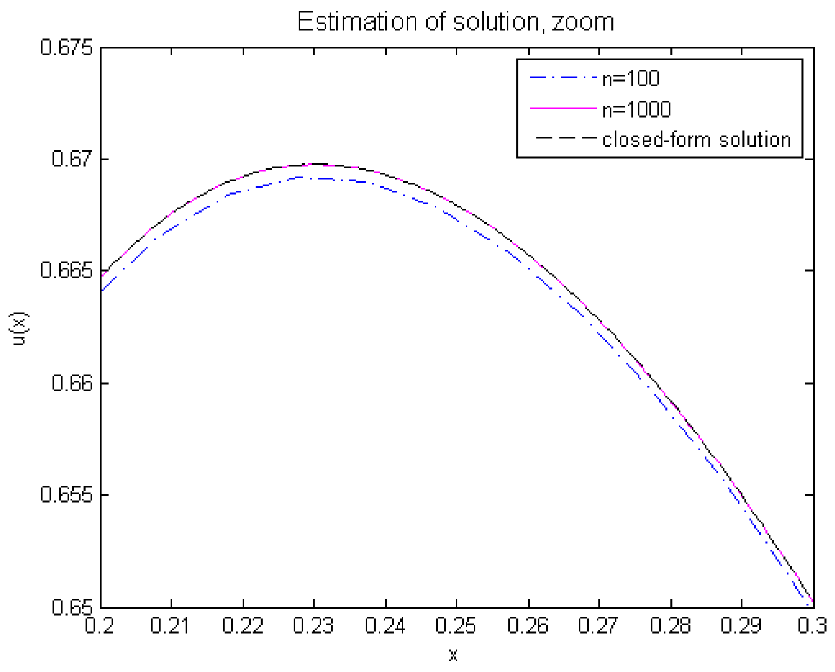

Figure 3.2. Zoom of Fig. 3.1. It is seen that a further increment of n from 100 to 1000 actually improves the accuracy of the result.

From Fig. 3.1 and Fig. 3.2 it is evident that there is a deviation between the closed-form solution **u** and the numerical solution gained by the algorithm made in the project, and that the deviation decreases with increasing number of investigated grid points n. To see how this deviation actually reacts on a change in number of grid points, which is directly related to the step length, consider the relative error ε_i given by

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{3.2}$$

in which v_i is the i'th element of the numerical solution **v** gained by the c++ code described in Sec. 2.2, and u_i is the i'th element of the closed-form-solution **u** calculated by the formula (3.1) with $x = (i+1)h$ as the relation between i and x for steplength h.

n	h	$\log(h)$	ε_i
10	0.090909	-1.04139	-1.1797
100	0.009901	-2.00432	-3.08804
1000	0.000999	-3.00043	-5.08005
10000	0.000099	-4.00004	-7.07934
100000	10^{-5}	-5	-8.888

Table 3.1. The table shows different relative errors ε_i for different n's corresponding to different steplengths h. The steplength h and logarithm to the steplength is calculated in Excel, whilst the relative error ε_i is calculated as in (3.2) using the c++ code. When the number of grid points n is increased to 100,000, the precision on the fifth digit of ε_i is lost in the c++ code, which is why only the first four digits of ε_i for $n = 100{,}000$ are shown in the table.

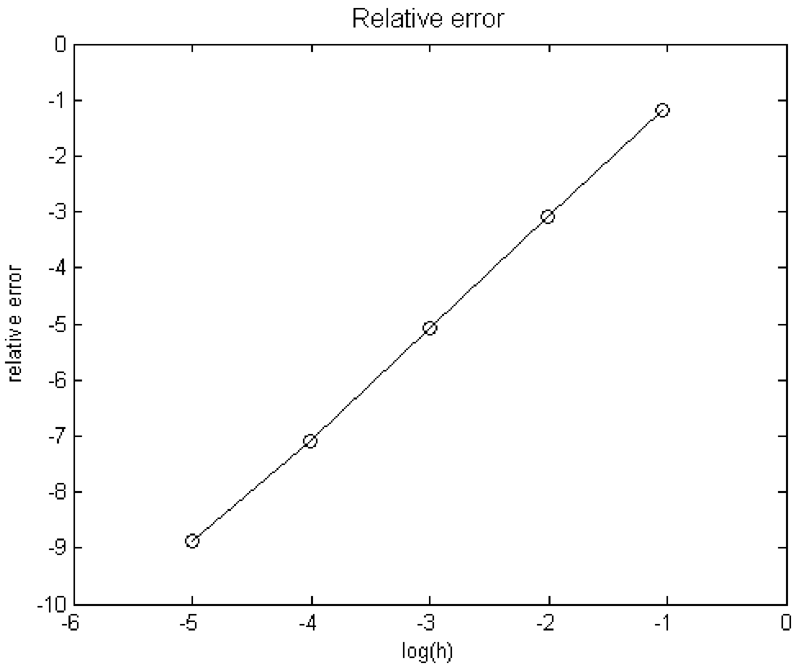

Figure 3.3. Plot of the relative error of the numerical solution to the closed-form solution as a function of the step length. The data point with the greatest relative error is for $n = 10$, while the data point for the smallest relative error is for $n = 10^5$. These data points are connected by a almost straight line with a slope of approximately 2.

In the graph given in Fig. 3.3, the data points form points on an almost straight line. Calculating the slope of the straight line from the ε_i and $\log(h)$ values for $n = 10$ and $n = 100$ gives

$$\text{slope} = \frac{\Delta\varepsilon_i}{\Delta\log(h)} = \frac{-3.08804-(-1.1797)}{-2.00432-(-1.04139)} \approx 1.98 \approx 2 \tag{3.3}$$

This is an expected result, since the total error ε_{total} goes as $\mathcal{O}(h^2)$, with h being the step length, due to the two contributions to the total error, namely the approximation error ε_{approx} and the round off error ε_{ro}. Since ε_{total} goes as $\mathcal{O}(h^2)$, the relation between $\varepsilon_i = \log(\varepsilon_{total})$ and $\log(h)$ goes as $\varepsilon_i = 2\log(h) + \text{constant}$.

One could, however, imagine that when n is further increased, the round off error will increase, and the linear dependence between ε_i and $\log(h)$ will disappear.

3.2 Comparing the Algorithm to the LU-decomposition Method

The elapsed time in the c++ code for both our algorithm and the LU decomposition method provided by the library Armadillo, can be computed by including the following code.

```
using namespace std;
#include "time.h"
int main()
{
   \\ declaration of variables
   ...
   clock_t start , finish;
   start = clock();
   // code
   finish = clock();
   ( (finish - start)/CLOCK_PER_SEC );
   ...
}
```

By adding these code lines to the algorithm made in this project, the precision of *start* and *finish* is too bad, and for all investigated n's in this project (see Tab. 3.1 for n values), the elapsed time is shown to be equal to zero. However, by adding these code lines around the Armadillo code lines for LU decomposition and solving the linear set of equations using the lower triangular and upper triangular matrix, we get an elapsed time of 0.02 s for $n = 1{,}000$, and the Armadillo code could not be run for $n = 10^5$.

This yields that our program is more efficient than the LU decomposition method for solving this specific problem.

CHAPTER 4

CONCLUSION

We were successfully able to formulate a code to solve the one dimensional Poisson equation. We compared the precision of numerical solution with the closed form solution for different step length of the variable x and found that a decrease in step length causes an increase in precision. It was found that the relation between the relative error and the logarithm to the step length was a straight line with a slope of 2.

We learned how to handle arrays and matrices in c++, and that the method for solving a problem should be chosen only after taking into account the nature of the problem as well as the number of flops.

We got introduced to the library armadillo and few of its functions that can make ones life easier and found that for this specific problem, it was more efficient to use the algorithm given in this project than solving the problem by the the LU decomposition method given by the Armadillo library. This is due to the fact that the Armadillo code for the LU decomposition method uses significantly more flops than our code, resulting in longer execution time for the LU decomposition method. This furthermore, lead to the case that the Armadillo code could not be run for 100,000 grid points, which our code could.

BIBLIOGRAPHY

[1] M. Hjorth-Jensen, “Computational physics - lecture notes fall 2015,” August 2015.

APPENDIX

A MATLAB CODE FOR PLOTTING NUMERICAL AND CLOSED-FORM SOLUTION

```
close all
clear all
clc
%Plot of nummerical approximation with n=10, n=100 and n=1000 and
%closed-form formula

%Import data gained from the cpp code

filename = 'Results.xlsx';
sheet = 4;
xlRange = 'B3:C12';

[v,T,vT] = xlsread(filename, sheet, xlRange);
x10=v(:,1);y10=v(:,2);

filename = 'Results.xlsx';
sheet = 5;
xlRange = 'B3:C102';

[v,T,vT] = xlsread(filename, sheet, xlRange);
x100=v(:,1);y100=v(:,2);

filename = 'Results.xlsx';
sheet = 6;
xlRange = 'B3:C1002';

[v,T,vT] = xlsread(filename, sheet, xlRange);
x1000=v(:,1);y1000=v(:,2);

%Define closed-form solution
n=1000;

h=1/(n+1);

x = 0:h:1;
y_exact = 1-(1-exp(-10))*x-exp(-10*x);
```

```
%Plot data and closed-form solution

figure
plot(x100,y100,'-.b',x1000,y1000,'m',x,y_exact,'--k')
legend('n=100','n=1000','closed-form solution')

xlim([0.2 0.3])
ylim([0.65 0.675])

title('Estimation of solution, zoom','FontSize',12)
xlabel('x')
ylabel('u(x)')
```

APPENDIX

B MATLAB CODE FOR PLOTTING THE RELATIVE ERROR

```
close all
clear all
clc
%Plot of nummerical approximation with n=10, n=100 and n=1000 and
%closed-form formula

%Import data and define closed-form solution

filename = 'Results.xlsx';
sheet = 7;
xlRange = 'C4:D8';

[v,T,vT] = xlsread(filename, sheet, xlRange);
h=v(:,1);error=v(:,2);

%Plot data and closed-form solution

figure
plot(h,error,'-ko')

xlim([-6 0])
ylim([-10 0])

title('Relative error','FontSize',12)
xlabel('log(h)')
ylabel('relative error')
```