Numerical integration using linear algebra

Anders P. Åsbø
(Dated: September 10, 2019)

derp

CONTENTS

I. INTRODUCTION

Numerical integration is an important cornerstone of computational physics, and as such it is important to understand its limits in terms of numerical precision and time spent computing. In this report I have looked at the specific case of numerically integrating a linear, second order differential equation as a system of linear equations, with the pretext of solving a one-dimensional variant of Poisson's equation.

II. FORMALISM

A. Underlying theory

If I have a charge distrobution $\rho(\vec{r})$, as a function of the position $\vec{r}$, Possion's equation gives the electrostatic potential Φ

$$\nabla^2 = -4\pi\rho(\vec{r}).$$

If I then assume both $\rho(\vec{r})$, and Φ to bi spherically symetric, the equation can be simplified to

$$\frac{1}{r^2}\frac{d\Phi}{dr}r^2\frac{d\Phi}{dr} = -4\pi\rho(r),$$

where $r = |\vec{r}|$. Substituting $\Phi(r) = \frac{1}{r}\phi(r)$ gives

$$\frac{d^2\phi}{dr^2} = -4\pi\rho(r),$$

which, for simplicity, can be written as

$$-u''(x) = f(x),$$

with $u = \phi$, $f = -4\pi\rho$, and $x = r$.

Having arrived at a simple formulation of the initial problem, I can now discretize it using the second order differential approximation

$$-\frac{u(x+h) + u(x-h) - 2u(x)}{h^2} = f(x).$$

Since I will be opperating with a discreete set of variables $x_i \in [0, 1]$ where $x_i = ih$ for $i = 1, 2, 3, ..., N$, I can write the afformentioned approximation as

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i,$$

where $u_i = u(x_i)$. The step-size is defined as $h = \frac{x_N - x_0}{N}$, and I impose the Dirichlet boundary conditions $u(0) = u(1) = 0$.

B. A general algorithm for square, tridiagonal matricies

The discretized approximation obtained in II.B can be rearranged to

$$-1u_{i-1} + 2u_i - 1u_{i+1} = h^2 f_i,$$

which describes a tridiagonal, $N \times N$-matrix

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & -1 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \dots & \dots & -1 & 2 \end{bmatrix},$$

and with the unkowns in a vector

$$\vec{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix},$$

and the values of $h^2 f_i = g_i$ in a vector

$$\vec{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_N \end{bmatrix},$$

I can express the integration problem as a matrix equation

$$A\vec{u} = \vec{g}.$$

For a general matrix

$$A = \begin{bmatrix} d_1 & a_1 & 0 & \dots & \dots & 0 \\ b_1 & d_2 & a_2 & \ddots & \ddots & \vdots \\ 0 & b_2 & d_3 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{N-1} \\ 0 & \dots & \dots & \dots & b_{N-1} & d_N \end{bmatrix},$$

A way to solve the matrix equation is by Gaussian elimination. I start by subtracting row 1 multiplied by $\frac{b_1}{d_1}$ from row 2, and continue by subtracting row 2 multiplied by $\frac{b_2}{d_2}$ from row 3. This continues all the way down such that a general algorithm can be written as

$$d_i^* = d_i - \frac{b_{i-1}a_{i-1}}{d_{i-1}^*}, \tag{1}$$

with the condition that $d_1^* = d_1$. The same pattern aplies to $\vec{g}$, such that

$$g_i^* = g_i - \frac{b_{i-1}g_{i-1}}{d_{i-1}^*}, \tag{2}$$

where $g_1^* = g_1$, such that we get a new vector $\vec{g}^*$ with the adjusted values. The two above algorithems constitutes the decomposition and forward substitution of the matrix, and gives $u_N = \frac{g_N^*}{d_N^*}$. The remaining values of u can then be calculated recursivly as

$$u_i = \frac{g_i^* - a_i u_{i+1}}{d_i^*}, \tag{3}$$

with $i = N-1, ..., 2, 1$. This algorithm constitutes the backward substitution.

III. IMPLEMENTATION

A. Implementing the general algorithm

To execute the numerical integration using (1), (2), and (3), I wrote the program "project.py"(A.1) which takes the number of step points N, as well as a label for the data files, as input from the user, and initializes an array of linearly spaced values of $x_i \in [0, 1]$, as well as the step-size. Furthermore, it calls a custom module "data_generator.py"(A.3), which generates saves an array of f_i values to a textfile. "data_generator.py"(A.3) also generates a seperate textfile containing an array of the analytical solution to the differential equation, since I am using the function

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x},$$

$$u'(x) = (1 - e^{-10}) - 10e^{-10x},$$

$$u''(x) = -100e^{-10x} = -f(x).$$

Both the textfile containing f_i, and the analytical solution are read by "project.py"(A.1), which stores them as arrays. The program also initializes empty arrays for $\vec{u}$, $\vec{d}^*$, and $\vec{g}^*$. The tridiagonal matrix is initialized as three arrays, one holding b_i, one holding d_i, and one holding a_i. Thus I avoid filling memory with the zero-elements.

"project.py"(A.1) continues, by setting the boundary conditions, and looping through (1), and (2) in one loop, and (3) in a second loop. For the decomposition and forward substitution, the factor $\frac{b_{i-1}}{d_{i-1}^*}$ is calculated before including it in (1), and (2), reducing the number of FLOPS from $6(N-1)$ to $5(N-1)$. Due to the boundary conditions, the backward substitution requires $3(N-2)$ FLOPS, resulting in a total of $8N - 11 \approx 8N$ FLOPS for sufficently large values of N.

Finally, "project.py"(A.1) saves the numerial solution $\vec{u}$ to a textfile, and plots it against the analytical solution.

B. Implementing a specialized version of the algorithm

Because the specific tridiagonal matrix in the problem has all diagonal elements equal to 2, and all non-zero, off-diagonal elements equal to -1, there is some optimization to be done, thus a specialized version of the integration program can be found in "project_specialized.py"(A.1).

Substituting in the known values, (3) can be reduced to

$$g_i^* = g_i + \frac{g_{i-1}^*}{d_i^*},$$

reducing the backward substitution to $2(N-2)$ FLOPS. Furthermore, the adjusted, diagonal elements can be expresed by an explicit formula

$$\frac{1}{d_i^*} = \frac{2i}{2(i+1)}.$$

Utilizing NumPy's elementwise opperations, "project_specialized.py"(A.1) precalculates all $\frac{1}{d_i^*}$ in parallell, reducing the decomposition and forward substitution to $2(N-1)$ FLOPS. This effectivly halves the number of FLOPS of the total algorithm to $4N-6 \approx 4N$, the arrays for the initial matrix elements were removed to as they are no longer needed.

IV. ANALYSIS

A. Plotting the genneral algorithm

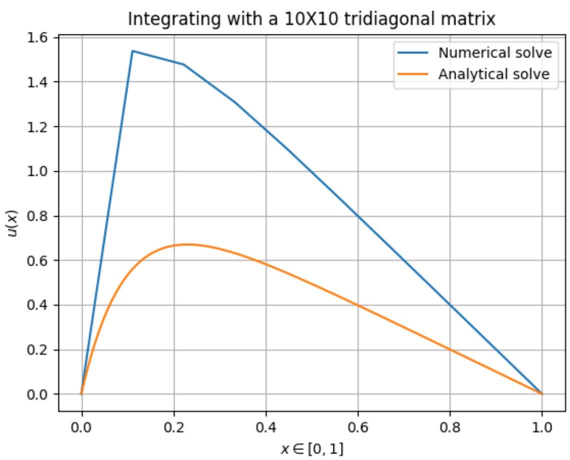

Figure 1. Plot of numerical, and analytical solution, using the general algorithm with $N = 10$.

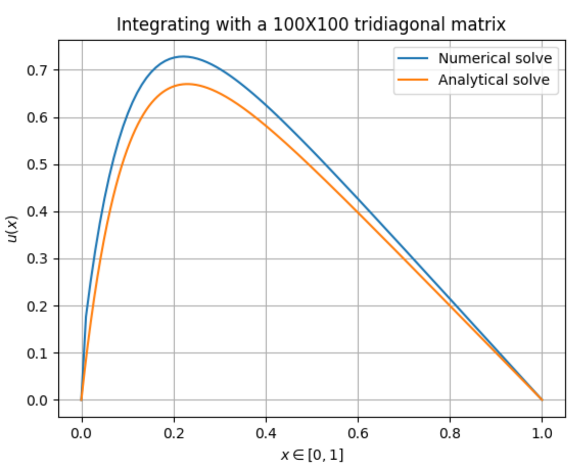

Figure 2. Plot of numerical, and analytical solution, using the general algorithm with $N = 100$.

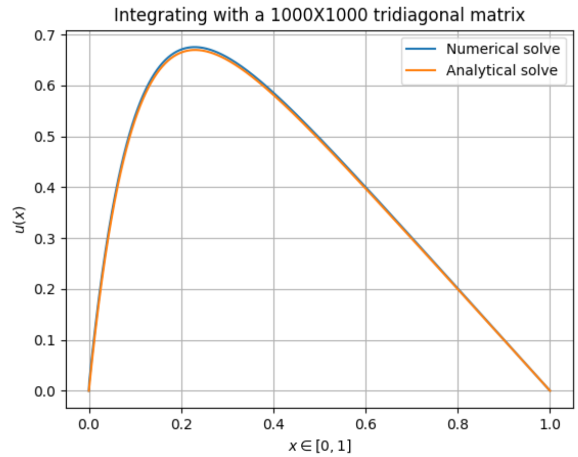

Figure 3. Plot of numerical, and analytical solution, using the general algorithm with $N = 1000$.

Using "project.py"(A.1), I plotted the numerical and analytical solutions to Possion's equation for a spherically symetrical charge distrobution, using a $N \times N$ matrix. From comparing figure 1 with figure 2, and figure 3, I see that the correspondence with the analytical values decreases as the number of matrix elements increases. It is rather unsuprising that a greater N, and therfore smaller step size h, leads to greater accuracy.

B. Benchmarks of the general and specializeed algorithms

N	Time [s]
1×10^1	$3.596\,200 \times 10^{-4}$
1×10^2	$6.853\,430 \times 10^{-4}$
1×10^3	$6.667\,227 \times 10^{-3}$
1×10^4	$6.290\,169 \times 10^{-2}$
1×10^5	$6.611\,870 \times 10^{-1}$

Table I. Table of time elapsed in seconds on algorithm for "project.py" with corresponding value of N

N	Time [s]
1×10^1	4.5119×10^{-5}
1×10^2	6.6955×10^{-5}
1×10^3	$7.452\,46 \times 10^{-4}$
1×10^4	$7.311\,16 \times 10^{-3}$
1×10^5	$7.114\,03 \times 10^{-2}$

Table II. Table of time elapsed in seconds on algorithm for "project_specialized.py" with corresponding value of N

I mesaured the time both "project.py", and "project_specialized.py" used to execute the general and specialized versions of the algorithm. This was done by saving the time on the CPU-clock directly before and after executing the two sequential loops that constitute the complete algorithm. Table I shows the time it took for "project.py", and table II shows the time it took for "project_specialized.py".

C. Error analysis of the specialized algorithm

$\log_{10} h$	$\log_{10} \epsilon_{i,max}$
-1	-0.0176522
-2	-0.842497
-3	-1.59133
-4	-2.41863
-5	-3.29336
-6	-4.19081
-7	-5

Table III. Table of $\log_{10} h$, and the corresponding $\log_{10} \epsilon_{i,max}$, where h is the step-size used, and $\epsilon_{i,max}$ is the coresponding maximum of the relative error

D. Comparison with LU-decomposition

N	Time [s]
1×10^1	$2.777\,14 \times 10^{-4}$
1×10^2	$1.737\,73 \times 10^{-3}$
1×10^3	$3.077\,32 \times 10^{-2}$
1×10^4	$6.144\,22$
1×10^5	NaN

Table IV. Table of time elapsed in seconds on LU decomposition and solve for "LUdecomp.py" with corresponding value of N

V. CONCLUSION

Appendix A: Program files

All code for this report was written in Python 3.6, and the complete set of files can be found at https://github.com/FunkMarvel/CompPhys-Project-1.

1. project.py

```
# Project 1 FYS3150, Anders P. Åsbø
# general tridiagonal matrix.
import data_generator as gen
import numpy as np
import os
import matplotlib.pyplot as plt
import timeit as time


def main():
    """Program solves matrix equation Au=f, using decomposition, forward
    substitution and backward substitution, for a tridiagonal, NxN matrix A."""
    init_data()  # initialising data

    # performing decomp. and forward and backward sub.:
    decomp_and_forward_and_backward_sub()

    save_sol()  # saving numerical solution in "data_files" directory.

    plot_solutions()  # plotting numerical solution vs analytical solution.

    plt.show()  # displaying plot.
```

```python
def init_data():
    """Initialising data for program as global variables."""
    global dir, N, name, x, h, anal_sol, u, d, d_prime, a, b, g, g_prime
    dir = os.path.dirname(os.path.realpath(__file__))  # current directory.

    # defining number of rows and columns in matrix:
    N = int(eval(input("Specify number of data points N: ")))
    # defining common label for data files:
    name = input("Label of data-sets without file extension: ")

    x = np.linspace(0, 1, N)  # array of normalized positions.
    h = (x[0]-x[-1])/N  # defining step-siz.

    gen.generate_data(x, name)  # generating dataanal_name set.
    anal_sol = np.loadtxt("%s/data_files/anal_solution_for_%s.dat" %
                          (dir, name))

    u = np.empty(N)  # array for unkown values.
    d = np.full(N, 2)  # array for diagonal elements.
    d_prime = np.empty(N)  # array for diagonal after decom. and sub.
    a = np.full(N-1, -1)  # array for upper, off-center diagonal.
    b = np.full(N-1, -1)  # array for lower, off-center diagonal.
    # array for g in matrix eq. Au=g.
    f = np.loadtxt("%s/data_files/%s.dat" % (dir, name))
    g = f*h**2
    g_prime = np.empty(N)  # array for g after decomp. and sub.


def decomp_and_forward_and_backward_sub():
    """Function that performs the matrix decomposition and forward
    and backward substitution."""
    # setting boundary conditions:
    u[0], u[-1] = 0, 0
    d_prime[0] = d[0]
    g_prime[0] = g[0]

    start = time.default_timer()  # times algorithm
    for i in range(1, len(u)):  # performing decomp. and forward sub.
        decomp_factor = b[i-1]/d_prime[i-1]
        d_prime[i] = d[i] - a[i-1]*decomp_factor
        g_prime[i] = g[i] - g_prime[i-1]*decomp_factor

    for i in reversed(range(1, len(u)-1)):  # performing backward sub.
        u[i] = (g_prime[i]-a[i]*u[i+1])/d_prime[i]
    end = time.default_timer()
    print("Time spent on loop %e" % (end-start))


def save_sol():
    """Function for saving numerical solution in data_files directory
    with prefix "solution"."""
    path = "%s/data_files/solution_%s.dat" % (dir, name)
    np.savetxt(path, u, fmt="%f")


def plot_solutions():
    """Function for plotting numerical vs analytical solutions."""
    x_prime = np.linspace(x[0], x[-1], len(anal_sol))

    plt.figure()
    plt.plot(x, u, label="Numerical solve")
    plt.plot(x_prime, anal_sol, label="Analytical solve")
    plt.title("Integrating with a %iX%i tridiagonal matrix" % (N, N))
    plt.xlabel(r"$x \in [0,1]$")
    plt.ylabel(r"$u(x)$")
    plt.legend()
    plt.grid()


if __name__ == '__main__':
    main()


# example run:
"""
$ python3 project.py
Specify number of data points N: 1000
Label of data-sets without file extension: num1000x1000
"""
# a plot is displayed, and the data is saved to the data_files directory.
```

2. project__specialized.py

```
# Project 1 FYS3150, Anders P. Åsbø
import data_generator as gen
import numpy as np
import os
import matplotlib.pyplot as plt
import timeit as time


def main():
    """Program solves matrix equation Au=f, using decomposition, forward
    substitution and backward substitution, for a Toeplitz, NxN matrix A."""
    init_data()  # initialising data

    # performing decomp. and forward and backward sub.:
    decomp_and_forward_and_backward_sub()

    save_sol()  # saving numerical solution in "data_files" directory.

    # plot_solutions()  # plotting numerical solution vs analytical solution.

    # plt.show()  # displaying plot.


def init_data():
    """Initialising data for program as global variables."""
    global dir, N, name, x, h, anal_sol, u, d, d_prime, a, b, g, g_prime
    dir = os.path.dirname(os.path.realpath(__file__))  # current directory.

    # defining number of rows and columns in matrix:
    N = int(eval(input("Specify number of data points N: ")))
    # defining common label for data files:
    name = input("Label of data-sets without file extension: ")

    x = np.linspace(0, 1, N)  # array of normalized positions.
    h = (x[0]-x[-1])/N  # defining step-siz.

    gen.generate_data(x, name)  # generating dataanal_name set.
    anal_sol = np.loadtxt("%s/data_files/anal_solution_for_%s.dat" %
                          (dir, name))

    u = np.empty(N)  # array for unkown values.
    s = np.arange(1, N+1)
    d_prime = 2*(s)/(2*(s+1))  # pre-calculating the 1/d_prime factors.
    f = np.loadtxt("%s/data_files/%s.dat" % (dir, name))
    g = f*h**2
    g_prime = np.empty(N)  # array for g after decomp. and sub.


def decomp_and_forward_and_backward_sub():
    """Function that performs the matrix decomposition and forward
    and backward substitution."""
    # setting boundary conditions:
    u[0], u[-1] = 0, 0
    g_prime[0] = g[0]
    start = time.default_timer()
    for i in range(1, len(u)):  # performing decomp. and forward sub.
        g_prime[i] = g[i] + g_prime[i-1]*d_prime[i-1]

    for i in reversed(range(1, len(u)-1)):  # performing backward sub.
        u[i] = (g_prime[i] + u[i+1])*d_prime[i-1]

    end = time.default_timer()
    np.savetxt("looptime%i" % N, np.array([end-start]))


def save_sol():
    """Function for saving numerical solution in data_files directory
    with prefix "solution"."""
    path = "%s/data_files/solution_%s.dat" % (dir, name)
    np.savetxt(path, u, fmt="%f")


def plot_solutions():
    """Function for plotting numerical vs analytical solutions."""
    x_prime = np.linspace(x[0], x[-1], len(anal_sol))

    plt.figure()
    plt.plot(x, u, label="Numerical solve")
    plt.plot(x_prime, anal_sol, label="Analytical solve")
    plt.title("Integrating with a %iX%i tridiagonal matrix" % (N, N))
```

```
    plt.xlabel(r"$x \in [0,1]$")
    plt.ylabel(r"$u(x)$")
    plt.legend()
    plt.grid()


if __name__ == '__main__':
    main()


# example run:
"""
$ python3 project_specialized.py
Specify number of data points N: 1000
Label of data-sets without file extension: opti1000x1000
"""
# a plot is displayed, and the data is saved to the data_files directory.
```

3. data_generator.py

```
# Project 1 FYS3150, Anders P. Åsbø.
import numpy as np
import os


def main():
    """This program calculates the log10 of the relative error for
    different data sets generated with "project_specialized.py", and saves
    it in a textfile table with the log10 of the step-size."""

    dir = os.path.dirname(os.path.realpath(__file__))
    name = input("Label data: ")
    # preping arrays:
    N = np.array([10, 100, 1000, 10000, 100000, 1000000, 10000000])
    h = np.empty(len(N))
    epsilon = np.empty(len(N))

    for i in range(len(N)):  # reading data from files:
        u_num = np.loadtxt("%s/data_files/%s%i.dat" % (dir, name, 1+i))
        u_anal = np.loadtxt("%s/data_files/anal_solution_for_%s%i.dat"
                            % (dir, name, 1+i))

        h[i] = np.log10(1/len(u_num))  # calculating log 10 of step-size.

        # calculating log10 of relative error:
        err = np.abs((u_num[1:-2]-u_anal[1:-2])/u_anal[1:-2])
        epsilon[i] = np.max(np.log10(err))

    # storing the results
    table = np.empty((len(N), 2))
    table[:, 0] = h
    table[:, 1] = epsilon
    np.savetxt("%s/data_files/error_table_%s.dat" % (dir, name), table,
               fmt="%f")


if __name__ == '__main__':
    main()

# example run:
"""
$ python3 erroranlaysis.py
erroranlaysis.py:17: RuntimeWarning: divide by zero encountered in log10
  epsilon[i] = np.max(np.log10(err))
erroranlaysis.py:16: RuntimeWarning: divide by zero encountered in true_divide
  err = np.abs((u_num[1:-2]-u_anal[1:-2])/u_anal[1:-2])
"""
# table of log10 of stepsizes, and max errors is saved to file
```

4. LUdecomp.py

```
# Project 1 FYS3150, Anders P. Åsbø
import scipy.linalg as sp
import data_generator as gen
```

```python
import numpy as np
import timeit as time
import os

dir = os.path.dirname(os.path.realpath(__file__))


def main():
    """Program uses LU-decomposition to solve Au=g"""

    # getting dimensions of matrix, and labeling data:
    global N
    N = int(eval(input("Give value for N: ")))
    name = "LUtest%i" % N

    # generating data:
    x = np.linspace(0, 1, N)
    h = (x[-1]-x[0])/N
    gen.generate_data(x, name)
    g = np.loadtxt("%s/data_files/%s.dat" % (dir, name))*h**2

    A = np.zeros((N, N))  # creating zero matrix.
    A = Afunc(A)  # populating the three diagonals.

    start = time.default_timer()  # timing solve.
    LU, piv = sp.lu_factor(A)  # decomp and forward sub.
    u = sp.lu_solve((LU, piv), g)  # backward sub.
    end = time.default_timer()
    print("Time spent on LU %g" % (end-start))  # printing elapsed time.

    # saving numerical solution:
    np.savetxt("%s/data_files%s.dat" % (dir, name), u, fmt="%f")


def Afunc(A):
    """Function that populates tridiagonal matrix with 2 along diagonal,
    and -1 as the non-zero, off-diagonal elements."""
    for i in range(N):
        for j in range(N):
            if i == j:
                A[i, j] = 2
            elif i == j+1:
                A[i, j] = -1
            elif j == i+1:
                A[i, j] = -1

    return A


if __name__ == '__main__':
    main()

# example run:
"""
$ python3 LUdecomp.py
Give value for N: 1e4
Time spent on LU 6.23319
"""
# data is saved to files.
```