QFredDet — Software Library for Computing Entropy of Continuous Variable Quantum Systems Using Fredholm Determinants

Marek Sawerwain[1[0000−0001−8468−2456]] and Joanna Wiśniewska[2[0000−0002−2119−3329]]

[1] Institute of Control & Computation Engineering, University of Zielona Góra, Licealna 9, Zielona Góra 65-417, Poland
M.Sawerwain@issi.uz.zgora.pl
[2] Institute of Information Systems, Faculty of Cybernetics, Military University of Technology, Gen. S. Kaliskiego 2, Warszawa 00-908, Poland
JWisniewska@wat.edu.pl

Abstract. Fredholm determinants play a vital role in mathematics and physics, particularly in the theory of integral equations theory and random matrix theory. Recently, they have been proposed as an appropriate measure (especially for infinite dimensional continuous quantum systems) of quantum entanglement, which is one of the basic quantum information resources applied to the construction of the quantum computational machines. While the most studies rely on sequential numerical methods to compute Fredholm determinants, parallel programming techniques can be also effective. In this paper, we present an approach that leverages multi-core processors and GPUs, combined with an appropriate quadrature scheme, and is implemented in Python for broad accessibility. We also demonstrate how the proposed routines can be used to compute the von Neumann entropy for superposition of Fock states.

Keywords: Fredholm determinants · entropy computation for quantum systems · parallel computations · numerical quadrature

1 Introduction

In general, the Fredholm determinant is a mathematical concept used in functional analysis and integral equations. It is an extension of the determinant to infinite-dimensional spaces, especially for integral operators. It has long served as a practical tool in fields such as multivariate statistics, electrical engineering, and finance. More recently, it has found increasing applications in the field of quantum physics and computing. Fredholm determinants enable the numerical evaluation of certain probability distributions in random matrix theory, in particular the distribution functions of the Gaussian unitary ensemble [2]. They are also related to the Hamiltonian of coupled Painlevé V systems [19]. We can also utilize Fredholm determinants in analysis of one-dimensional fermionic chains

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

[7]. Furthermore, they have been proposed as a potential measure of quantum entanglement [9].

In this article, we focus on the numerical calculation of Fredholm determinants. The function's implementation for computing the determinants has a cubic complexity but it requires values of quadrature (e.g. Gauss-Legendre or Clenshaw-Curtis) and the growing number of quadrature points causes significantly increasing time of computation, so we propose utilizing parallel approach in CUDA [13]. The solution presented in this work is based on Gauss-Legendre quadrature. It is implemented in Python using the Numba [11] and CuPy [14], [3] libraries to enable efficient approximate computation of the Fredholm determinant. To the best of the authors' knowledge, the implementation of a parallel version of numerical procedure for the Fredholm determinant has not been reported in the literature so far.

The structure of this manuscript is as follows: Sec. 1.1 introduces the notations, symbols, abbreviations, and definitions used throughout the article. Sec. 2 provides a short overview of Fredholm determinants and revisits its fundamental applications. Sec. 3 presents serial numerical procedures implementing the calculation of Fredholm determinants. Then, the computational complexity of the mentioned method is analyzed what allows exposing how parallel techniques may be applied in this area. More precisely, we show in Sec. 4, how to carry out the parallel reduction to obtain an acceleration of the calculations. Sec. 4.2 presents an example of using developed implementations for Python. Conclusions are contained in Sec. 5. Acknowledgments and References are the last parts of the paper.

1.1 Notations, symbols, and abbreviations

Before starting the presentation of our solution concerning the application of parallel processing techniques to the calculation of the values of Fredholm determinants, a summary of the notation and the most important abbreviation symbols, and acronyms used in this work are collected in Table 1.

2 Fredholm determinants

Fredholm's work [5] is undoubtedly the most important work that laid the foundations of modern analysis and operator theory. Erik Ivar Fredholm analyzed the solvability of the following equation of the second kind:

$$u(x) + z \int_a^b K(x,y)u(y)dy = f(x), \quad x \in (a,b), \tag{1}$$

where the function f and kernel K are assumed to be continuous functions. Fredholm also showed that recalled equation is uniquely solvable if the following determinant, today called Fredholm determinant,

$$d(z) = \sum_{k=0}^{\infty} \frac{z_n}{n!} \int_a^b \cdots \int_a^b \det \Big(K(t_p,t_q) \Big)_{p,q=1}^{n} dt_1 \dots dt_n, \tag{2}$$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

Table 1. Some symbols, notations, sets and functions used in the paper

Notation	Description
Q	quantum state, $Q \in E(\mathcal{H})$
$d(z)$	Fredholm determinant
$d_Q(z)$	approximation of Fredholm determinant
$K(x, y)$	kernel function for quadrature
$E_2(0, s)$	the probability that an interval of length s does not contain an eigenvalue of the Gaussian unitary ensemble
z	a complex or real number
δ_{ij}	Kronecker delta, $\delta_{ij} = 0$ if $i \neq j$ and $\delta_{ij} = 1$, if $i = j$,
$T_{d_q(z)}(K, z, a, b, m)$	computational complexity of Fredholm determinant routine
vEN(∞)	von Neumann entropy
vENH(Q)	von Neumann entropy calculated by the Fredholm determinant
vrENH(Q)	renormalized von Neumann entropy calculated by the Fredholm determinant
$\mathbb{R}$, $\mathbb{C}$, $\mathbb{N}$	sets of real, complex, and integer numbers,
i, j, k	integer numbers usually used as indices
m	the number of quadrature points
$1 \ldots n$	means the sequence of $1, 2, 3, \ldots, n$

for $z \in \mathbb{C}$ is a function and $d(z) \neq 0$. Very often $d(z)$ is written in the following form:

$$d(z) = \det\left(I - zK \restriction_{L^2(a,b)}\right). \tag{3}$$

Fredholm's work is utilized in many contemporary areas of modern science not only in the theory of compact operators but also in the physics atomic collision theory, inverse scattering, renormalization of quantum field theory, random matrix theory, combinatorial growth processes.

In [8,9], we demonstrated that the Fredholm determinant can be applied to extend the von Neumann entropy formula, derived using the theory of Fredholm determinants, in particular their standard regularization methods. The approach advocated in these works suggests employing techniques from the theory of Fredholm-type integral equations, including approximate methods, to develop controllable numerical procedures for computing Schmidt data associated with the Schmidt decomposition of solutions to the Schrödinger equations. It is also possible to show that von Neumann entropy can be expressed with the Fredholm determinant to cover the space of all, mixed states including as well, quantum states, which in the discussed two-particle system is the set of all non-negative trace class linear operators Q acting in the space $\mathcal{L}_2(\mathbb{R}^6)$ and obeying the condition $\mathrm{Tr}\,(Q) = 1$ (where $\mathrm{Tr}\,(\cdot)$ is standard trace operator). This formulation naturally enables a number of new applications, e.g. in the analysis of quantum key distribution [16].

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

2.1 Fredholm determinants for computing the von Neumann entropy of quantum continuous variables states

Based on the results of [8], for a Hilbert space $\mathcal{H}$ of infinite dimension, $\dim(\mathcal{H}) = \infty$, and for $Q \in E(\mathcal{H})$, one can show that the von Neumann entropy of infinite-dimensional states takes infinite values:

$$\mathrm{vEN}(\infty) = \{Q \in E(\mathcal{H}) : \mathrm{Tr}\,(-Q \log Q) = \infty\}, \tag{4}$$

and moreover that this set is dense in $E(\mathcal{H})$ with respect to the L_1 topology.

However, for the complement set, i.e. for $Q \in \mathrm{vEN}(\infty)^c$ such that $\mathrm{tr}(-Q \log Q) < \infty$, one can provide a representation of the form:

$$(Q^{-Q} - \mathbb{I}) \in L_1(\mathcal{H}), \tag{5}$$

which allows one to prove that

$$\mathcal{D}(Q) = \det(\mathbb{I}_{\mathcal{H}} + \mathfrak{f}(Q)), \tag{6}$$

where $\mathfrak{f}(Q) = Q^{-Q} - \mathbb{I}$. It can be shown that $\mathfrak{f}(Q)$ is trace-class, and furthermore that

$$\mathrm{vENH}(Q) = -\mathrm{Tr}\,(Q \log Q) = \log \mathcal{D}(Q). \tag{7}$$

Referring again to [8], one can show that for any state $Q \in E(\mathcal{H})$ the condition $(Q^{-Q} - \mathbb{I}) \in L_2(\mathcal{H})$ holds, which leads to the definition of the normalized von Neumann entropy:

$$\mathrm{vrENH}(Q) = \mathrm{Tr}\left(-Q \log Q + (Q^{-Q} - \mathbb{I})\right), \tag{8}$$

which is continuous and finite with respect to the $L_2(\mathcal{H})$ topology.

Remark 1. Numerical counterparts of formulas (6) and (8) naturally require an appropriate choice of the kernel function K. This will be demonstrated by means of an example presented in Sec. 4.4.

3 Numerical evaluation of Fredholm determinants

In [2], a value of the determinant $d(z)$ (Eq. 2) is calculated using the determinant for the matrix sized $m \times m$ as Nyström equation $d_Q(z)$:

$$d_Q(z) = \det\left(\delta_{ij} + z w_j K(x_i, x_j)\right)_{i,j=1}^{m}. \tag{9}$$

Wherein, if the coefficients w_j for a given quadrature are positive, the symmetrical variant of the determinant $d_Q(z)$ should be applied, i.e.:

$$d_Q(z) = \det\left(\delta_{ij} + z\sqrt{w_i} K(x_i, x_j)\sqrt{w_j}\right)_{i,j=1}^{m}. \tag{10}$$

Calculating $d_Q(z)$ needs the quadrature, however, according to [2] utilizing the Gauss–Legendre [18] or Curtis–Clenshaw [17] quadrature, we can generate a code for Matlab [12] or Octave [4] as a short function:

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

```
(1) function d = FredholmDet(K,z,a,b,m)
(2)     [w,x] = QuadratureRule(a,b,m);
(3)     w = sqrt(w);
(4)     [xi,xj] = ndgrid(x,x);
(5)     d = det(eye(m)+z*(w'*w).*K(xi,xj));
```

which allows solving Eq. 10 and is characterized by the complexity $O(m^3)$ (this results from the complexity of calculating the value of the determinant with the function `det`). A more detailed analysis of the computational complexity, that we want to conduct now, allows us to directly indicate the dependencies between the data and show that the parallel computational techniques may be fully utilized to obtain a significant acceleration of computations.

We utilize Python as the main programming language in this article, so the FredholmDet function's form is depicted in Fig. 1:

```
(1) def fredholm_det(K, z, a, b, m):
(2)     w,x=quadrature_rule(a,b,m)
(3)     w=numpy.sqrt(w)
(4)     xi,xj=numpy.meshgrid(x, x, indexing='ij')
(5)     d=numpy.linalg.det( numpy.eye(m)+z*numpy.outer(w,w)*K(xi,xj) )
(6)     return d
```

Fig. 1. Function fredholm_det implemented Python. It requires the quadrature_rule function to calculate a quadrature and kernel K function. At the beginning of each line, the ordinal number is placed to simplify referring to fragments of the source code

This function is a counterpart of the original code given in [2] and also requires a function calculating quadrature: `quadrature_rule`. All vector and matrix operations are implemented by the use of the Numerical Python package – NumPy [10].

3.1 Computational complexity

In general, the complexity of `fredholm_det` function, using symbols as in the implementation for Matlab/Octave, may be expressed as:

$$T_{d_q(z)}(K,z,a,b,m) = T_Q(a,b,m) + T_{\sqrt{w}}(w) + T_g(x) \\ + T_{\det}(m,m) + \Big(T_{eye}(m) + T_{zw}(z,w) + T_K(xi,xj)\Big). \quad (11)$$

Particular markings T are assigned to the following code lines:

- $T_Q(a,b,m)$ – determines a complexity of line (2), i.e. the complexity of `quadrature_rule` function which calculates weights and quadrature's points

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

in a range a, b with m points. If the Gauss-Legendre quadrature is applied, as in this work, then:

$$T_Q(a, b, m) = O(m^3). \tag{12}$$

- $T_{\sqrt{w}}(w)$ – line (3) corresponds to computing a root of quadrature's weights with a linear complexity depending on the number of points m: $O(m)$.
- $T_g(x)$ – line (4) refers to the `meshgrid` procedure which calculates a grid with the complexity $O(m^2)$.
- a complexity of line (5) has to be described as

$$T_{\det}(m, m) + (T_{eye}(m) + T_{zw}(z, w) + T_K(xi, xj)), \tag{13}$$

  so in general we have $O((T_K(xi, xj) \cdot O(m^2)) + O(m^3))$, where calculating the kernel function is marked as T_K and this function may have a unit execution time but it is executed for each point in the grid sized $m \times m$. Additionally, it is dependent on the kernel function, therefore its complexity is not less than a quadratic function but a specific form of the kernel function can increase it.

Remark 2. The linear algebra operations related to the calculation of the argument T_{zw} can be expressed as a quadratic function. The entire calculation of the determinant is described as a polynomial of cubic degree. In general, we have here the complexity described by a third-degree polynomial, where the argument is T_K, i.e. the complexity of the kernel function. The kernel function in the case of more complicated kernel forms, what we emphasize here once again, can naturally only increase the degree of the polynomial, or even change the complexity class.

Assuming that the kernel function for a point on the grid has a unit complexity, the total complexity $T_{d_q(z)}(K, z, a, b, m) = O(m^3)$. This is a convenient complexity from a practical point of view what is also verified by the example of the procedure's NumPy version working time given in Sec. 4 (in subsection 4.1). □

4 Parallel computations of Fredholm determinant

In this part of our work, we introduce function `fredholm_det` realized in Python using the Numba and CuPy packages. Before the analysis of parallel implementation in Sec. 4.2, we present obtained results for the version based on NumPy package in Sec. 4.1.

After analyzing the computational complexity in Sec. 3.1, three areas in the `fredholm_det` function can be identified where parallel processing can be applied:

- calculation of quadrature weights and points,
- preparing matrix $m \times m$,
- computing of matrix determinant.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

All three steps can be executed in parallel, with a significant reduction in computation time expected, especially for larger numbers of quadrature points. In the proposed solution, we use the Numba package to implement the second task, while the procedures from the first and third steps utilize the CuPy package to compute the quadrature points and, ultimately, calculate the determinant value for the prepared matrix.

4.1 The basic version performance

It should be emphasized that `fredholm_det` function is characterized by sufficient efficiency $O(m^3)$ where m is the number of quadrature's points. We can easily show this, even without direct usage of parallel computing, by calculating the $E_2(0,s)$ value, i.e. the probability that an interval of length s does not contain an eigenvalue of the Gaussian unitary ensemble, what is given by the Fredholm determinant:

$$E_2(0,s) = \det\left(I - A_s \restriction_{L^2(0,s)}\right), \tag{14}$$

and

$$A_s u(x) = \int_0^s \frac{sin(\pi(x-y))}{\pi(x-y)} u(y)dy, \tag{15}$$

where the sine kernel $A_s u(x)$ was used. In the following part of the article, we apply this quantity as a synthetic benchmark to evaluate the performance of the proposed solution.

Results referring to the performance of procedure shown in Fig. 1 are presented in Tbl. 2. The experiments were performed on two modern processors (AMD Ryzen 9 7950X (CPU 1), Intel Xeon W-2245 (CPU 2)) and in both cases, the dependence of the operating time on the number of quadrature points is visible. Therefore, the presented calculations also show a very good convergence of the function `fredholm_det`.

It should be emphasized that increasing the number of quadrature intervals, what is not justified in the case of probability described by $E_2(0,s)$, is characterized in this case by exponential complexity [1], [2] because of the approximation of the Fredholm determinant value by the quadrature. Therefore, selecting e.g. 8192 quadrature points significantly increases the running time of the entire `fredholm_det` procedure, to an average of ≈250 seconds for the Intel Xeon W-2245 processor. The procedure of determining the quadrature points (in this case Gauss-Legendre approach) takes the most of the time: about ≈240 seconds. The remaining two computational tasks, i.e. preparing the matrix $m \times m$ and calculating its determinant, take only about 3.1–3.2 seconds.

In this case, it can be seen that if a given computational task related to the determinant $d_Q(z)$ requires a larger number of quadrature points, the computational time in practice, despite the use of modern processors and the NumPy package, requires several minutes of work. It is therefore reasonable to look for further possibilities of shortening the operation time, e.g. by using GPU systems.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

Table 2. Performance characteristics of the basic Numpy version of the calculation for the basic example with probability $E_2(0, s)$, time in seconds (s). The column m describes the number of quadrature points, CPU 1 is AMD Ryzen 9 7950X processor (16 physical cores), and CPU 2 is Intel Xeon W-2245 (8 physical cores). In both cases, Python 3.11.9 and Windows 11 were used. The column Value provides probability values $E_2(0, s)$ for $s = 0.1$.

m	CPU 1	CPU 2	Value
5	0.00011020s	0.00079250s	0.9002570308592994
10	0.00011340s	0.00102080s	0.9002570308593463
20	0.00017330s	0.00103600s	0.9002570308593464
50	0.00056660s	0.00360290s	0.9002570308593462
100	0.00844520s	0.02522680s	0.9002570308593477
250	0.06502550s	0.21752530s	0.9002570308593448
500	0.28021040s	0.73422160s	0.9002570308593484
1000	0.84063120s	2.20266480s	0.9002570308593508
2000	4.87566096s	12.7754558s	0.9002570308593361
4000	40.5951504s	76.6527348s	0.9002570308593187
8000	243.3006925s	252.954025s	0.9002570308593004

4.2 Parallel variant of Fredholm determinant implementation

An analysis the procedure `fredholm_det` (Fig. 1) reveals that line (2) and the procedure of calculating the determinant are characterized by complexity $O(m^3)$, whereas, the remaining code elements can be described by complexity $O(m^2)$. These mentioned three components may be implemented in parallel what will help to shorten the calculation time.

The first element that can be improved by using multiple computing cores of GPUs is the code from line (5), marked in bold (the NumPy package is represented by np):

$$(5) \quad \mathrm{d} = \mathrm{np.linalg.det}(\mathbf{np.eye(m)} + \mathbf{z} * \mathbf{np.outer(w, w)} * \mathbf{K(xi, xj)}),$$

which is responsible for preparing the matrix $n \times m$ from which we calculate the determinant. Since the number of quadrature points that can be used is limited in the practical case to several thousand, currently available GPU systems offering, for example, 16384 computing cores for the consumer NVIDIA RTX Geforce 4090 model (24 GB VRAM, 16384 CUDA Cores, clock approx. 2.5 GHZ), allow for full coverage of the computational task, what translates into a constant computational time for the number of quadrature points from 1000 to several thousand. In general, the implementation method uses the concept of a computational grid available in CUDA technology [13]. Fig. 3 presents the applied construction of a computational grid.

Importantly, even direct using of the computational grid, e.g. for $m = 4096$, means operating time of the order of 0.00040799 seconds (it should be noted that these times refer to an already compiled computational kernel; the first

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

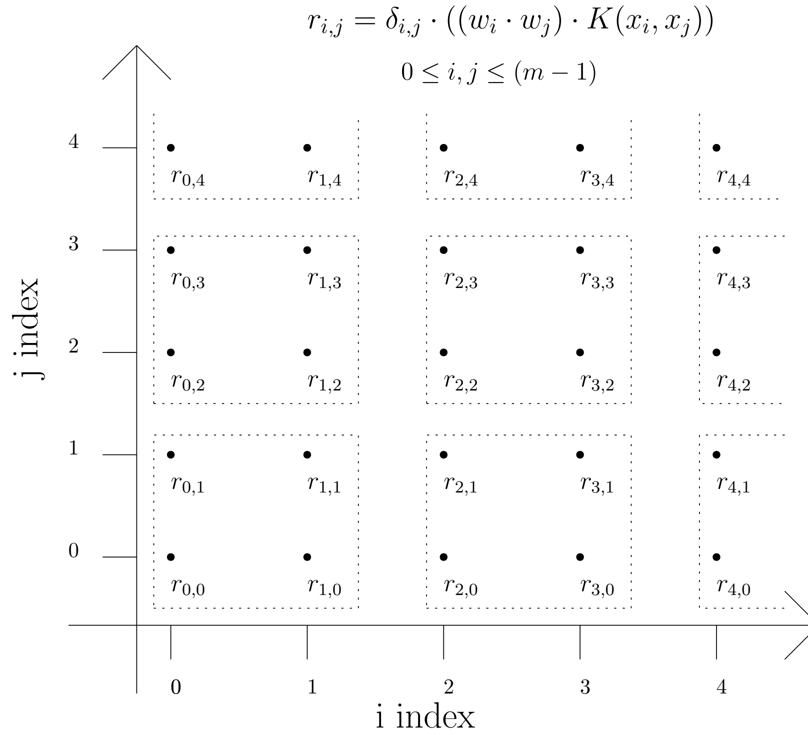


Fig. 2. An example configuration of the computational grid for executing the computation step from line (5) in the code from Fig. 1. Particular points represent elements of a matrix sized $m \times m$, and these coordinates are generated by the computational grid mechanism available in CUDA technology. All computations for individual points are performed in parallel within blocks, with each computational block outlined using dotted frames. Naturally, the execution requires weight vectors (variable w) and quadrature points (variable x) of length m. Although the direct approach to grid construction, as shown in the figure, it does not utilize shared memory (which could store temporary computations or data related to the vectors w and x), but still provides a significant practical reduction in computation time. For a large number of points – on the order of hundreds or even thousands – this approach can accelerate matrix preparation for determinant computation by a factor of two to three thousand times

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

execution of a CUDA computational procedure is always significantly longer), compared to about three seconds for the NumPy code, using double precision numbers. This means a speedup of over 5000 times for this single computational step.

The source code for $\mathbf{np.eye(m)} + \mathbf{z} * \mathbf{np.outer(w, w)} * \mathbf{K(xi, xj)}$, implemented with the use of the Numba package, is relatively concise and selected parts are shown in Fig. 3. The function `fredholm_step_line5` fully realizes the mentioned fragment, where the role of the computational kernel K e.g. from Eq. 2 is played by the device function d_ksinc.

The implementation code fragments also provide the description of the kernel function: `d_ksinc`. The form of the kernel is:

$$K(x, y) = \mathrm{sinc}(\pi(x - y)). \tag{16}$$

It can be seen that its form in the implementation is not direct – although the Numpy package supports many special functions, the Numba package, which supports GPUs, unfortunately does not provide the definition of function `sinc`. This inconvenience does not affect the final performance.

Remark 3. The source code of the snippet in Fig. 3 may be found at [15].

4.3 Performance of parallel implementation

Fig. 4 presents the results of calculations for different quadrature values. The GPU performance exceeds the CPU calculations for the number of points above 600 points. The reduction in calculation time for 4000 quadrature points is already significant, because it is twelvefold, and for 8000 it is already over fiftyfold. It should also be noted that the calculations are performed using double precision numbers and the graphics card used, i.e. the consumer model NVIDIA RTX 4090, offers significantly lower performance ≈ 1.2 TFLOPS for double precision calculations than for single precision (≈ 82 TFLOPS). However, the number of available computing cores 16384, allows achieving high acceleration values for calculating the Fredholm determinant using many quadrature points. But, up to about 500 square points, it is easy to observe that the acceleration value is not better than a traditional processor, which is related to the fact that a traditional processor still has a much higher clock speed than GPU cores, which with less computation, GPU calculations will be significantly slower for small computing tasks.

The final acceleration values are influenced by the procedures for determining the eigenvalues during the calculation of the Gauss-Legendre quadrature points and the process of computing the matrix determinant value. Naturally, the time of executing these tasks determines the total time of the procedure calculating the value of the Fredholm determinant approximation, e.g. for $m = 8000$ the values (in fractions of seconds) that can be obtained are:

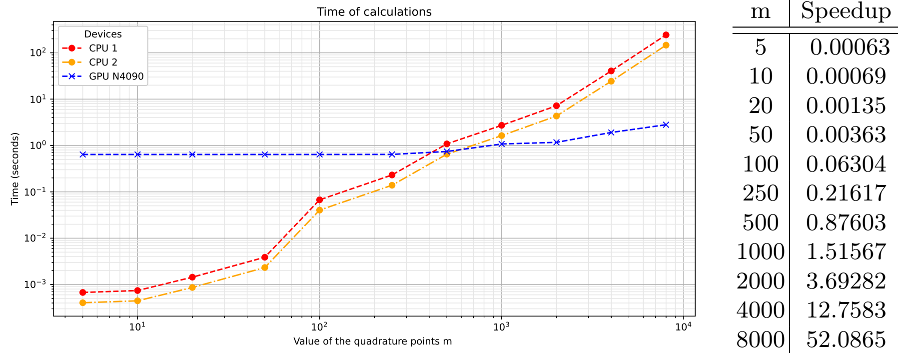
- time of calculating quadrature points: 2.6034144000 s.,
- matrix preparation: 0.00045769 s.,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

```
@cuda.jit(device=True, inline=True)
def d_ksinc(x,y):
    tmp=numpy.pi*(x-y)
    if tmp==0.0:
        return 1.0
    else:
        r=numpy.sin(numpy.pi*tmp)/(numpy.pi*tmp)
    return r

@cuda.jit
def fredholm_step_line5(w, x, m, R):
    i, j = cuda.grid(2)
    if i<m and j<m:
        wsi=math.sqrt(w[i])
        wsj=math.sqrt(w[j])
        tmp = (wsi*wsj) * d_ksinc(x[i], x[j])
        R[i,j] = d_kronecker_delta(i,j) - tmp

...

R = numpy.zeros(m*m).reshape(m,m)

w = numpy.ascontiguousarray(w); x = numpy.ascontiguousarray(x)
R = numpy.ascontiguousarray(R)

w_d = cuda.to_device(w) ; x_d = cuda.to_device(x)
R_d = cuda.to_device(R)

# preparation the computation grid
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(m / threadsperblock[0])
blockspergrid_y = math.ceil(m / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

...
fredholm_step1[blockspergrid, threadsperblock](w_d, x_d, m, R_d)
...
```

Fig. 3. Code snippets for computing the Fredholm determinant, specifically the fredholm_step_line5 function, which represents the matrix preparation in line (5) of the code from Fig. 1. The provided code was developed in Python using the Numba package

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49



m	Speedup
5	0.00063
10	0.00069
20	0.00135
50	0.00363
100	0.06304
250	0.21617
500	0.87603
1000	1.51567
2000	3.69282
4000	12.7583
8000	52.0865

Fig. 4. Computation times (the graph has logarithmic axes) of the values for $E_2(0,s)$ for two processors CPU 1 and CPU 2 (identical to the "Basic Numpy" version and the NVIDIA RTX 4090 graphics card). The acceleration or deceleration values (between CPU 2 and GPU) for different numbers of quadrature points m are also given

– calculating of matrix determinant: 0.1987810 s.

what, compared to 250 seconds for a traditional processor, is a significant benefit of using GPUs in the context of calculating the Fredholm determinant values.

4.4 Entropy calculation for superposition of Fock states

In [6], authors propose calculating von Neumann entropy values for superposition of $|0\rangle$ and an arbitrary Fock state $|m\rangle$. Mentioned case is a binary superposition which may be expressed as:

$$|\psi\rangle = \frac{|0\rangle + z|m\rangle}{\sqrt{1+z^2}} \tag{17}$$

where $z \in \mathbb{R}$. State $|\psi\rangle$ is subjected to unitary operation U which amplifies the Fock state $|m\rangle$:

$$U = e^{-v} e^{-\tau \hat{a}^\dagger \hat{b}^\dagger} e^{-v(\hat{a}^\dagger \hat{a} + \hat{b}^\dagger \hat{b})} e^{\tau^* \hat{a}\hat{b}}, \tag{18}$$

with $v = \ln\cosh|\xi|$ and $\tau = \frac{\xi}{|\xi|}\tanh|\xi|$. Finally, the entropy value may be defined as:

$$S(z) = \frac{1}{1+z^2} S_0 + \frac{z^2}{1+z^2} S_m. \tag{19}$$

Symbols S_0 and S_m stand for counterparts of states $|0\rangle$ and $|m\rangle$ in convex combination of mentioned superposition. That means the entropy here is a monotonically increasing function depending on the value of ξ.

Applying the numerical approach described in Sec. 3 to calculating the von Neumann entropy value requires specifying the kernel function that will be passed to the quadrature function. In the case of the Fock state, this can be a kernel of the form:

$$K(x,y) = \frac{\tanh(x-y)}{x+y}. \tag{20}$$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

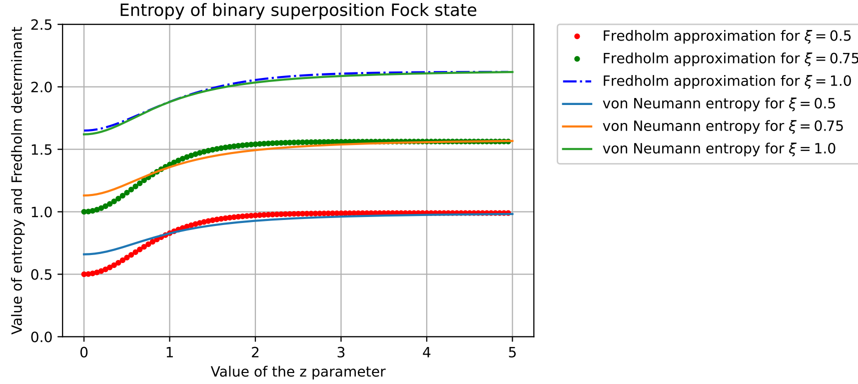


Fig. 5. Approximations of the von Neumann entropy values for the superposition of $|0\rangle$ and an arbitrary Fock state $|m\rangle$. The graph shows the values obtained using Eq. 19 and approximations calculated applying the Fredholm determinants. As one can see, as the parameter z increases, we obtain increasingly better approximations, and the overall behavior of the approximation correctly imitates the dynamics of changes in the von Neumann entropy value calculated by the formula Eq. 19

This allows us to obtain approximations, which are visualized in Fig. 5. Naturally, it must be noted that this is an approximation, so the value of the parameter ξ takes different values here, e.g., for $\xi = 0.75$. Calculating the value of the Fredholm determinant comes down to the following line of code:

```
max_r=5; val_of_z = entropy_with_fred_det( z , 1.5, 2, max_r)
```

where 1.5 acts as the ξ parameter and 2 is the number of quadrature intervals. The Python code for the entropy calculation function in this case is as follows:

```
def entropy_with_fred_det_xi075(r, z, m, max_r):
    a=0
    b=r*2
    w,x=gauss_legendre_quadrature(a,b,m)
    w = numpy.sqrt(w)
    xi,xj = numpy.meshgrid(x, x, indexing='ij')
    ker_mat = numpy.nan_to_num( kernel_fnc(r, xi, xj), 0.0)
    d = numpy.linalg.det( numpy.eye(m)
                          + z * numpy.outer(w,w) *  ker_mat )

    return d
```

5 Summary

The paper presents an implementation of a computational procedure based on Gauss-Legendre quadrature within the Python ecosystem, using the NumPy and

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

CuPy libraries to enable GPU-accelerated computations. The proposed approach is applied to the calculation of the von Neumann entropy for continuous-variable quantum states. To the best of the authors' knowledge, this constitutes one of the first GPU-based implementations in this context. In contrast, the work of [2] provides general-purpose routines designed for CPU architectures. The achieved accelerations enable practical reduction of the computational time for those cases where many quadrature points must be used. It should also be emphasized that the use of the Numba and CuPy packages offers a relatively easy transfer of the problem of calculating the approximation of the Fredholm determinant to the Python language. In the case of a smaller number of square points, the NumPy package for a traditional processor (CPU) can also be successfully used in the calculation of the approximate value of the Fredholm determinant. Other numerical approaches can also be employed; however, they typically require the computation of eigenvalues. An advantage of the CuPy library is that this functionality is supported directly on the GPU.

Although only one quadrature was briefly presented in this article, the availability of the function calculating the inverse fast Fourier transform allows us to implement the approximation of the Fredholm determinant based on the Clenshaw-Curtis quadrature in a similar way. It is also worth emphasizing that the proposed solution is based on free and fully open Python environments and packages supporting calculations.

The application of techniques based on the Fredholm determinant also makes it possible, at the theoretical level, to formulate appropriate numerical procedures for computing the values of Schmidt coefficients in systems where one of the subsystems is described by a continuous, infinite-dimensional Hilbert space.

Acknowledgments. This work was co-financed by Military University of Technology under research project UGB 531-000091-W500-22 and by a subsidy for research projects in Technical Computer Science and Telecommunication discipline in University of Zielona Góra for year 2026.

Disclosure of Interests. All authors declare that they have no conflicts of interest.

References

1. Bornemann, F.: Numerical evaluation of Fredholm determinants and Painlevé transcendents with applications to random matrix theory. In: Workshop on Integrable Systems and Scientific Computing, 15 - 20 June 2009, ICTP (2009)
2. Bornemann, F.: On the numerical evaluation of Fredholm determinants. Math. Comp. **79**, 871 – 915 (2010)
3. CUPy: Cupy – Numpy & SciPy for GPU (2025), https://cupy.dev/
4. Eaton, J.W., Bateman, D., Hauberg, S., Wehbring, R.: GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations (2020)
5. Fredholm, I.: Sur une classe d'équations fonctionnelles. Acta Mathematica **27**(none), 365 – 390 (1903). https://doi.org/10.1007/BF02421317

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49

6. Gagatsos, C.N., Karanikas, A.I., Kordas, G., Cerf, N.J.: Entropy generation in gaussian quantum transformations: applying the replica method to continuous-variable quantum information theory. npj Quantum Information **2**(1), 15008 (Feb 2016). https://doi.org/10.1038/npjqi.2015.8
7. Gamayun, O., Lychkovskiy, O., Caux, J.S.: Fredholm determinants, full counting statistics and Loschmidt echo for domain wall profiles in one-dimensional free fermionic chains. SciPost Phys. **8**(3), 036 (2020). https://doi.org/10.21468/scipostphys.8.3.036
8. Gielerak, R.: Renormalized von neumann entropy with application to entanglement in genuine infinite dimensional systems. Quantum Information Processing **22**, 311 (2023). https://doi.org/10.1007/s11128-023-04059-1
9. Gielerak, R., Wiśniewska, J., Sawerwain, M.: Infinite-dimensional quantum entropy: The unified entropy case. Entropy **26**(12) (2024). https://doi.org/10.3390/e26121070
10. Harris, C.R., Millman, K.J., et al.: Array programming with NumPy. Nature **585**(7825), 357–362 (Sep 2020). https://doi.org/10.1038/s41586-020-2649-2
11. Lam, S.K., Pitrou, A., Seibert, S.: Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. pp. 1–6 (2015)
12. MATLAB: 9.7.0.1190202 (R2019b). The MathWorks Inc., Natick, Massachusetts (2018)
13. NVIDIA: Cuda, release: 12.8 (2025), https://developer.nvidia.com/cuda-toolkit
14. Okuta, R., Unno, Y., Nishino, D., Hido, S., Loomis, C.: Cupy: A numpy-compatible library for nvidia gpu calculations. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS) (2017)
15. Sawerwain, M., Wiśniewska, J., Wróblewski, M., Gielerak, R.: Source code of examples of fredholm determinants calculations (2025), https://github.com/qMSUZ/EntDetector/tree/main/examples-fredholm
16. Szczepanik, W., Niemiec, M.: Optimizing routing in quantum key distribution networks using the artificial fish swarm algorithm. International Journal of Applied Mathematics and Computer Science **35**(4), 667–675 (2025). https://doi.org/10.61822/amcs-2025-0047
17. Trefethen, L.N.: Is Gauss quadrature better than Clenshaw–Curtis? SIAM Review **50**(1), 67–87 (2008). https://doi.org/10.1137/060659831
18. Waldvogel, J.: Fast construction of the Fejér and Clenshaw–Curtis quadrature rules. Bit Numer Math **46**, 195 – 202 (2006). https://doi.org/10.1007/s10543-006-0045-4
19. Xu, S.X., Zhao, S.Q., Zhao, Y.Q.: On the fredholm determinant of the confluent hypergeometric kernel with discontinuities. Physica D: Nonlinear Phenomena **461**, 134101 (2024). https://doi.org/https://doi.org/10.1016/j.physd.2024.134101

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_49