Developing of Scientific Software Applications in Python. I. Transformation of Hubbard Hamiltonian into the Matrix and its Diagonalization

Ł. Herok[1], R. Szczęśniak[1,2], A.P. Durajski[2*]

[1]*Institute of Physics, Jan Długosz University*
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland

[2]*Institute of Physics, Częstochowa University of Technology*
Al. Armii Krajowej 19, 42-200 Częstochowa, Poland
[*]*E-mail: adurajski@wip.pcz.pl*

Received: 28 April 2015; revised: 06 September 2015; accepted: 07 September 2015; published online: 06 November 2015

Abstract: In order to perform larger scale physics research in the area of superconductivity, we have developed an application that can transform the Hubbard Hamiltonian into a matrix and diagonalize it to find the selected model's energy spectrum. For that purpose we have used the Python language and its wide ecosystem. This paper proves that selected tools are capable of creating scientific applications in a general sense. After a short introduction into the physics problem and the designed algorithm we will present the computer science problems and their solutions in creating usual scientific programs, in particular: performance and parallelization issues, storage of input data and the results, bottlenecks detections, as well as optimization and testing. The most interesting examples of the developing cycle will be described to give a prepared solution for implementing the other scientific software.

Key words: Python, scipy, hamiltonian matrix, sparse matrix

I. INTRODUCTION

Nowadays many problems from physics and the other natural sciences can be investigated with numerical computations. Computer science serves as a field for theoretical experiments, so it is crucial for a scientist to have appropriate tools at the ready. This paper presents the survey of Python language libraries and its packages in creating scientific software with all its issues. The main goal was to prepare a program that could handle transformation of the Hubbard Hamiltonian for asked number of sites into Hamiltonian matrix. The next step was the matrix diagonalization. When implementing this solution we had to face the usual problems that have to be solved in most scientific computation programs, especially in physics and chemistry, including:

- parallelization and concurrent access to data,
- handling large amount of data in the memory,
- storing results,
- code and algorithms optimization,
- creating code prepared for further developing and changing.

The scientists' programming environment, beside a powerful language, should have a wide scientific ecosystem with rich mathematical and engineering libraries. The researcher can focus on solving a given issue and not be distracted and slowed down by coding well-known algorithms. The Python language meets this requirement perfectly, with a vast number of packages for numeric computation, mathematics and statistic libraries even dedicated for physics society. Python is a general-purpose, dynamic programming language. Interpreted languages are usually considered to be slower in computation time than their compiled equivalents. It is not always be true and it depends on the case [1]. However, they are really faster in program development, which means that we get the working code, the required tool, faster. Dynamic languages can be successfully used for serious work in large-scale applications without suffering a performance penalty or significantly increasing software complexity [2]. They are also more convenient in daily work and allow one to introduce ad-hoc changes faster, which could be a great benefit during experiments. The article is organized as follows. Section II describes a physical problem to be solved, presents foundations of the program architecture and gives insight into specific algorithms. Finally, it reports statistical data associated with the chosen way of solving the issue. Section III discusses several important Python environment techniques that were used while creating the tool. It shows the encountered problems and the way they were handled giving the hints for other scientist interested in developing their own tools in Python.

II. HAMILTONIAN TRANSFORMATION INTO MATRIX REPRESENTATION

Transformation of the Hubbard Hamiltonian into a matrix for more than three sites is tedious work that should be done numerically. The best approach would be to process all equations in a symbolic way to achieve a result that can serve as an input for further calculations, for example substitute different values for parameters and do matrix diagonalization. This approach generates a huge amount of data in the form of equations that are elements of large sparse matrices. That reveals the problem of handling this data in a memory and its later, persistent storage for the next stages. The problem of transformation could be split into smaller pieces so it is natural that it should be implemented in a multiprocessing way.

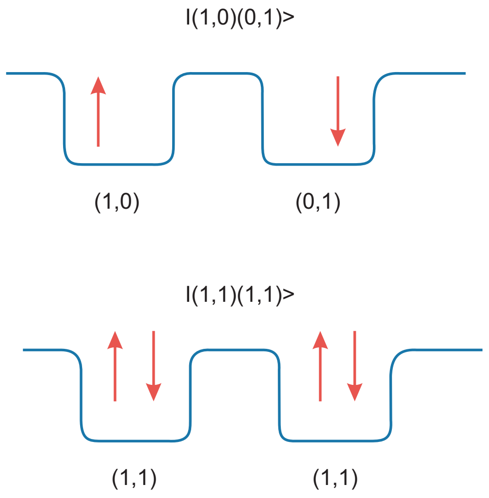


Fig. 1. Example states of two site model

II. 1. Hubbard Hamiltonian

A Hubbard Hamiltonian that models an electron system has the following form [3]:

$$H = t\sum_{\sigma,i,j}^{N} c^{\dagger}_{i\sigma}c_{j\sigma} + \epsilon_0 \sum_{\sigma,i}^{N} c^{\dagger}_{i\sigma}c_{i\sigma} + U\sum_{i}^{N} c^{\dagger}_{i\uparrow}c_{i\uparrow}c^{\dagger}_{i\downarrow}c_{i\downarrow}, \quad (1)$$

where N is a number of sites, t is a hoping integral, ϵ_0 describes a reference energy, U represents on-site Coulomb interaction, $c^{\dagger}$ and c stand for a fermion creation and annihilation operator, respectively. Indexes i and j describe the sites while σ is spin of an electron. To find the spectrum energy of the modelled system the Hamiltonian needs to be transformed into the matrix form, and after substitution of the hamiltonian parameters, according to a chosen model, it should be diagonalized.

II. 2. Algorithm idea

According to the Equation 1, the hamiltonian for a two-site model (N=2) takes the following form:

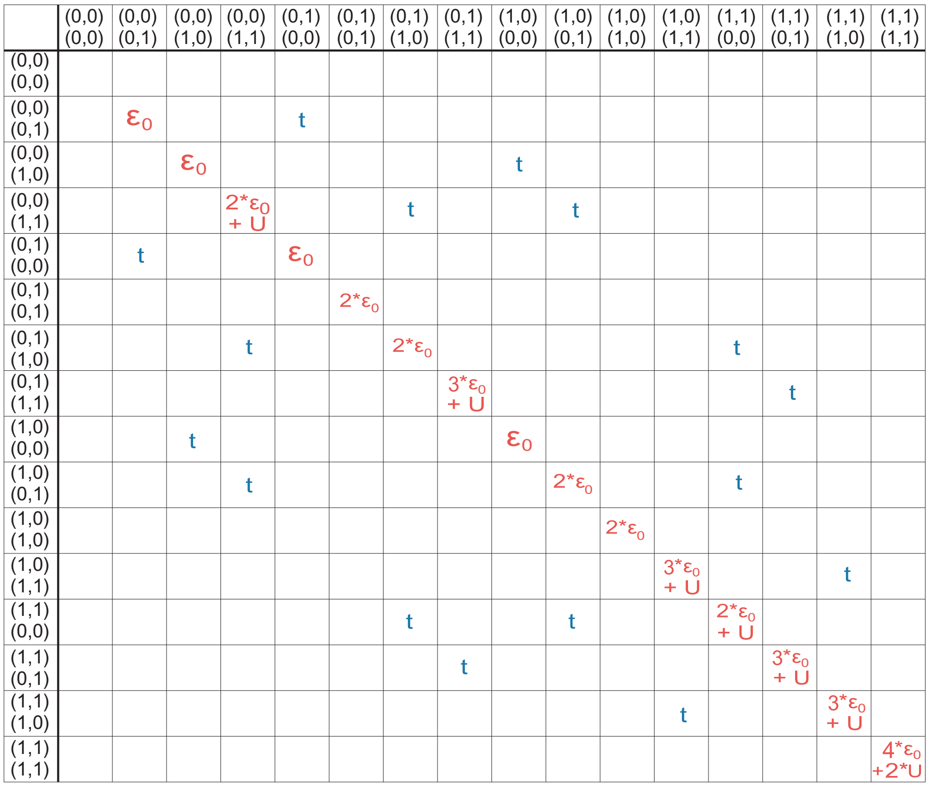
$$\begin{aligned} H = {} & t\sum_{\sigma}(c^{\dagger}_{1\sigma}c_{2\sigma} + c^{\dagger}_{2\sigma}c_{1\sigma}) \\ & + \epsilon_0\sum_{\sigma}(c^{\dagger}_{1\sigma}c_{1\sigma} + c^{\dagger}_{2\sigma}c_{2\sigma}) \\ & + U(c^{\dagger}_{1\uparrow}c_{1\uparrow}c^{\dagger}_{1\downarrow}c_{1\downarrow} + c^{\dagger}_{2\uparrow}c_{2\uparrow}c^{\dagger}_{2\downarrow}c_{2\downarrow}) \end{aligned} \quad (2)$$

To transform the Hubbard Hamiltonian into matrix representation it needs to find its influence on every state that the system can take. In the discussed situation there is 4^2 number of states. They can be visualised using the diagrams presented in Figure 1.

Due to the Pauli exclusion principle on every site there can exist at most two electrons with opposite spins. If an electron with the up spin exists, then it is noted by 1 on the first position in the site bracket if it does not exist, then it is marked by 0. Based on the calculated equation according to the pattern $< final\_state|H|initial\_state >$ the Hamiltonian matrix can be built. The result is presented in Figure 2. We can observe that the reference energy electron system and the on-site Coulomb potential U occupy the diagonal elements. On the other hand, the hopping integral t defines off-diagonal cells. The matrix prepared in such a form, with parameters in the cells, is a good entry point for further research in order to find the energy spectrum of the examined model. For that purpose it would be necessary to compute the Hamiltonian parameters for the selected model, and substitute those values in the matrix to perform diagonalization.

II. 3. Program's architecture

General actions that need to be taken to do the transformation, discussed in the previous sections, are presented in Figure 3. The program computation flow consists of three main phases:



	(0,0) (0,0)	(0,0) (0,1)	(0,0) (1,0)	(0,0) (1,1)	(0,1) (0,0)	(0,1) (0,1)	(0,1) (1,0)	(0,1) (1,1)	(1,0) (0,0)	(1,0) (0,1)	(1,0) (1,0)	(1,0) (1,1)	(1,1) (0,0)	(1,1) (0,1)	(1,1) (1,0)	(1,1) (1,1)
(0,0) (0,0)																
(0,0) (0,1)		ε_0			t											
(0,0) (1,0)			ε_0						t							
(0,0) (1,1)				$2*\varepsilon_0 + U$			t			t						
(0,1) (0,0)		t			ε_0											
(0,1) (0,1)						$2*\varepsilon_0$										
(0,1) (1,0)				t			$2*\varepsilon_0$						t			
(0,1) (1,1)								$3*\varepsilon_0 + U$						t		
(1,0) (0,0)			t						ε_0							
(1,0) (0,1)				t						$2*\varepsilon_0$			t			
(1,0) (1,0)											$2*\varepsilon_0$					
(1,0) (1,1)												$3*\varepsilon_0 + U$			t	
(1,1) (0,0)							t			t			$2*\varepsilon_0 + U$			
(1,1) (0,1)								t						$3*\varepsilon_0 + U$		
(1,1) (1,0)												t			$3*\varepsilon_0 + U$	
(1,1) (1,1)																$4*\varepsilon_0 + 2*U$

Fig. 2. Hamiltonian matrix for $N = 2$

1. Determination influence of the Hamiltonian on every state
2. Preparation the Hamiltonian matrix in a symbolic form
3. Matrix diagonalization with substituted values for the hamiltonian parameters

The results of each phase are stored in the database, and they work as an input data for every next phase. Thanks to that the program can be started in any phase. The only requirements are: availability of data in the database from previous phases, and user-provided parameters for computation. Every phase is implemented in a parallelized way to maximally utilize available computation resources and deliver results as fast as possible. It is marked in Figure 3 by shadowed blocks. Processing in the first and second phases should be realized on structures that can represent and process equations in a symbolic form. Symbolic calculation is very slow compared to floating point operations [4]. However, in our case it is crucial to achieve results in a symbolic form to speed up the next stages of research. They will be repeated plenty of times starting from our preprocessed data. The third phase, matrix diagonalization, takes as an input matrix with symbolic equations that could be used for many different Hamiltonian parameters values, depending on the tested model. Diagonalization is made using the Lanczos method that returns eigenvalues and corresponding eigenvectors.

II. 4. Results

The program was built in the Python language according to the discussed algorithm. Below we present some statistical information about its performance. The following results were obtained on the machine with configuration: 12 CPU cores Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz and 64 GB RAM running under: Linux Fedora 19, Python 3.3.2, numpy 1.9.1, scipy 0.14.0, sympy 0.7.4.1 and PostgreSql 9.2.8. Tables 1 and 2 present processing time and size of output data in the database, for example the number of sites. $H|ket>$ presents statistics for computing hamiltonian influence on a state, and $< bra|H|ket >$ for building a hamiltonian matrix. Final program was equipped with a dedicated reporting tool. Results for a particular number of sites can be viewed as an HTML document in a matrix form as shown in Figure 2. The next section presents technical solutions for mentioned issues in the Python language.

III. IMPLEMENTATION IN PYTHON ENVIRONMENT

Python is a dynamic programming language that allows fast development of new programs. It is a high-level language that allows one to focus on solving the problem rather

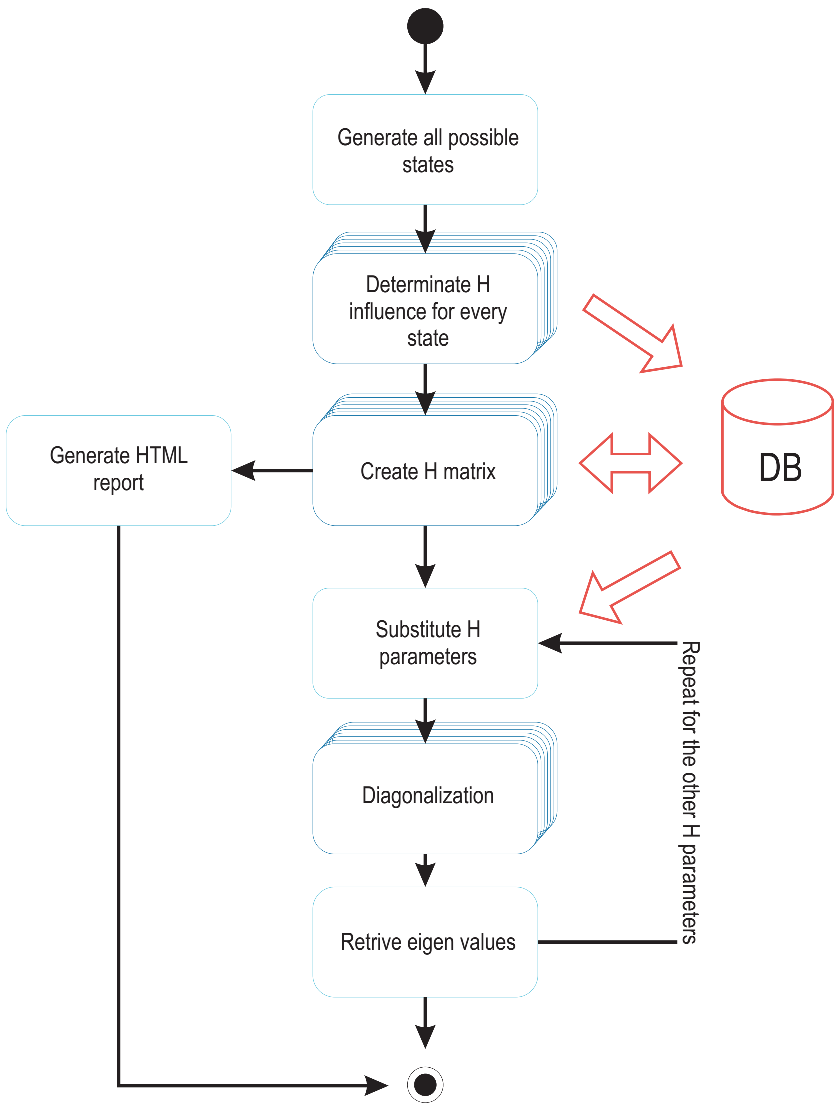

Generate all possible states
Determinate H influence for every state
Create H matrix
DB
Generate HTML report
Substitute H parameters
Repeat for the other H parameters
Diagonalization
Retrive eigen values

Fig. 3. The program algorithm

than fighting with low level programming issues and in the end greatly enhance productivity. Python has a "batteries included" philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages [5]. This philosophy connected with rich scientific packages makes it a great choice for developing scientific software. Python integrates well with other programming languages. For that reason it can serve as a great control language and glue code from different programming languages combining it all together in one program that works [6]. This can really speed up the development process thanks to using previously created routines. An important factor for scientists and engineers is Python's clarity of syntax. It makes the program code easy to read, understand and maintain. It is enforced by proper indentation, namespaces, well-designed built-in instructions and documentation strings. The clarity of the code, easy usage and availability of a rich base of packages, makes it a significant competitor in comparison with Matlab, Fortran, and C/C++ [7]. All listed aspects allow one to focus on the problem and treat the programming language as a tool which supports creating a clear, readable code that yields the idea behind it and documents the experiment [2].

The Python language has two core versions 2 and 3, which differ in syntax and are incompatible with each other. Nowadays Python 2 is still more often used, but it is only a matter of time when its successor Python 3 becomes a daily standard. The program in this work was fully implemented in Python 3.

III. 1. Performance consideration

Scientists are usually trained to work with the compiled languages Fortran or C/C++. These languages offer performance needed to solve large numerical problems. The problem is that these languages are not easy to use, the implementation is hard to change and maintain while the final

Tab. 1. Computation time

	Sites		
	6	7	8
$H\|ket>$	00h 00m 20s	00h 01m 56s	00h 11m 22s
$< bra\|H\|ket >$	00h 03m 25s	00h 38m 18s	09h 37m 37s

Tab. 2. Size of output data in the database tables

	Sites					
	6		7		8	
	Size [MB]	Rows	Size [MB]	Rows	Size [MB]	Rows
$H\|ket>$	0.7	4096	35	16384	179	65536
$< bra\|H\|bra >$	35	65535	207	360447	1031	1900543

solution is delivered very slow. The final program code hides it's idea behind the language syntax, data structures and instructions. Python is a high-level interpreted language. For many cases it will be slower in execution time than its C/C++ equivalents [8]. As mentioned before, the Python language is flexible in the manner of glueing with other languages. So even if a bottleneck is detected, the part of the program could be rewritten in Fortran or C/C++ code. In our case the Lanczos algorithm provided by the scipy package is implemented in the Fortran language and distributed in the Fortran77 ARPACK package [9]. It contains subroutines designed to solve large scale eigenvalue problems. Usage of this package in the Python code is straightforward as it was shown in Listing 1.

Listing 1. Usage of the Fortran77 package with the Lanczos algorithm

```
from scipy.sparse.linalg import eigsh
vals, vecs = eigsh(matrix, eigs_number,
    return_eigenvectors=True,
    which='LM')
```

Another approach widely used to increase performance of the Python program is to use Cython [10], an optimising static compiler for the Python programming language. It makes writing C extensions for Python as easy as Python itself. In the end we get a readable high-level language code that is compiled to a very efficient C code. In that case we obviously lose dynamic language interactivity.

III. 2. Parallelization

The Python language is equipped with two solutions for concurrent code execution contained in separated packages multithreading and multiprocessing. The appropriate choice of a tool depends on the task to be executed, if it is CPU bound or IO bound. The multithreading module suffers because of the Global Interpreter Lock [5] so despite easy use it cannot utilize multi CPU cores at the same time. For that reason the main usage of this module is reduced to introduce parallelism of execution during the IO operations. Scientific numeric computations are the class of problems that are CPU-bound. The multiprocessing module allows the programmer to fully leverage multiple processors on a given machine so it is a good choice for carrying on simultaneous numerical computations. In our case concurrent execution appears three times: applying hamiltonian for all states, creating hamiltonian matrix, diagonalization. The first two cases are implemented on our own, the third one is handled by an imported scipy package, and ARPAC package underneath. The framework for the first and second case is the same. We start one process manager that runs a pool of workers with a number equal to the number of available CPU cores. The difference is in providing data to process and collecting results. For the first phase, applying hamiltonian for all states, all possible states are generated in the main process, and each worker is ordered with one state and should return the result to the parent process which is responsible for collecting results and storing it in the database. In the second case, the matrix creation phase, the worker process gets information about the range of data to process. It reads a portion of input data from the database and stores back the results directly in the database. Example implementation for the described phases is illustrated by code in Listings 2 and 3.

Listing 2. Parallelization of applying hamiltonian for all states

```
state_cnt = 4 ** sites
pool = Pool(maxtasksperchild=1000)
res = [pool.apply_async(apply_hket,
    (sites, state))
    for state in range(state_cnt)]
pool.close()
pool.join()
```

Listing 3. Parallelization of matrix creation

```
pool = Pool(maxtasksperchild=1)
[pool.apply_async(compute_slice,
    (sites, slice_no, size, quantity))
    for slice_no in range(quantity)]
pool.close()
pool.join()
```

The idea of parallelization in the matrix creation phase is to slash the list of previously computed data in slices. Each worker is ordered the slice number to compute. The worker process is responsible for loading the indicated slice of data from the database into the memory. Then it undertakes computation against its slice and all the other slices. At the same time every worker can keep in the memory a maximum of two slices that it operates on. Taking into account this constraint the number of slices and slice size depend on the available machine's RAM memory and number of CPUs. It is demanded that the slice size be as large as possible to minimize communication with database, due to the IO bound.

Before coming to this solution, of distributing data and collecting results from workers, attempts were made to pass to each worker process the complete list of data. This quickly ended in out of memory problem and large delays in transferring the data to subprocesses. The second attempt was to use the operating system mechanism Copy On Write. The operating system copies data that is held in global variables to subprocess only when the subprocess or the main process is trying to change it. At first sight it promised high efficiency and memory saving. Unfortunately, we ended up with uncontrolled coping data, on accessing list elements, causing out of memory problems.

III. 3. Data types and structures

The real challenge was to handle data in symbolic representation. The sypmpy package with its Expr class for algebraic expressions appeared to be helpful. In connection with its subpackage physics.quantum it allowed us to hold all required expressions in a symbolic form and carry all mathematical operations.

Listing 4. Handling symbolic expressions with the sympy package.

```
U = symbols('U')
k = Ket((0,1),(0,0))
expr = - U * k
print(expr)
>>> -my*|(0, 1)(0, 0)>
```

In the first attempt the system state was represented by a generic Python data type: tuple [5]. Its presentation reflects the adopted physics state presentation. The tuple was enveloped by the Ket object in discussed symbolic expressions structures. All possible states form series of succeeding binary numbers from 0 to 2N, left padded with zeros to 2N places. Discovering this allowed us to create a lightweight algorithm, where states were generated using built-in Python functions for handling binary numbers.

Tab. 3. The results of the state generator for two-site model

Decimal	Binary	Ket
0	0000	$\|((0,0),(0,0))>$
1	0001	$\|((0,0),(0,1))>$
2	0010	$\|((0,0),(1,0))>$
...	...	...
7	1111	$\|((1,1),(1,1))>$

Another interesting problem was to find an appropriate data structure for sparse matrices. The Hamiltonian matrix is a sparse matrix, which means that most of its cells hold 0 value. It would be a waste of computer resources, and even impossible to create an array of matrix size in the memory. This problem was handled differently in the phase of matrix creation and diagonalization. During matrix creation the results are kept in a list [5] a built-in Python collection. Its elements are represented by named tuples as it was shown in Listing 5. For the diagonalization procedure we need to whole matrix into memory. Python's package scipy.sparse provides several implementations of data structures that are capable of handling a sparse matrix. In our case we have chosen to use the dok_matrix, because this structure was most convenient for loading it from the database tables.

Listing 5. Named tuple

```
element = namedtuple('MatrixTuple',
    ['bra', 'ket', 'result'])
```

Listing 6. Sparse matrix

```
from scipy.sparse import dok_matrix
dm = dok_matrix(4**8, 4**8))
dm[bra, ket] = result
```

Pointed data structures allowed us to hold a large, sparse matrix in a memory and access its elements much faster.

III. 4. Profiling

The working program was analyzed for the performance bottlenecks. For that purpose we have used cProfile [5] tool. cProfile provide deterministic profiling of Python programs. It produces a set of statistics that describe how often and for how long various parts of the program were executed. According to the analysis of the most time-consuming operation it was found that the slowest instruction was comparing equality of Bra and Ket states during matrix generation.

Listing 7. Profiling Python code

```
import cProfile
cProfile.runctx('compute_slice(sites, 0,
    size, quantity)', globals(), locals
    (), sort=2)
```

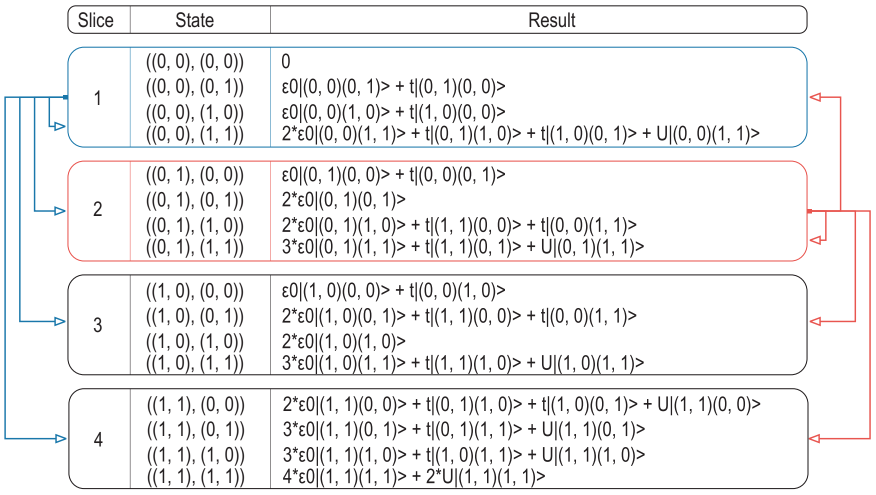

Slice
State
Result
1
((0, 0), (0, 0))
((0, 0), (0, 1))
((0, 0), (1, 0))
((0, 0), (1, 1))
0
ε0|(0, 0)(0, 1)> + t|(0, 1)(0, 0)>
ε0|(0, 0)(1, 0)> + t|(1, 0)(0, 0)>
2*ε0|(0, 0)(1, 1)> + t|(0, 1)(1, 0)> + t|(1, 0)(0, 1)> + U|(0, 0)(1, 1)>
2
((0, 1), (0, 0))
((0, 1), (0, 1))
((0, 1), (1, 0))
((0, 1), (1, 1))
ε0|(0, 1)(0, 0)> + t|(0, 0)(0, 1)>
2*ε0|(0, 1)(0, 1)>
2*ε0|(0, 1)(1, 0)> + t|(1, 1)(0, 0)> + t|(0, 0)(1, 1)>
3*ε0|(0, 1)(1, 1)> + t|(1, 1)(0, 1)> + U|(0, 1)(1, 1)>
3
((1, 0), (0, 0))
((1, 0), (0, 1))
((1, 0), (1, 0))
((1, 0), (1, 1))
ε0|(1, 0)(0, 0)> + t|(0, 0)(1, 0)>
2*ε0|(1, 0)(0, 1)> + t|(1, 1)(0, 0)> + t|(0, 0)(1, 1)>
2*ε0|(1, 0)(1, 0)>
3*ε0|(1, 0)(1, 1)> + t|(1, 1)(1, 0)> + U|(1, 0)(1, 1)>
4
((1, 1), (0, 0))
((1, 1), (0, 1))
((1, 1), (1, 0))
((1, 1), (1, 1))
2*ε0|(1, 1)(0, 0)> + t|(0, 1)(1, 0)> + t|(1, 0)(0, 1)> + U|(1, 1)(0, 0)>
3*ε0|(1, 1)(0, 1)> + t|(0, 1)(1, 1)> + U|(1, 1)(0, 1)>
3*ε0|(1, 1)(1, 0)> + t|(1, 0)(1, 1)> + U|(1, 1)(1, 0)>
4*ε0|(1, 1)(1, 1)> + 2*U|(1, 1)(1, 1)>

Fig. 4. The idea of parallelization creation of matrix

As it was mentioned in the very beginning, a state was represented by a tuple, formed as binary digits grouped by parentheses, enveloped by a Bra or Ket object. For models consisting of more than five sites, a binary state represenation started to be quite a complicated object to compare. After detecting a problem with performance, representation was switched to decimal numbers. Previous binary notation, eg. $|((0,1),(0,1),(1,1)) >$, was replaced by decimal notation, eg. $|23 >$. That change had also a positive influence on reducing usage of the memory and storage space. There was no need to change the instructions of states comparison. Added were only functions for translating a decimal number to formal state representation for presentation purposes. Performance growth after optimization is presented in Table 4.

Tab. 4. Performance growth after optimization

	Sites		
	6	7	8
Before	00h 04m 23s	01h 20m 06s	25h 17m 19s
After	00h 03m 25s	00h 38m 18s	09h 37m 37s
Performance growth [%]	22	52	62

Another problem that occurred during developing the transformation tool was lack of memory. The problem appeared in parallel implementation of matrix creation. With the help of the resource package and functions available in the sys module we discovered that there was a problem with freeing memory for allocated variables in the sympy package. The resource package provided basic mechanisms for measuring and controlling system resources utilized by a program, when functions in the sys module were used for investigating object sizes. The problem was solved by forcing clearing the sympy cache and restarting the worker process after every job.

Listing 8. Solving out of memory problem in the sympy package

```
sympy.clear_cache()
pool = Pool(maxtasksperchild=1)
```

III. 5. Persistent storage

The program generates a large amount of data that needs to be persisted for particular phases. The simplest way of storing data are flat files. According to Grays Law's [11], as a storage engine for our application, the PostgreSql relational database was chosen. Postgresql is the most advanced open source database which can handle multiple client access at the same time. Using a database instead of file storage solves a couple of problems, including concurrent access to data, concurrent writings, convenient and fast retrieving data, and scalability [12]. The data generated by the first and second phases are stored in separate tables with a structure presented in Figure 5. The obj column holds serialized objects: number of sites, state and its resulting equation. The data in other columns are added demonstratively in order to facilitate direct viewing of results.

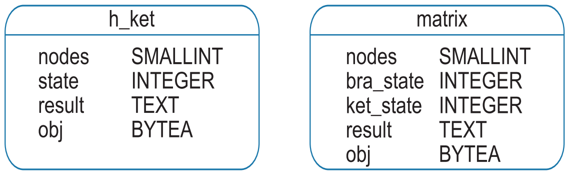


Fig. 5. Database structure

For our application the relational database has been chosen, but looking at the structure of the data and usage scenarios probably a document database could be a better choice.

III. 6. Testing

Tests are a kind of insurance that allows future program modification, optimization and development. It serves also as a project documentation and usage example. The whole development process was carried out in the Test Driven Development technique. It resulted in the high quality code that was checked to work as it was expected. The Python environment provides several testing frameworks: unittest, nose, pytest, and doctest. In our development cycle we stayed with the unittest framework that supports test automation, aggregation of tests into test cases, and helpers for easy test scenario creation.

Listing 9. Test case for applying hamiltonian on a state

```
class TestH(unittest.TestCase):
  def test_apply(self):
    u, e, t = symbols('u e t')
    expected = t*KetDec(59) - t*KetDec
        (62) + 2*u*KetDec(47) - 5*e*
        KetDec(47)
    self.assertEqual(apply(n=3, state
        =47), expected)
```

Using the unit test framework we have also implemented an integration test for diagonalization procedure to make sure it returns solutions in accordance with mathematical equation $Av = \lambda v$, where λ is an eigenvalue of matix A, and v is its corresponding eigenvector. A complete test suit allowed us for safe program refactoring, optimization and modifications that appeared many times in the development cycle.

III. 7. Science packages

As it was said in the very beginning, Python has a strong scientific ecosystem that is centralized around SciPy stack. It gathers a variety of Python open-source extensions for mathematics, engineering, physics and other disciplines. It enriches the scientist's kit with tools for:

- fundamental library for scientific computing, statistical functions and numerical methods (SciPy library),
- n-dimensional homogenous array package allowing fast array manipulation (NumPy),
- symbolic mathematics and physics (Sympy),
- result visualization with comprehensive Plotting (Matplotlib),
- interactive console with Mathematica like notebooks (iPython),
- and many others.

In our case we have used Scipy packages for matrix diagonalization. The Sympy module allowed us to write down and carry on hamiltonian transformation in a symbolic way and in the final step substitute real values for symbols. iPython deserves special attention. It makes the most of Python interactivity. It provides access to the programing language and all its packages through well-known Mathematica or Matlab like notebooks [5]. It allows live programing for testing ideas and at the same time provides a visual running environment for your programs. Ipython exhibits the scientific nature of the Python language.

IV. CONCLUSIONS

In this work we have shown the process of building the scientific application in the Python language and its environment. We have designed and created the program for hamiltonian to matrix transformation that will be used in our further research. In this article we have shared our experiences in building specialized physics application in the Python language. The outlined approach is so general that it is applicable to many different scientific areas and common classes of problems. It was proven that Python can successfully replace Matlab, C or Fortran environments allowing for faster and more convenient software development whilst preserving performance in the final product. This general purpose programming language with a vast number of ready to use packages, often on a liberal open source license, is a great choice for scientists and engineers [13]. A powerful interactive interpreter reveals its scientific nature, it is ready for live experimentations and fast idea validations.

In the future work we are planning to extend the presented software to implement functionality of transformation into a matrix for any hamiltonian provided in the second quantization notation. We will also work on increasing performance to prepare software for analyzing larger models: over ten-sites. It will be done by implementing most time-consuming operations in Cython and finding places where the algorithm could be simplified by finding symmetric problems. We will also consider another approach to decrease computation time, scattering calculations on multiple client machines by implementing the MPI – Message Passing Interface [14, 15].

References

[1] T.E. Oliphant, *Python for Scientific Computing, Computing in Science & Engineering* **9**, 10-20 (2007).

[2] D.M. Beazley, P.S. Lomdahl, *Extensible Message Passing Application Development and Debugging with Python*, Proceedings 11th International Parallel Processing Symposium, 650-655 (1997).

[3] J. Spałek, *Wstęp do fizyki materii skondensowanej*, Wydawnictwo Naukowe PWN, Warszawa 2015.

[4] H. Fangohr, *Python for Computational Science and Engineering*, Faculty of Engineering and the Environment University of Southampton, Southampton 2014.

[5] Python Software Foundation, Python Documentation, https://docs.python.org, 2015.

[6] D.M. Beazley, *Scientific Computing with Python*, Astronomical Data Analysis Software and Systems IX ASP Conference Series **216**, San Francisco 2000.

[7] D.M. Beazley, P.S. Lomdahl, *Building Flexible Large-Scale Scientific Computing Applications with Scripting Languages*, 8^{th} SIAM Conference on Parallel Processing for Scientific Computing, Minnesota 1997.

[8] X. Cai, H. P. Langtangen, H. Moe, *On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations*, Scientific Programming **13**, 31-56 (2005).

[9] R.B. Lehoucq, D.C. Sorensen, C.Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, University of Leeds, 1997.

[10] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, K. Smith, Cython: *The Best of Both Worlds*, Computing in Science and Engineering **13**, 31-39 (2011).

[11] A. S. Szalay, J. A. Blakeley, *Gray's laws: database-centric computing in science*, [in:] T. Hey (ed.), S. Tansley (ed.), K. Tolle (ed.), *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Redmond, p. 5-11, 2009.

[12] P. Buneman, *Why Don't Scientists Use Databases?*, Microsoft PowerPoint Presentation to National e-Science Centre, (2002).

[13] A. Noreen, K. Olaussen, *A Python Class for Higher-Dimensional Schroedinger Equations*, arXiv:1503.04607 [physics.comp-ph], (2015).

[14] M.R B. Kristensen, B. Vinter, *Numerical Python for scalable architectures*, Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, New York, art. 15 2010.

[15] J.K. Nilsen, X. Cai, B. Høyland, H. P. Langtangen, *Simplifying the parallelization of scientific codes by a function-centric approach in Python*, Computational Science & Discovery **3**, (2010).

Łukasz Herok is a PhD candidate at the Jan Dlugosz University in Częstochowa, Faculty of Theoretical Physics. He received a master degree in Computer Science at Silesian University of Technology. His research focus on generalization Eliashberg functions for more precise superconductivity exploration. His long-term interest is developing software for quantum computers. He is also a professional software architect and business analyst who designs and implements dedicated software solutions for manufacturing companies.

Radosław Szczęśniak is a Professor at the Częstochowa University of Technology and the Jan Dlugosz University in Częstochowa (Institute of Physics). His subject of interest is theoretical physics, particularly the theory of the superconducting state and the transport phenomena.

Artur Durajski is an Assistant Professor at the Institute of Physics, Częstochowa University of Technology and a vice-president of Częstochowa Department of Polish Physical Society. In 2011, he received his MSc Eng. degree in Physics at the Częstochowa University of Technology and in 2014 he received his PhD degree (cum maxima laude) in Physics at the University of Zielona Góra. His research interests concern high-pressure thermodynamic properties of phonon-mediated superconductors, the strongly correlated systems and modified electron-phonon pairing mechanism applied to cuprates.