# FASTER KENZO COMPUTATIONS VIA SAGEMATH, AND VICE VERSA

### JOSE DIVASÓN, MIGUEL MARCO BUZUNÁRIZ, AND ANA ROMERO

ABSTRACT. This work presents some improvements on the efficiency of both Kenzo and SageMath, by means of parallelization techniques and an existing interface that connects both systems.

#### INTRODUCTION

Kenzo [2] is a computer algebra system devoted to algebraic topology which, in particular, implements several algorithms to compute homology groups of infinite structures using the method of effective homology [4]. In addition, it also permits to compute algorithmically homotopy groups combining the Whitehead tower method [5] and the effective homology technique. As far as we know, Kenzo is the only program able to carry out this kind of computations over infinite structures, which makes it a powerful software. In order to spread and ease the use of Kenzo, we developed an interface and an optional package of Kenzo within Sagemath [1]. That work permitted using Kenzo and some of its external packages without any Common Lisp knowledge and enhanced the SageMath system with new capabilities in algebraic topology (dealing in particular with simplicial objects of infinite nature).

In this work, we improve the efficiency of our Kenzo–SageMath interface by combining the power of both computer algebra systems, considering in particular the computation of homology groups. On the one hand, we use parallel computations in SageMath (using multiprocessing in Python) to accelerate Kenzo computations. On the other hand, Kenzo can improve SageMath capabilities beyond topological computations. One such example is the Smith normal form of an integer matrix, for which Kenzo provides a faster implementation than Pari (which is what SageMath uses).

# 1. Computation of homology groups in Kenzo

The computation of homology groups in Kenzo is done by means of the effective homology method [4]. When an object (for instance, a simplicial set) X is built in Kenzo, a particular case of homology equivalence  $C_*(X) \iff E_*$  is automatically constructed, where  $C_*(X)$ is the chain complex associated with X and  $E_*$  is a chain complex of finite type (called effective) such that its homology groups are isomorphic to those of  $C_*$ ,  $H_*(C) \cong H_*(E)$ . Since  $E_*$  is finitely generated in each degree, the homology groups of  $E_*$  can be determined

The first and third author are supported by grant PID2020-116641GB-I00 funded by MCIN/ AEI/ 10.13039/501100011033. The second author has been partially supported by PID2020-114750GB-C31 and E22\_20R: Álgebra y Geometría.

The talk at the meeting EACA 2022 has been given by the third author.

by means of elementary operations on matrices. In this way, the homology groups of the object X, which can be of infinite nature, can also be determined (thanks to the isomorphism  $H_*(X) \cong H_*(C_*(X)) \cong H_*(E)$ ). Given a chain complex  $C_*$ , its homology groups  $H_n(C_*)$  are defined as  $H_n(C_*) = \operatorname{Ker} d_n / \operatorname{Im} d_{n+1}$ , where  $d_*$  denotes the differential map of the complex. These groups are determined in Kenzo by means of the following algorithm.

| <b>THEOLUGINE IN TOTAL STORES OF COMPANY OF COMPANY</b> | Algorithm | 1: | Homology | groups | of a | chain | complex. |
|---|-----------|----|----------|--------|------|-------|----------|
|---|-----------|----|----------|--------|------|-------|----------|

**Input:** A chain complex  $C_*$  with effective homology and an integer n. **Output:** The homology group  $H_n(C_*)$ .

- 1 Consider the effective chain complex associated with  $C_*$ , denoted  $E_*$ .
- **2** Construct the differential matrix of  $E_*$  of degree *n*, denoted  $D_n$ .
- **3** Construct the differential matrix of  $E_*$  of degree n + 1, denoted  $D_{n+1}$ .
- 4 Compute the kernel of  $D_n$ , denoted  $K_n$ , by diagonalization techniques [3].
- **5** Return the quotient of  $K_n$  by  $D_{n+1}$ , by using again diagonalization techniques.

Using profiling, we detected that 99% of the required time was devoted to instructions in lines 2 and 3 of Algorithm 1, that is, determining the differential matrices of the effective chain complex. This is due to the fact that these matrices are built by determining the image of the differential map of each generator of the chain complex  $E_*$  on the required degrees, and the differential morphisms of the effective chain complex  $E_*$  are constructed by means of complicated maps describing the effective homology of the object [4].

# 2. Improving Kenzo and SageMath

The existing interface between SageMath and Kenzo [1] connects both programs via the ECL library (a library interface to Embeddable Common Lisp), which is itself loaded as a C-library. This permits the interface to be very fast, being the efficiency between native Kenzo and Kenzo loaded in Sagemath rather similar. The idea of this work is to exploit both systems to improve the efficiency of computations. All the code is publicly available at https://github.com/jodivaso/EACA22.

2.1. Improving Kenzo via SageMath. We have applied parallelization techniques to Kenzo thanks to the SageMath interface. That is, we use existing multiprocessing Python libraries to run parallel computations in Kenzo. Unlike other programming languages, Python multithreading (via threading and asyncio Python packages) does not allow parallelization, but only concurrency. The reason is the existence of a Global Interpreter Lock (GIL), whose purpose is to allow only one thread to hold the control of the Python interpreter. Thus, the standard way to run in parallel a task in Python is by means of independent subprocesses, without sharing memory among them. However, programmers need to somehow share objects between subprocesses (for example, to share the input matrix with the subprocess that will compute something of it). This is solved thanks to serialization (or *pickle* in Python jargon): the process whereby a Python object is converted into bytes. Then, if the main process needs to send two different matrices to two subprocesses (like if we parallelize lines 2 and 3 in Algorithm 1), then two serializations are done (one for each matrix). The matrices are then unpickled in the subprocesses, computation is done in each subprocess, results are also serialized and finally unpickled in the main process.

The Kenzo interface defines the class KenzoObject, which is simply a wrapper in Sage around a Kenzo object, i.e., it only contains an EclObject. However, pickling over an EclObject is not supported, since ECL does not natively support serialization of its objects. To overcome this limitation, we have added now an attribute named command, in which we store how the object has been built. This variable stores the commands to reconstruct the Kenzo object by means of Python code (which internally will run Lisp code thanks to the interface). Then, KenzoObject has two attributes now: the EclObject already built and the command to build it. We do this for each object that can be constructed in Kenzo via the interface (Cartesian product, wedge, loop space, join, spheres, ...).

To serialize a KenzoObject (and also any of its inherited classes, like KenzoChainComplex) the trick is to define our own method to serialize (pickle), which will only serialize the command, but not the EclObject. We also defined the deserializing method (unpickle), where the command is executed to reconstruct the EclObject. Therefore, we can input and output KenzoObjects between processes. We parallelize Algorithm 1 in two ways:

- (1) Computing  $D_n$  and  $D_{n+1}$  in two different processes (parallelism by matrices).
- (2) Computing separately the columns of  $D_n$  and  $D_{n+1}$  in *m* processes, and then reconstructing  $D_n$  and  $D_{n+1}$  (parallelism by columns).

The former is done by means of multiprocessing.pool. Only Python code is necessary to get parallelism. The latter requires to define new Lisp code that, given  $E_*$ , a degree kand two indexes i and j, computes the columns from i to j of  $D_k$ . Additionally, we also have an optional parameter to select the number of cores to use (to maximize the performance, by default is set to the available number of logical cores). Finally, we reconstruct each one of the matrices from their columns and continue with the process.

Our benchmarks show that execution times improve noticeably, although it depends on the space. For instance, to compute the homology in dimension 13 of the cartesian product of the Eilenberg–MacLane spaces  $K(\mathbb{Z},3)$  and  $K(\mathbb{Z}/5\mathbb{Z},7)$  requires 6627.8s without doing any parallelism, 6457.28s using the parallelism by matrices and 1734.5s using parallelism by columns. Thus, the computing time is reduced by around 75% in such an example.



The figures presented above show the execution times (in a logarithmic scale) of the computation of homology in different dimensions of:

- (1) Cartesian product of the Eilenberg–MacLane spaces  $K(\mathbb{Z},3)$  and  $K(\mathbb{Z}/5\mathbb{Z},7)$ .
- (2) Cartesian product of the loop space of the Eilenberg–MacLane space  $K(\mathbb{Z},3)$  and the 3-sphere.

(3) Join product of S and the loop space of S, where S is the wedge of a 2-sphere and a 3-sphere.

The figures show that computing times do not improve too much using parallelism by matrices. This is due to the fact that the computation of  $D_{n+1}$  is usually much harder than  $D_n$ , consuming most of the time. Parallelization by columns improves computation times with respect to the version with no parallelism, and in most cases in a notably manner. The exception is in low dimensions, whose execution is almost immediate (less than one second). In those cases it is slower due to the overhead of running subprocesses, serializations and so on. The experiments have been performed on an Intel i7-4790, with 8 logical cores. In principle, one could expect reducing the computing time by a factor of 8, but in this problem is not possible since the computation of each column does not require the same time: some of them are harder and produce the bottleneck. Even so, the improvement is important.

2.2. Improving SageMath via Kenzo. Kenzo relies on the Smith form of integer matrices for its computations. Because of that, it includes a quite optimized implementation. With the appropriate glue code, Kenzo implementation can be used from SageMath. We compared the performance of this approach with the native SageMath implementation (provided by Pari).

It can be seen that Kenzo implementation is clearly faster than native SageMath, both for dense and sparse matrices. The difference for matrices of size 100 is about one order of magnitude. Kenzo has also much more predictable timings.



## References

- 1. Julián Cuevas-Rozo, Jose Divasón, Miguel Marco-Buzunáriz, and Ana Romero, Integration of the Kenzo system within SageMath for new algebraic topology computations, Mathematics 9 (2021), no. 7.
- 2. X. Dousson, J. Rubio, F. Sergeraert, and Y. Siret, *The Kenzo program*, http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/, 1999.
- T. Kaczynski, K. Mischaikow, and M. Mrozek, *Computational homology*, Applied Mathematical Sciences, vol. 157, Springer, 2004.
- 4. J. Rubio and F. Sergeraert, Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs, University of Genova, 2006, http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Genova-MAP-2006-v3.pdf.
- 5. G. Whitehead, *Fiber spaces and the Eilenberg homology groups*, Proceedings of the National Academy of Science of the United States of America **38** (1952), no. 5, 426–430.

University of La Rioja. c/Madre de Dios 53. 26006 Logroño, Spain. Email address: jose.divason@unirioja.es

Universidad de Zaragoza/IUMA. Email address: mmarco@unizar.es

University of La Rioja. c/Madre de Dios 53. 26006 Logroño, Spain. Email address: ana.romero@unirioja.es

#### 4