REPORT ON OpenDreamKit DELIVERABLE D6.9

Shared persistent Memoisation Library for PYTHON/SAGE

MICHAEL TORPEY

Due on	28/02/2019 (M42)
Delivered on	02/04/2019; updated 29/08/2019
Lead	University of St Andrews (USTAN)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/143`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #143 ON 2019-08-30

- **WP6:** Data/Knowledge/Software-Bases
- **Lead Institution:** University of St Andrews
- **Due:** 2019-02-28 (month 42)
- **Nature:** Other
- **Task:** T6.9 (#131)
- **Proposal:** p. 54
- **Final report** (sources)

Persistent memoisation is the computational process of storing a program's outputs and retrieving them later, instead of re-running programs that are guaranteed to return the same answer. This approach may be useful in a wide range of fields but is particularly relevant to mathematicians: indeed, it's common for them to face compute intensive exact problems with small inputs and small outputs.

Results can be stored for future calls to a function in the present program session, in later sessions, or even for other users in different parts of the world. Beside the advantages of recalling old results instead of repeating work, a human-readable cache could also be used by researchers to create a reproducible record of computed results that can be used outside the context of the memoisation system itself.

The aim of D6.9 is to establish a persistent memoisation framework to cache results in Python and GAP across sessions, in a way that is easy to deploy and configure, and allows for results to be shared reliably between different researchers. D6.9 requires a "shared persistent memoisation library for Python/Sage", which is fulfilled by `pypersist`, a new Python library written by the current author. A corresponding GAP package, `Memoisation`, was also written, featuring similar features and with a certain level of interoperability between the two systems. This is a step in the general effort of WP6 toward bridging together Data, Knowledge, and Software notably for Virtual Research Environments.

In this report, we present a review of some existing tools for memoisation, describe the two new packages, show an example of their use together, and give an overview of the future direction of this project.

1. Introduction

Persistent memoisation refers to the computational practice of storing program results whenever they are computed, and then looking up and retrieving them later, instead of re-running programs that are guaranteed to return the same answer. This approach allows users to avoid unnecessary computation, and makes it possible to create an archive of results that can be used for other purposes later.

A memoised function can be seen as a middle ground between a database and a computer program: a function can have a large (perhaps infinite) domain which makes storing all its outputs in a database impractical; but it may also have a long enough run-time that re-computing an output each time it is called is undesirable. The practice of memoisation – that is, recalling results that happen to have been pre-computed, and computing results that are unknown – bridges the gap between these two extremes.

This approach may be useful in a wide range of fields, but it is particularly relevant to mathematics because of the type of problems mathematicians wish to solve. Mathematics is a rare example of a field with algorithms that typically require a lot of processing, but have a small input and small output.

Many algorithms in the real world have a large input and a small output – for example, the problem of determining which streamed TV show to recommend to a viewer takes as its input a huge amount of data about past choices consumers have made, and returns a single title as its output. The output of this function can be stored easily, but so much time would be spent processing the large input, and the same input would be repeated so infrequently, that this type of memoisation would have little advantage.

Conversely, many algorithms have a small input and a large output – for example, rendering a plain text file on a screen takes a small amount of data as an input, but creates an image which is likely to be large if stored. In this case, the input can be dealt with quickly, but memoising the output using a disk is likely to be slower than recomputing it from scratch each time.

In mathematics, however, we frequently encounter computational problems with both small input and small output. For instance, we may wish to know whether a given permutation group G is simple; the input may be two or three generators for G, which between them use only a few bytes on a computer, and the output is a single boolean. But despite the small sizes of these data sets, the algorithm may take a long time to complete, and may use a large amount of memory while in progress. Thus, memoising results of this type will use very little disk space, require very little processing, and may avoid a huge amount of work when recalling previously computed results.

It is also worth noting that pure mathematical problems are likely to have discrete inputs and outputs, and are therefore unlikely to suffer from numerical noise. This makes them even more suitable for memoisation, since we do not have a high density of almost-equal inputs that nonetheless would have to be memoised separately.

Furthermore, there is no reason that a memoisation cache should be limited to a single user. A memoisation framework could save results to a shared directory for immediate use by other researchers connected by a network, or could even use an online database for the cache, so long as sufficient precautions were taken to avoid reading and writing clashes.

Besides the obvious advantages of recalling cached results instead of repeating work, a memoisation framework that allows a human-readable cache could be used by researchers to create a reproducible record of computed results that can be included in papers, or shared with collaborators that have no knowledge of the memoisation system.

The aim of OpenDreamKit Task 6.9 is to establish a persistent memoisation framework to cache results in Python and GAP across sessions, in a way that is easy to deploy and configure, and allows for results to be shared reliably between different researchers. Deliverable 6.9 requires

a "shared persistent memoisation library for Python/SAGE", which is fulfilled by PYPERSIST, a new Python library written by the current author. The corresponding software for GAP has also been created, in the form of GAP's MEMOISATION package, which has approximately the same features as PYPERSIST, and aims to be as compatible as possible with it.

In this report, we will present a review of some existing tools for memoisation (Section 2), describe the new Python library PYPERSIST and the new GAP package MEMOISATION (Section 3), show how to what extent they can be used together (Section 4) to share data across systems, and give an overview of the future direction of this project (Section 5).

2. EXISTING TOOLS

In this section we will consider some tools that already exist for memoisation, in Python, SAGE, and GAP. Some of the tools we discuss are only intended for a subset of the complete memoisation process, while others are more full-featured but lack some important desirable features. A comparison of these tools is shown in Table 1.

	Updated in last year	Python versions	Function decorator	Memory caching	Disk caching	Database caching	Method support	Compiled function support	Custom keys	Custom pickling	Metadata	SAGE support	Compiled
SAGE `func_persist`	✓	SAGE	✓	✓	✓							✓	
SAGE `cached_function`	✓	SAGE	✓	✓				✓	✓			✓	✓
SAGE `cached_method`	✓	SAGE	✓	✓			✓	✓	✓			✓	✓
`persist`		2/3		✓	✓								
`PyMemoize`	✓	2/3	✓	✓	✓		✓						
`redis-simple-cache`		2/3	✓	✓	✓	✓	✓						
`dogpile.cache`	✓	2/3	✓	✓	✓	✓	✓		✓				
PYPERSIST	✓	2/3/SAGE	✓	✓	✓	✓	✓		✓	✓	✓	✓	
GAP operation/attribute	✓	—		✓			✓	✓					
MEMOISATION	✓	—		✓	✓	✓	✓	✓	✓	✓	✓		

TABLE 1. Comparison of memoisation tools in Python/SAGE and in GAP

2.1. **GAP**

GAP functions can be installed as methods for an object's *operations* and *attributes*, and if they are, their results are stored for the duration of the current GAP session. However, these results are never saved to disk unless the session itself is stored, and the option is only available for methods of an object. Since there is a lack of any other well-established tools for memoising functions, the MEMOISATION package was created. It contains all the features described in Table 1, except for those specific to Python, and for the fact that it is written in pure GAP and is therefore not compiled. The features are shown in more detail in the next chapter.

2.2. SAGE

There exists in SAGE a simple function decorator, `func_persist`, which can be applied to a function to save its output to disk. This is a working memoisation tool, and even this functionality has great practical use, but the tool has very little configurability: the only option that can be given is to specify the directory in which results are stored. SAGE also contains two more configurable tools for memoising functions and methods, respectively named `cached_function` and `cached_method`. These allow a function or method to be memoised with a custom key-generating function, meaning that arguments can be pre-processed in an intelligent way, perhaps to sort arguments, or discard arguments that do not affect the return value of a function. They are also implemented in Cython, meaning that they are compiled rather than interpreted, and perform faster than `func_persist`. They also support compiled functions written in Cython. However, they only support memory caching – that is, they only store results in memory while the current program is running, and do not save results to disk for use in a later session.

These three decorators are useful, but limited. Not only do they lack certain options such as database caching and custom output pickling, but they rely on other parts of the SAGE system, and therefore cannot be used more broadly in generic Python programs. Python memoisation tools exist outside SAGE that can in principle be applied to SAGE.

2.3. Python

Many memoisation tools are available on PyPI, each with a slightly different philosophy and set of features. Table 1 summarises four examples of these, indicating each one's features and status: `persist` and `redis-simple-cache` have not been updated in several years, while `PyMemoize` lacks even the custom key function provided by the SAGE tools.

Perhaps the most promising of the existing tools is `dogpile.cache`, an API for caching function and method outputs to a variety of backends; this is a well-established, full-featured system, but it lacks certain features such as custom result pickling – particularly useful to mathematicians who may want to store output using OpenMath, or in some human-readable form for sharing.

There would be an argument for modifying and improving `dogpile.cache` to add these features, rather than creating a new tool. However, given the wide scope of the `dogpile` project, which is largely focused on thread management, it was decided to proceed with a new tool that could be adapted more easily to suit the specific requirements of the project, such as bespoke support for SAGE-specific objects, and compatibility with the GAP MEMOISATION package which was developed in parallel.

To fulfill these requirements, the PYPERSIST package was created. It runs in standard Python versions 2 and 3, as well as in SAGE, and it contains all the features mentioned in Table 1 except those features related to Cython. More information about PYPERSIST is given in Section 3.

3. PYPERSIST AND MEMOISATION

To address the requirements of this deliverable, two closely related pieces of software were created: the Python package PYPERSIST and the GAP package MEMOISATION. They have approximately the same features, as shown in Table 1, and they were designed to be compatible, as explained in Section 4.

3.1. Features

We now summarise some of the features of PYPERSIST and MEMOISATION. The detailed workings of the packages are outside the scope of this report, but full documentation is available at `https://pypersist.readthedocs.io` and `https://gap-packages.github.io/Memoisation/`. The features of the packages are also demonstrated in interactive notebooks hosted on Binder and linked in the respective readme files.

Usage. PYPERSIST provides a function decorator, `persist`, which is used to memoise a function. After this decorator is imported, a Python function can be memoised by simply typing `@persist` above it. In GAP, which does not have function decorators, the MEMOISATION package instead provides `MemoisedFunction`, which simply takes a function and returns a memoised version of it. This minimal setup produces memoisation with safe defaults, and allows users to exploit these packages with no additional knowledge of their features. However, further customisation is possible by adding arguments to the decorator in the form `@persist(...)` or by providing an options record as the second argument to `MemoisedFunction`. Examples are shown in Figure 1.

```
@persist(funcname="power_number",
         cache="file://my_results",
         pickle=str,
         unpickle=int)
def power(x, y):
    return x ** y
```

```
power := MemoisedFunction({x, y} -> x ^ y,
      rec(funcname := "power_number",
          cache := "file://my_results",
          pickle := String,
          unpickle := Int));
```

FIGURE 1. Memoising a function in PYPERSIST and MEMOISATION

Configurable cache location. By default, results in PYPERSIST and MEMOISATION are saved respectively to the `persist/<funcname>` and `memo/<funcname>` subdirectories of the current working directory. However, by providing a string as the `cache` argument of `@persist` or `MemoisedFunction`, this location can be changed (see Figure 1).

Configurable function name. In order to avoid confusion between two different functions with the same name, a `funcname` argument can be supplied, which should be a string that uniquely identifies the function. This string could include a version number, author name, or anything else required to ensure uniqueness.

MongoDB cache. Support is provided for caching results in a MongoDB database, via the REST interface provided by Python's `eve` package. If a `cache` argument is provided which starts with `mongodb://`, then it is interpreted as the address of a MongoDB server which can send and receive results for the function, and that server will be used instead of saving results locally. Such a server can be set up using the `mongodb_server/run.py` script distributed identically with PYPERSIST and MEMOISATION.

Custom key function. Each entry in a function's cache is stored using a unique key based on the arguments supplied to the function; two sets of arguments should have the same key only if they produce the same output. By default, PYPERSIST uses a tuple of the arguments, with their names, sorted alphabetically, and with any default arguments removed; this allows for equivalent calls such as `foo(x=3, y=5)` and `foo(y=5, x=3)` to be treated equally. Since GAP does not support keyword arguments, MEMOISATION simply uses a list of the arguments. However, in both systems, users can specify a `key` argument, which should be a function that takes the same arguments as the memoised function and produces a key based on them. A developer could thus choose to ignore arguments that control, for example, verbosity, or other options that do not affect the function's return value.

Custom hashing. When a result is stored in the cache, it is stored using a hash of its key. This allows a record to be looked up quickly, and perhaps removes the necessity of storing the keys at all. By default, a key is hashed using its SHA-256 encoding, which makes it extremely

unlikely that a hash collision will ever be encountered. However, a custom hash function can be specified using a `hash` argument to the decorator, and in this case it will be used to produce a hash instead of the default method. If an injective hash function is chosen, then its inverse can be specified using the `unhash` argument; this allows hashes to be double-checked, and a list of previously-encountered keys to be produced.

Storing keys. By default, a key is not stored after it has been hashed, and correctness is assumed due to the vanishingly small probability of an SHA-256 hash collision. However, for complete correctness, the packages support an argument `storekey` which, if set to true, stores keys along with results, and checks them when looking up results, raising an error if a hash collision is detected. This also allows a user to iterate over the stored keys of a cache, as mentioned below.

Custom pickling. In order to store an object in a cache, it needs to be preserved on disk or in a database. For this purpose, we require sometimes complex objects to be *pickled* and *unpickled* – that is, converted to a string for storing, and converted back when retrieved later. The default methods use Python's `pickle` module (or, if appropriate, SAGE's own pickling functions) or GAP's `IO_Pickle`, and a base 64 encoding to produce strings that can be written without invisible or difficult-to-display characters. However, like much of these packages' functionality, custom `pickle` and `unpickle` functions can be specified that do this a different way. This can allow results to be stored in a human-readable way which can then be used outside Python and GAP, or even to be stored using a format such as OpenMath for use in a completely different computer system.

Manual cache interaction. It may be necessary at some time to modify a memoisation cache manually – perhaps to remove an inaccurate result, or to add a result that was found before the cache was created. This can be done directly in both packages. In PYPERSIST, a memoised function `foo` has an attribute `foo.cache` which can be used much like a dictionary, reading, setting and deleting entries as desired. Furthermore, if `storekey` is set to true, or if an `unhash` method is provided, `foo.cache` is an iterable object with attributes `keys`, `values` and `items` that can be used in loops. Similarly in MEMOISATION, a memoised function `foo` has a component `foo!.cache` which satisfies `IsLookupDictionary`; it can therefore be used with the GAP functions `AddDictionary`, `KnowsDictionary` and `LookupDictionary`, if a user wishes to manipulate it directly.

3.2. Software engineering

Both PYPERSIST and MEMOISATION were written using modern open-source software technologies and practices, with an emphasis on clarity of understanding and ease of use. The report on OpenDreamKit D1.5 contains, in Section 4.3.2.1, a checklist of software engineering best practices. The packages fulfill all the requirements in that list, as summarised in Tables 2 and 3.

Version control	✓	Git
Tests	✓	*pytest* suite with 98% code coverage
Automated tests	✓	
Continuous integration	✓	Travis runs test suite for every commit
Automatic building of releases	✓	PyPI release script

TABLE 2. Software engineering good practice checklist for PYPERSIST

Version control	✓	Git
Tests	✓	GAP test suite with 98% code coverage
Automated tests	✓	
Continuous integration	✓	Travis runs test suite for every commit
Automatic building of releases	✓	`ReleaseTools` for GAP (see D5.15)

TABLE 3. Software engineering good practice checklist for MEMOISATION

3.3. **Dissemination**

The report on OpenDreamKit D1.5 also contains, in Section 4.3.2.2, a checklist of best practices for dissemination. The packages also fulfill all of these requirements, as summarised in Tables 4 and 5.

Host code publicly	✓	`https://github.com/mtorpey/pypersist`
Reference Manual (APIs)	✓	`https://pypersist.readthedocs.io`
Tutorial (for beginning users)	✓	Examples section in readme, and Binder demo with extensive annotations: `binder/demo.ipynb` (linked from readme and manual)
Examples	✓	
Live interactive online demos	✓	
Support mechanisms	✓	Github issues
How to cite the output?	✓	Explained in readme and manual
Installation mechanism	✓	Installation via *pip*, explained in readme and manual
High level description accessible to non-experts	✓	In readme, manual and Binder
URLs/Blog/etc to and from OpenDreamKit project	✓	Links to OpenDreamKit in readme, in manual, and on PyPI distribution page
Grant acknowledgements	✓	Acknowledged in readme and manual
Open Source license	✓	GPL v2 or later
Workshop	✓	Demo and discussion at *Free Computational Mathematics* (CIRM, Luminy, Month 42)
Engaging users	✓	

TABLE 4. Dissemination good practice checklist for PYPERSIST

Development of both packages was open from the start, available publicly on Github, including their full commit histories. The issue trackers on their Github pages have been used to track upcoming new features, as well as bugs and other problems. All documentation, tools, and setup are included directly in the Git repository, so that everything is kept in one place and can be tracked as a single unit.

The packages' documentation is mostly generated directly from the source files, using Sphinx and GAPDoc/Autodoc respectively. This means that code is documented where it is written, making the code easier to understand and avoiding the need to duplicate documentation with extra comments. Numpy-style docstrings are used throughout PYPERSIST – this widely used standard was chosen for its minimal syntax, which allows the strings to be easily read and understood in plaintext without any processing or compilation. A readme is included in each project, written in Markdown, with a simple two-sentence introduction, a guide to installation, and a small number of examples showing how to get started. The manual for each package is hosted online, PYPERSIST on *ReadTheDocs* and MEMOISATION on its own website, both of

Host code publicly	✓	`github.com/gap-packages/Memoisation`
Reference Manual (APIs)	✓	`gap-packages.github.io/Memoisation`
Tutorial (for beginning users)	✓	Examples section in readme, and Binder demo at `binder/demo.ipynb` (linked from readme)
Examples	✓	
Live interactive online demos	✓	
Support mechanisms	✓	Github issues
How to cite the output?	✓	Explained in readme and on website
Installation mechanism	✓	Installation via `PackageManager`, explained in readme and manual
High level description accessible to non-experts	✓	In readme and manual
URLs/Blog/etc to and from OpenDreamKit project	✓	OpenDreamKit link and logo in readme
Grant acknowledgements	✓	Acknowledged in readme
Open Source license	✓	BSD 3-clause licence
Workshop	✓	Demo and discussion at *Workshop on Mathematical Data* (Cernay, Month 48)
Engaging users	✓	

TABLE 5. Dissemination good practice checklist for MEMOISATION

which are linked from the appropriate Github page. In this way, the documentation is accessible, clear, and maintainable.

The minimal examples in the readme are supplemented by the Jupyter notebooks included with the projects. These notebook can be loaded locally by anyone with a Jupyter installation, but for better accessibility they are also hosted on Binder, with the address linked at the bottom of each readme, as well as in a badge at the top. The PYPERSIST demo loads the most recent version of PYPERSIST, and therefore also acts as an additional test suite, since it requires the package to be in a working state in order to demonstrate its features. The outputs of the executable cells are not saved into the notebooks – instead it is left to readers to execute the cells themselves, producing an interactive experience which may encourage the reader to experiment, and may lead to a better understanding of how the package works. The PYPERSIST notebook was demonstrated at the *Free Computational Mathematics* conference held in CIRM, Luminy, France, in Month 42, prompting a discussion and feedback from users. A demo of both packages working together was shown at the *Workshop on Mathematical Data* held in Cernay in Month 48 (it approximately followed Section 4) also prompting discussion.

In order for the packages to be used, they must be easy to install. The PYPERSIST package is hosted on PyPI at `https://pypi.org/project/pypersist`, making installation as simple as typing `pip install pypersist` for any user connected to the internet. The MEMOISATION package can now be installed using GAP's own package manager, another new piece of GAP infrastructure from OpenDreamKit. Each installation method is explained in the appropriate project's readme, and the procedure for PYPERSIST is also on the first page of its manual.

4. INTEROPERABILITY

By default, PYPERSIST caches are not compatible with MEMOISATION caches. GAP and Python have different data types, and the default `pickle` and `hash` functions are different

between the two packages. For this reason, the default cache locations are separate when caching to disk, and different namespaces are used when writing to a database.

However, it is possible to customise a cache in a variety of ways, as described in Section 3.1, and it is certainly possible to specify options that allow a PYPERSIST cache and a MEMOISATION cache to share results with each other. The following example was presented as a demo at the *Workshop on Mathematical Data* mentioned above.

4.1. **Example**

Consider the following fragment of Python code. It defines a function `prime_factors` that performs deliberately badly, including a 2-second sleep, but should correctly calculate the prime factors of an integer.

```
from time import sleep
from pypersist import persist

@persist(
    cache="file://shared",
    key=lambda n: n,
    hash=str,
    pickle=lambda L: "\n".join(map(str, L)),
    unpickle=lambda s: [int(n) for n in s.split("\n") if n!=""],
)
def prime_factors(n):
    factors = []
    d = 2
    while n > 1:
        while n % d != 0:
            d += 1
        factors.append(d)
        n /= d
    sleep(2)
    return factors
```

The function itself has been memoised by being decorated with `@persist`, and several custom options have been specified:

- `cache` defines the location of the cache as being inside `shared/` in the current directory;
- `key` defines the single input `n` as a unique key for each memoised result;
- `hash` is the function that simply writes an integer out as a string in base 10 (this, with ".out", will be the filename);
- `pickle` takes the list of prime factors, and prints each one to a separate line in a text file, also in base 10;
- `unpickle` turns this file back into the list of integers.

This will result in quite a human-readable cache: results will be stored inside the present directory, in `shared/prime_factors/`, and each result will be saved in a single file with `<n>.out` as the filename, and one prime factor printed on each line. This cache could be used as an educational tool, and bears no traces of having been created by PYPERSIST at all.

GAP has a much better method for computing prime factors: the library function `Factors`. We memoise this as follows.

```
LoadPackage("Memoisation");

prime_factors := MemoisedFunction(Factors, rec(
    funcname := "prime_factors",
    cache := "file://shared",
    key := IdFunc,
    hash := String,
    pickle := L -> JoinStringsWithSeparator(List(L, String), "\n"),
    unpickle := str -> List(SplitString(str, "\n"), Int)
));
```

Note that all the same memoisation options have been specified: `cache`, `key`, `hash`, `pickle` and `unpickle` have been translated into GAP code and included just as in the PYPERSIST example. However, note that an additional argument `funcname` has been specified; this is so that we use the directory `shared/prime_factors/` instead of the default `shared/Factors/`.

These two functions will now load and store each other's results. If we compute a new example in Python first, it will take a whole 2 seconds to complete, but if we calculate it first in GAP and then in Python, it will appear to complete instantly, as the overhead of reading a list of integers from a file is negligible. This is not only an example of how flexible a cache can be in these two packages, but also an example of how different mathematical systems can communicate with each other and share results.

5. FUTURE DIRECTION

Although PYPERSIST and MEMOISATION now have many features and are well-tested, they are still in development. It is hoped that as more developers begin to use them, they will provide feedback that will inform development further, whether by revealing bugs, or by requesting new features. There are already several features that are planned, and work will continue on this project as time goes on. In this section we describe some aims for the future direction of persistent memoisation in GAP, SAGE and Python.

5.1. Features

CouchDB. Support currently exists in PYPERSIST and MEMOISATION for a MongoDB database backend. This works well, but there are certain features offered by the CouchDB system that might also be useful. CouchDB emphasises ease of setup, and allows information to be added and retrieved from the database via HTTP, with a human-readable web-based front-end that would allow human-readable results to be shared instantly on computation via a website. It is likely that implementing this would require little deviation from the existing code for MongoDB, and could result in big improvements in ease of use, ease of setup, and interoperability.

Better metadata. PYPERSIST and MEMOISATION have the capacity to store metadata, which may be useful for storing information about a given computation. However, no metadata is stored by default, and metadata is not validated in any way when a result is loaded from the cache. Some decision should be made on what metadata should always be stored, and functionality should be added that allows this metadata to be used where relevant, for example by allowing a cache to be filtered by date or user. Some useful fields could be where and when a result was originally computed, comments from the user or the author of the function, and information about the system that produced the result.

Provenance tracking. One use of metadata would be to record where a given result came from, allowing researchers to evaluate the trust they are willing to place in it. A limited form of this can be achieved by sharing results using a version control system such as Git, in which the timestamp and author of a commit give some useful information. But a more full-featured provenance-tracking system could be added in the future, recording the exact time and place that a computation was performed, as well as the user that called it, and version numbers for both PYPERSIST/MEMOISATION and the function being memoised.

Permissions. In a multi-user memoisation cache, it would be useful to have explicit support for permissions handling. Most of the functionality for this could be delegated to the backend – for example, any database system has support for user permissions, and Git repositories can be set up to accept certain requests from only certain users – but it would be useful if PYPERSIST

and MEMOISATION handled this properly instead of simply raising an error when an interaction is unsuccessful.

Verbosity. Results in PYPERSIST are stored and retrieved silently, and users are not made aware of its existence in any way except when errors are raised (for example due to a hash collision or HTTP error). The MEMOISATION package, on the other hand, displays messages to the screen showing its progress, so that interested users have a record of exactly what is being written and read at what time and in what location. Different levels of verbosity are also available, showing different levels of information. This verbosity should be included in PYPERSIST as well.

SAGE integration. As shown in Table 1, there exist three different tools already in SAGE for memoising data. Now that PYPERSIST has overtaken `func_persist` in terms of features, it is possible that it could replace this tool entirely; with some additional work (support for compiled functions, optimization), it could replace `cached_function` and `cached_method` as well. With careful integration and testing to ensure that there were no regressions, PYPERSIST could be imported and used instead of those tools, possibly with the addition of some SAGE-specific options that would allow it to be used as effectively as possible. However, PYPERSIST should remain a generic Python package that can be used outside as well as inside SAGE.

Search by property. The core behaviour of a memoisation system is to look up a single input and return a single output if one exists. However, once a cache with a number of results has been built up, a user may wish to search through it to find all input–output pairs that satisfy a particular property. For instance, if a boolean function called `is_prime` is memoised, and has been used to test the primality of several integers, a researcher who desires prime numbers might search through the cache for all inputs that returned `true`. This can be achieved using the current system, but some kind of indexing might be helpful, to allow answers to be returned more quickly.