Energy tracker for shared servers

Nathan Lemmers, Julien Lefèvre
lemmers@insa-toulouse.fr
julien.lefevre@univ-amu.fr

Abstract

In a context of ecological crisis, it is becoming necessary to measure the impact of our various actions on the environment, with the aim of reducing our co2 emissions. The first step in this process is the ability to efficiently and accurately measure the energy we spend.

In our case, we are interested in the emissions produced by our use of digital technology. As our use of new technology increases, it becomes necessary to measure its real impact, to have concrete data on what needs to be or can be reduced.

Some people have already looked into the matter. And there are now a number of Python modules that offer an evaluation of the energy expended during the execution of a Python code. However, to the best of our knowledge, none has yet been evaluated for its ability to accurately measure energy on a server used by several people simultaneously.
In this study, we'll focus on the 3 main existing digital power meters, *codecarbon* [2], *eco2ai* [3] and *carbontracker* [4].
We will also take a look at *GreenAlgorithm* [5], a website that enables the same type of measurement, but online, by entering key data such as execution time, CPU and GPU model, and so on.
The aim of our study is to offer new strategies for efficient measurement on a shared server, as well as a module that applies them to compare their effectiveness.

1 Existing application

There are now many applications available for assessing carbon impact. These digital power meters each have different ways of working.
We will concentrate here on 4 existing methods, which are described in detail and measured in the rest of the article. The first three, *eco2ai* [3], *codecarbon* [2] and *Carbontracker* [4] are Python libraries, which we need to add to our source code to get a measurement. These 3 methods divide their calculation of energy spent into three distinct parts, CPU, GPU and RAM consumption. Here are the details of how they measure, according to [1] and their documentation.

About GPU power consumption, these three methods use Python's pynvml library. This provides the power consumption of NVIDIA graphics cards. If the GPU is not from NVIDIA, the consumption is estimated at 0. It is therefore necessary to have this kind of GPU to get measurements.

Regarding RAM consumption, *codecarbon* and *eco2ai* use the reference value of 0.375W/GB, and they multiply it by the RAM used. The total ram information is retrieved via the *psutil library*v . *Carbontracker* uses RAPL files, but measures the total energy of the available RAM, not that used by the process.

About CPU consumption, *eco2ai* calculates the CPU usage factor via the *os* and *psutil* modules, and then multiplies it by the Thermal Design Power (TDP) of the current processor. If the processor is not in the database, the default value for the TDP is 100. *Carbontracker* and *codecarbon* both use RAPL files. To access this data, the processor must be Intel and the user must have an administrator access. If this is not the case, *codecarbon* will search its database for the TDP corresponding to the

processor, and if this is not available either, it will use the default value of 85 W, with a usage factor of 50%. *Carbontracker* won't measure CPU consumption if it doesn't have access, but will display a message in the terminal.

The last digital wattmeter we'll be talking about is *GreenAlgorithm*. This one is special in that it doesn't have to be implemented in the code, since it's an online tool. It is accessible on the Internet, and does not itself measure process consumption because it assumes that power will remain constant over time. This assumption is often correct in the case of deep or machine learning. Indeed, it is necessary to manually enter a great deal of information, such as GPU and CPU model, RAM used, execution time, and whether the code was running on a local server, a personal PC, or in the cloud.
It works with a database linking CPU and GPU models to their TDP. It then multiplies them by the usage factor, which we have to enter ourselves by hand, if we don't want the default value of 100%. About the RAM memory, the coefficient is 0.3725W/GB.

All this information is summarized in a comparative table in the 4.2.3 section, which also details the module we're proposing.

NB: Carbontracker will only work if it can measure at least one CPU or GPU.

2 Problems and suggested solutions for use on a shared server

The various modules presented in the previous section are particularly effective. However, they weren't designed to work on shared servers, which are fraught with complications. The remainder of this section will give a non-exhaustive list of the problems that arise in this context, and how we've tackled them when creating our own module.

First of all, for the module to be easily implementable and usable by the user in the server, it must not have to be added to the Python code that is intended to be executed. For this reason, we believe it's best to create an executable file that will itself launch the user's command. In our case, we can execute our monitoring procedure with the command to be executed in bash as argument. More concrete examples of use are given in appendix A.2.

2.1 CPU energy

The use of RAPL files to measure the energy consumed by the processor is highly accurate and efficient, but it requires the user to have administrator access (sudo). However, it is rarely the case when used on a shared server. It is therefore necessary to work with the usage factor and a processor database to make the code work.

To this end, we took a CPU usage factor measurement every 10 seconds via *psutil*, and averaged it over the overall execution time.
This method may seem efficient, but is not sufficient. In fact, *psutil* provides a global CPU usage factor, but we only want to measure the consumption of the code we're executing. In addition, sorting the usage factor with a certain PID[1] isn't suitable for us, because since we're launching the user's code on a subprocess and it's impossible to predict the pids of subprocesses it will create.

We therefore decided to search for the usage factor using a user-oriented method. To do this, we filter the *psutil* results by keeping only those corresponding to our user.
Here's how we measure it in python:

```
user_processes = [p.info for p in psutil.process_iter(['pid', '
    username', 'cpu_percent']) if p.info['username'] ==
    current_user]
cpu_total = sum(p['cpu_percent'] for p in user_processes)
cpu_cores = psutil.cpu_count(logical=True)
cpu_usage = cpu_total / cpu_cores if cpu_cores > 0 else 0
```

[1] A PID is a temporary number assigned by the operating system to a process or service

We can then calculate average processor usage, and measure its power consumption:

$$CPU\_consumption = \sum_{i=1}^{n} \frac{usagefactor_i}{n} * TDP * number\_of\_core * time$$

It should be noted that factor usage does not give a precise result on energy spent, but it does give an upper bound on consumption. This approximation is therefore a compromise between precision and energy consumption.

2.2 GPU energy

We can use *pynvml* to measure GPU consumption directly. However, the fact that our user works on a graphics card doesn't mean he's the only one working on it.
A method similar to that of CPU usage factor was considered, But the measurements we had were still far from reality. It seems that the GPU's usage factor is not correlated with its percentage of maximum resource utilization. Many operations are possible on a GPU, and an x% utilization of the graphics card is therefore not always equal to x% of its TDP. We observed the 99% usage factor with power consumption far from the limits of our graphics card.

Thanks to *pynvml* librairy, we can retrieve the energy expended in watts by the GPU. We can therefore obtain the average power consumption by taking a measurement every 10 seconds.
However, we're still looking at the power consumption of our user only. We've therefore decided to remove the GPU's passive consumption, so that we only have the energy impact of the code. Indeed, if we take into account the GPU's passive energy, it will be considered even in the case of code not using a GPU, if a graphics card has been allocated to the node.
It should be noted that the GPU's passive power consumption will still have to be taken into account when balancing server usage.

$$GPU\_consumption = \\ (average\_GPU\_consumption\_by\_pynvml - standby\_mode\_GPU) * time$$

2.3 RAM energy

About the energy consumed by the RAM memory, the modules presented in part 1 offer a quick calculation of the energy expended by the RAM. To do this, some multiply the RAM used by the system by their coefficient in W/GB. Indeed, we can not use the total ram in a large cluster with 400GB of RAM memory. Beacause we can't attribute the consumption of all the RAM to our current user, because he's not the only one using it
We decided to allocate only the RAM actually used by our user, using the same system as for the CPU, taking the average RAM usage of the current user. Here's the way we measure every 10 seconds to make an average:

```
user_processes = [p for p in psutil.process_iter(['pid', '
    username']) if p.info['username'] == current_user]
total_ram += sum(p.memory_info().rss / (1024 ** 3) for p in
    user_processes)
```

2.4 Static node consumption

Static node consumption is not the same as idle node consumption. Indeed, according to our measurements and to [11], it seems that when a command is launched on a node, a fixed consumption appears.
In the same way as for the GPU, we would tend to exclude passive node consumption as well. However, this is not of the same nature as the GPU's, as we observed that nodes were only powered when we launched an action on them, whereas the GPU consumed power passively all the time. To make this discovery, we used the power meter of grid5000, a computing cluster, and ran the stress command to measure CPU consumption as a function of factor usage. Our aim was to confirm the hypothesis of linearity in CPU consumption, but this led us to measure this node "activation", which has been demonstrated and highlighted by [11].

Here's the CPU consumption curve on grid5000's gemini-1 node, the hyperthreading is still activate, to simulate real-life use cases:

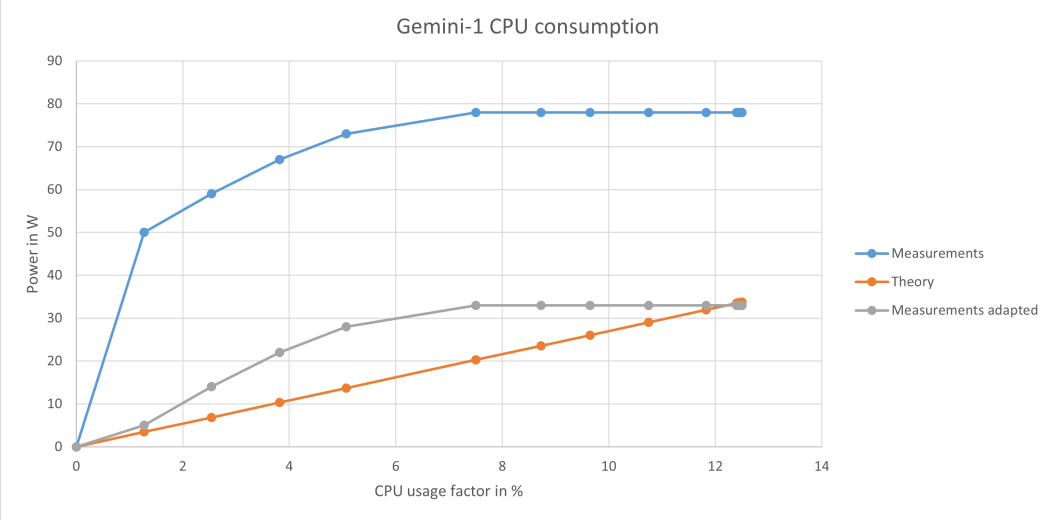

Figure 1: Gemini-1 CPU consumption

The grey line is the measured power consumption if the power measured as static is removed. Here, we have removed 45W. The assumption of linear CPU consumption is therefore an acceptable approximation. Numerous other measurements have been made to determine whether this static power consumption appears systematically. Here's the measurement for Neowise-10, another grid5000 node:

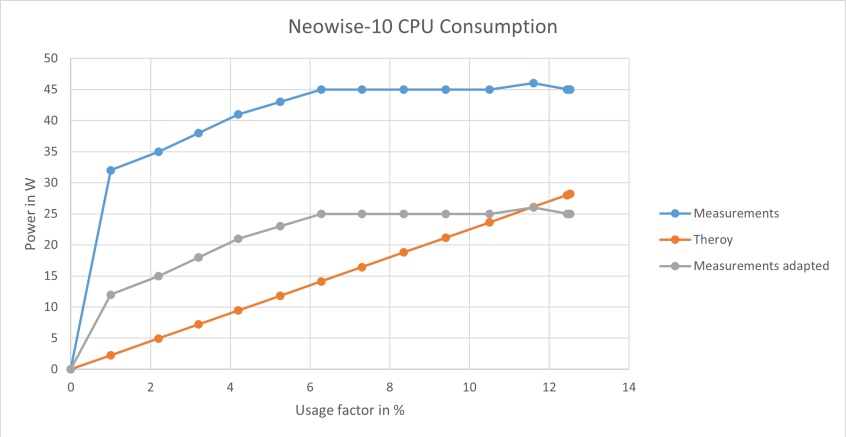

Figure 2: Neowise-10 CPU consumption

Note that we could not exceed a 12.5% usage factor on these shared nodes, because we asked for the use of a GPU. In addition, we see the appearance of a plateau halfway through our measurements. This is due to hyperthreading [12]. In order to confirm our results, we redid the measurements on the entire nodes, without requesting access to a GPU, and deactivated hyper threading. It should be noted that current users may not have the option of disabling hyperthreading, so these measurements are theoretical.

Final measurements was made on a Orion-1 and Gemini-1 :

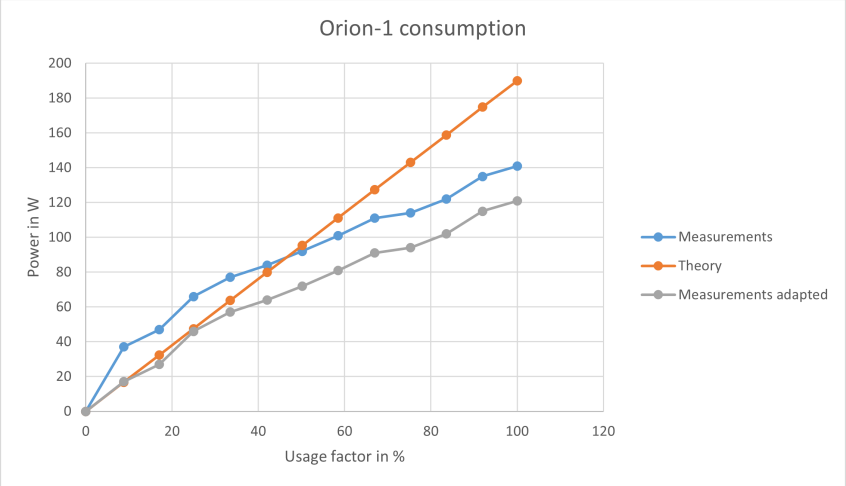

Figure 3: Orion-1 measures

This measurement shows that the node's passive power consumption is still observable, at around 20W. We can also see that the TDP is, as previously explained, a real upper limit to our consumption.

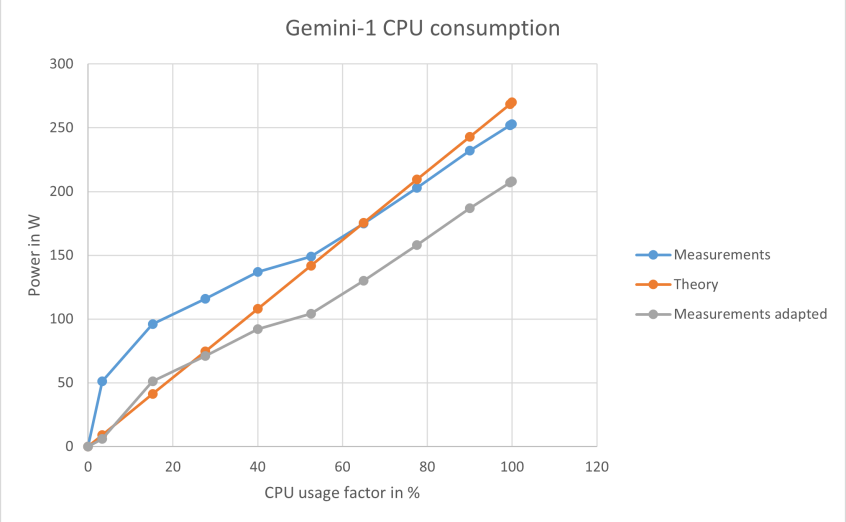

Figure 4: Gemini-1 CPU consumption without hyperthreading

By keeping the same value of 45W for static consumption, we find a coherent and adapted measurement below its theoretical maximum limit.

In conclusion, we can see that the problem of passive node consumption is not to be taken lightly, because it has a particular impact when CPU utilization is light. In addition, when we are accessing to a GPU in the node, the CPU utilization is limited to 12.5%, which can be a source of considerable inaccuracy. Indeed, at this percentage, and with the hyperthreading, the node's consumption represents almost the same as our CPU consumption. Fortunately, as this power is constant, it is negligible for resource-intensive codes.
In addition, it seems impossible to take into account with a Python module, since this consumption depends on many factors that depend on the node or the working environment.
In the same way as for passive GPU consumption, we decided not to take it into account in our module, but since this consumption is only "activated" when a command is run, it would seem necessary to find a way of measuring it.

3 Our module

In order to test solutions to the problems highlighted above, we have developed a module. This is functional and open source at : `https://github.com/nathanlemmers/EnergyTracker`
However, this module is not intended to replace the old ones, but rather to provide a basis for reflection and enable us to make measurements to confirm and compare our accuracy against existing solutions.

3.1 Code operation

3.1.1 Libraries used

Here's the list of libraries used in monitor.py:
threading, to run measurements in parallel and not slow down the code. *time* for temporal measurements, *psutil* and *pynvml* for CPU and GPU information/measurements. *eco2ai* [3] and *codecarbon* [2] for co2 measurement modules. Then *os* for creating files and folders, *subprocess* for launching user code, *sys* for accessing arguments, *cpuinfo* for CPU data (such as name), *datetime* for measuring execution time, *pandas* for power consumption calculations and *warnings* for error mess. Finally, *FR*, from [6], the librairie that provides real-time carbon intensity.

The *codecarbon* and *eco2ai* libraries enable independent co2 measurements. These two libraries also work in thread mode, and measure a given code. They allow us to have reference measurements, and to use them in case of malfunction of our measurements.
Our code includes a *start()*, which will launch the thread, and a *stop()*, which will stop it. We take advantage of this to launch the *codecarbon* and *eco2ai* measurements.

3.1.2 Measuring data for GreenAlgorithm

GreenAlgorithm [5] is an energy calculator for code execution available on the Internet. It is recognized [1] as one of the most accurate, with no need to be added directly to the code, making it highly adaptable. On the other hand, it requires you to manually enter values such as components, available memory, and algorithm execution time.

We've therefore decided to add the "GreenAlgorithm.txt" file, which will return all the values to be entered manually, along with the link to the website, so that the user can enter them easily.
To do this, we had to retrieve certain values. Most of these are fairly straightforward to obtain, but the difficulty lies mainly in the CPU and GPU utilization factor, as well as the average RAM memory used.
However, as we saw in the previous sections, our aim is to work mainly on servers. The way in which we retrieved these values was detailed in part 2.
Once we've retrieved these values, we can enter them in the file so that the user can then do it himself on the site: `http://calculator.green-algorithms.org/`.

We now understand quite clearly the main deficiencies of this method. First of all, it's a bit of a waste of time to have to enter the values manually, but that's not the most important thing.
GreenAlgorithm's database isn't complete, and although there are many choices, it's sometimes impossible to choose your own CPU or GPU. It's also not possible to enter the TDP values ourselves, so we're forced to try and choose a "close" component, without being convinced that they have the same characteristics.
Finally, GreenAlgorithm's main default is its use of the usage factor to measure the impact of GPUs. As we saw earlier, a GPU's usage factor is not directly correlated with its TDP usage ratio. This will lead to large overestimates of power consumption, which will be observed in part 4.2.

That's why we thought we had to adapt/improve GreenAlgorithm for our situation and our server. These explanations can be found in the next section.
Note that the "GreenAlgorithm.txt" file will only contain the information for the website if our adapted GreenAlgorithm module does not have the necessary data to operate.

3.2 GreenAlgorithm adapted to our system

This section is dedicated to the GreenAlgorithm_adapted (*GA_adapted* function), which have been developed to improve the efficiency and calculation of our greenhouse gas emissions, by calculating the energy expended for a code. To achieve this, it accesses the data contained in the "data_GA.csv" file. This file contains all the constants required to calculate energy consumption such as the CPU's TDP values. For more details on how to use this file, please refer to the appendix.

3.2.1 Real-time calculation of carbon intensity

The first decisive point in our code is the calculation of carbon intensity. Indeed, in most current modules, including *GreenAlgorithm*, *codecarbon*, *carbontracker* and *eco2ai*, a default value for each country is used to calculate the gco2/kWh equivalent. The average value in France is between 50 and 55 gco2/kWh.
However, these values are imprecise, and this reference value is extremely variable. **We therefore decided to calculate it ourselves for each code launch.**

To do this, we're using part of the code of electricity map [6]. Specifically, the "FR.py" file, which allows us to retrieve usage data for each energy generator in France. For example, how much energy is generated by nuclear power, wind power, etc. We then have the equivalent co2 emission values for each energy generator and can calculate our coefficient. Here's how we calculate it:

$$\sum_{i=1}^{n} \frac{c_i * v_i}{V_{\text{total}}}$$

n is the number of energy generator, c_i is the corresponding coefficient in our table and v_i is the current value of its use.
However, some of the orders that our code will evaluate are likely to last several hours/days/weeks. It is therefore not feasible to have a single measurement value.
So, we use the code launch date as a reference. If this exceeds one day, we take the last day's data, average it out and change our reference date by one day. We then continue the measurement until the end of the code, using the same method. Finally, we recover the missing data for the last day, counting only the values after the reference time.
With this method, we are therefore able to obtain an average measurement of our coefficient over the entire duration of the code.
It should be noted that we can't retrieve data for the current hour, but only 2 hours earlier, as information closer in time is not yet available.

3.2.2 The GreenAlgorithm function

In the end, we called our function *GreenAlgorithm_adapted*, because it's based on GreenAlgorithm's calculation data, and we were inspired by their code to make ours work. However, it is now more precise, automatic, and adapted to our GPUs, CPUs, and therefore to our servers. Energy calculations are then divided into three parts: CPU energy, GPU energy and RAM energy. Here are the formulas:

*powerNeeded_CPU = PUE_used * numberCPUs * CPUpower * usageCPU*
*powerNeeded_GPU = PUE_used * (GPU_power-GPU_idle)*
*powerNeeded_memory = PUE_used * (memory * power_memory)*

- *usageCPU* is the utilization coefficient of our CPU.
- *CPUpower* is the TDP value for our current CPU.
- *usageCPU* is the average usage factor of our CPU.
- *GPU_power* is the power measured by *pynvml* of our GPU during the experiment
- *GPU_idle* is the power measured by *pynvml* before the launching of the subprocess.

- *memory* is the average RAM space used during code execution, and *power_memory* is our coefficient : 0,3725W/GB.
- *PUE_used* is set at 1.67 as a reference value, but it can be specified.

3.2.3 Specifications summary table from [1]

Here's a table comparing the features of our module with those of others. Values for other modules are from [1]:

	GA adapted	Green-Algorithms	CodeCarbon	Eco2AI	CarbonTracker
General Information					
-Type of tool	Embedded package	Online calculator and Server-side tool	Embedded package	Embedded package	Embedded package
-Process/machine estimation	process	process	both	both	machine
-Measurement frequency (sec)	10	-	15	10	10
Energy Consumption CPU					
-Use Model of CPU	yes	yes	yes (if no tracking tool)	yes	no
-Use RAPL files or Power Gadget	no	no	yes	no	yes (RAPL files)
-Default TDP (Watt)	No	12 (normalized by core)	85	100	-
-Usage Factor considered	yes	yes	50% (if default TDP used)	yes	no
-Tool for usage factor	psutil	-	-	psutil	-
Energy Consumption GPU					
-Use Model of GPU	yes	yes	yes	yes	no
-Default TDP	no	200	no	no	no
-Tool to get power	pynvml		pynvml	pynvml	pynvml
-Usage Factor considered	yes	yes	no	no	no
-Tool for usage factor	pynvml	-	-	-	-
-Only Nvidia GPUs	yes	no	yes	yes	yes
Energy Consumption Memory					
-Measured	yes	yes	yes	yes	yes
-Source of information	psutil		system	system	RAPL files
-Usage Factor considered	yes	no	yes (if tracking mode)	yes	no
-Tool for usage factor	psutil	-	psutil	psutil	-
-Reference value used	0.3725 W/GB	0.3725 W/GB	0.375 W/GB	0.375 W/GB	-
Emission intensity					
-Default E.I value	yes	yes	yes	yes	yes
-Real time	Yes (France only)	no	no	no	yes (UK and Denmark only)
PUE					
-PUE considered	yes	yes	yes (just cloud)	yes	yes
-PUE configurable	yes	yes	no	yes	no
-Default PUE value	1.67	1.67	1	1	1.58
Compatibility					
-Non-Intel CPUs	yes	yes	yes	yes	no
-Non-Nvidia GPUs	no	yes	no	no	no

Figure 5: Specification summary table

4 Module results

4.1 Measurements characteristics

To test and compare the methods, we carried out measurements on different codes and different servers. Here are the characteristics of the two different servers:

	Volta (Mesocentre) (1)	Gemini-1 (Grid5000) (2)
Operating System	Linux	Linux
CPU		
-Quantity (physical cores)	4	20
-Model	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz	Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz *** 2**
-TDP	10	6.75
GPU		
-Quantity	1	1
-Model	Tesla V100-SXM2-32GB	Tesla V100-SXM2-32GB
-TDP	250	250
RAM		
-Quantity (Go)	40	504
Wattmeter		
-Available	no	yes
-Sampling Frequency		1 Hz

Figure 6: Server characteristics

We required the use of a GPU for each of our measurements, even when our code didn't use one, to highlight some of the limitations of other modules.

In the following, the mesocentre will be number 1, and Gemini-1 will be number 2. As you can see, Grid5000 allows us to use a Wattmeter. This enabled us to measure the real energy used by our algorithms, by getting the difference in power between using the algorithm and idle, multiplied by the runtime.

Here are the python algorithms where the experiment was carried out, and the execution time related to these algorithms and the server where they were running:

	Execution time (min)		CPU utilization	GPU utilization
	Mesocentre	Grid5000		
Tumor Detection (TuD)	27.46	15.03	yes	no
Image Recognition (ImR)	2.24	1.75	yes	no
Pytorch Image Recognition (PyImR)	1.72	1.44	yes	yes
MRI segmentation (MRISeg)	25.83	24.6	yes	yes

Figure 7: Execution time

"Tumor Detection" is an algorithm for detecting brain tumors using *sklearn*. It is available at [7].
"Image Recognition" use *tensorflow* for recognizing different images, it is accessible at [8].
"Pytorch Image Recognition" has the same purpose as the previous one, but uses *pytorch* for this, as well as the GPU. It is available at [9].
"MRI segmentation" segmentation from MRI data. For this, it uses *torch* and the GPU [10].

To simplify the next table, we'll rename the algorithms by acronyms, and the server used by a number. You can find the correspondence in the previous tables.

4.2 Measurements result

The measurements were taken on the servers presented above. The passive power consumption of the GPU and nodes has been subtracted from the wattmeter measurements, so as to evaluate our module on its operating target, i.e. power consumption linked only to the process.
In addition, the PUE has been set to 1 for all modules, since the wattmeter cannot take it into account. Here are the measurement results, in Wh:

Algorithms:	TuD1	TuD2	ImR1	ImR2	PyImR1	PyImR2	MRISeg1	MRISeg2
GreenAlgorithm	2.30	5.97	0.10	0.35	0.91	1.30	128.20	119.12
GA adapted	**2.39**	**5.26**	**0.11**	**0.35**	**0.56**	**0.6827**	**47.83**	**48.60**
CodeCarbon	72.78	22.75	4.47	2.42	3.14	2.6	117.61	82.41
Eco2ai	19.69	11.06	0.97	1.21	1.7	1.47	67.53	65.68
CarbonTracker	54.72	10,82	5.94	1,18	4.80	1.38	114.27	64,08
Wattmeter		6.76		0.43		0.62		72.57

Figure 8: Measurements

The Grid5000 measurements give us a good idea of the difference between the values measured by the modules and the wattmeter. We used the "wattmetre_power_watt" wattmeter, to get a measurement specific to our node and our use.

We decided to show the deviation from the actual data obtained by the wattmeter in percent. The target would therefore be 0, as this would mean a result equal to that of the wattmeter. Here's the graph:

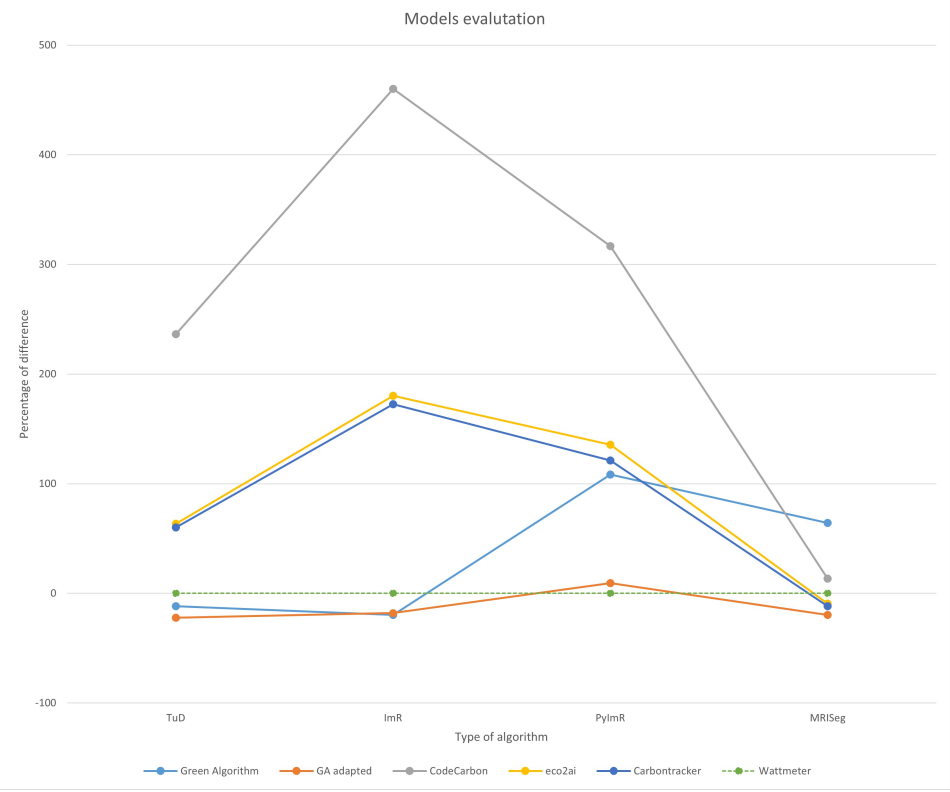

Figure 9: Models evaluation

NB: This figure has been drawn with lines for greater legibility, but it should be noted that each point is independent.

As we don't have sudo access, the measurement of energy spent per CPU and RAM has not been taken into account for *carbontracker*. And *codecarbon* had to make the calculation using the usage factor.
Carbontracker would therefore probably have greatly overestimated the real values, and the first two measurements show a real inconsistency, since our code didn't use a GPU, but *carbontracker* does measure values. We can conclude that it also measures passive GPU use, even when it's not related to our own use, because *42*[2] *watt * time* is approximatly equal to the value of *Carbontracker* in the first two measurements.

The second thing to note is the significant overestimation of GreenAlgorithm for the two last measures. In the last experiment, the measured usage factor was 99%.
This confirms the hypothesis of inconsistency between the GPU's usage factor and the proportion of energy expended relative to its limit. We can conclude that the use of usage factor for GPU power consumption measurements is imprecise and should be take with caution.

Finally, we can see that our module is quite accurate, with a maximum deviation of 22.22%.
As expected, the first two measurements slightly underestimate consumption. This is mainly due to hyperthreading, and is in line with the results presented in part 2.4

Now, it seems interesting to propose the same graph, but with wattmeter values that take into account the passive consumption of the GPU and node on activation. It should be noted that our module is not intended to measure these consumptions, so here's the measures:

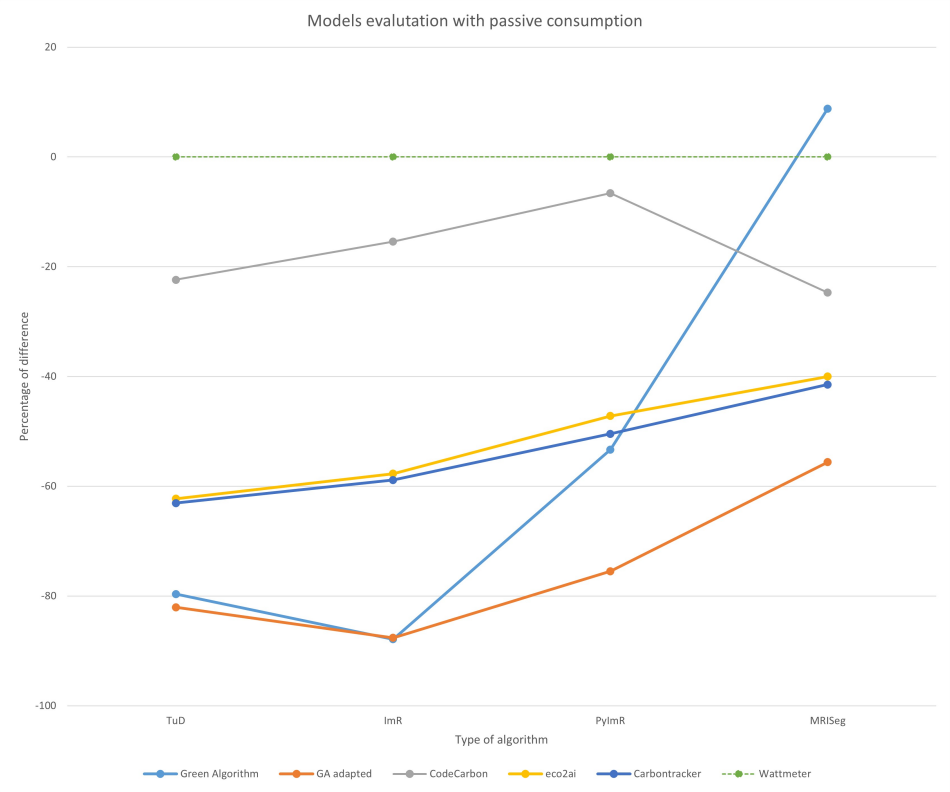

Figure 10: Models evaluation with passive consumption

We can therefore see that *codecarbon* is the most accurate in this case. The fact that it underestimates probably stems from the fact that it doesn't measure passive node consumption. However, it's still

[2]42 watts is the passive power consumption measured by the GPU on grid5000

the best option if you want to measure your own consumption as well as passive consumption.

CarbonTracker and *eco2ai* seem to have very similar measurements. However, this is not confirmed in the measurements on the mesocenter and on the following section with the stress and sleep tests. There is therefore no real concordance.

4.3 Stress and Sleep test

Since our module is not limited to deep learning algorithms, we decided to make some "reference" measurements, using the sleep and stress commands. *sleep* allows us to carry out no commands for a defined period, thus enabling us to take measurements with no load, while *stress* uses the maximum CPU resources available. These two measurements were taken over 5 minutes, in the same server as before, and on my own computer (P). Here is the result:

Algorithms:	**Sleep1**	**Stress1**	**Sleep2**	**Stress2**	**Sleep.P**	**Stress.P**
GreenAlgorithm	0,0338	0.414	0.043	2.80	0.0564	3.61
GA adapted	0,0351	0.3805	0.045	2.73	0.0587	3.76
CodeCarbon	10.8	13.92	7.29	7.27	2.51	2.57
Eco2ai	3.69	3.69	3.69	3.69	0.63	0.6762
Carbontracker	11.34	13.63	3.61	5.70	0.95	1.03
Wattmeter			0	2.42		

Figure 11: Stress and sleep execution

When running sleep on grid5000, no noticeable difference was visible on the wattmeter compared to before running the command, so we decided to set the value to 0.

We can notice that *codecarbon*, *eco2ai* and *carbontracker* greatly over-interpret the values during sleep. In fact, their values are almost identical between *sleep* and *stress*. And this is true on both servers, and even on the last measurements on my personal computer.

The conclusion of these measurements is that these modules are not designed to run on Linux command, as they are probably biased by the full use of the server itself, and they are designed for linear consumption applications such as deep learning algorithms. It is therefore worth keeping them as a reference for deep/machine learning applications, but our code remains the most accurate on these measurements.

eco2ai was the most precise during the first experiments, and that's why we chose it as a reference value for the "total.csv" file if our algorithm doesn't have the necessary data to run.

5 Conclusion

5.1 About out module

We're satisfied with the result we were able to achieve given the time we had available. In fact, the latest measurements on the *sleep* commands also give us an idea of the consumption of our module, as it would be problematic if it also significantly increased the carbon impact. We noticed during the *sleep* measurements that the CPU usage factor was negligible. Moreover, measuring in threads doesn't slow down the execution time proportionally, but in a fixed way, by simply adding the time taken to launch the start() command, and that of the stop() command.
The module is now operational, and we've made sure that it's easy to set up by using this document, even for people who aren't specialists in the field.
We're well aware of the limits of our proposal, and the limitations it implies. The interface is not yet easy to use, and it only works with certain types of GPU, while requiring manual input of CPU information. Moreover taking into account only the user's utilization considerably reduces its real impact. Indeed, the server's passive consumption has a considerable impact, which would have to be divided by all users in proportion to their usage.
In addition, it should be noted that the consumption of data production and storage is often underestimated, even though it represents a real part of AI consumption.

5.2 About the impact of digital technology and how to measure it

We hope that this will enable more people to consider the environmental impact of our digital practices, and that it will give everyone an idea of their own carbon footprint.
We also hope that our thoughts on how to measure the impact of a code on a server will be an interesting contribution to future modules aimed at improving accuracy and user interface.
In conclusion, within the time constraints we had, we remain quite happy with the results we have obtained, and sincerely hope that they will enable research on this subject to evolve more rapidly.

Acknowledgments

This work has been funded by Amidex Projet for interdisciplinarity 'UNITAE: what neuroscience in the Anthropocene era ?' `https://zenodo.org/records/10122489`. We thank Daniele Schön and INS for managing administrative aspects of the internship. We thank Julian Carrey, INSA Toulouse, Atecopol Toulouse, for having diffused the internship topic.

Author contributions Conceptualization, NL, JL; methodology, NL; investigation, NL; writing, NL; visualization, NL; supervision, JL; project administration, NL, JL; funding acquisition, JL.

6 References

[1] L. B. Heguerte, "How to estimate carbon footprint when training deep learning models? A guide and review," arXiv.org, Jun. 14, 2023. https://arxiv.org/abs/2306.08323

[2] "CodeCarbon.io." https://codecarbon.io/

[3] S. Budennyy, "Eco2AI: carbon emissions tracking of machine learning models as the first step towards sustainable AI," arXiv.org, Jul. 31, 2022. https://arxiv.org/abs/2208.00406

[4] "Global Monitoring Laboratory - Carbon Cycle Greenhouse Gases." https://gml.noaa.gov/ccgg/carbontracker/

[5] F. Lastname, Green Algorithms. https://www.green-algorithms.org/

[6] "@ElectricityMaps | Live 24/7 CO2 emissions of electricity consumption." https://app.electricitymaps.com/map

[7] MohamedAliHabib, "Brain-Tumor-Detection/Brain Tumor Detection.ipynb at master · MohamedAliHabib/Brain-Tumor-Detection," GitHub. https://github.com/MohamedAliHabib/Brain-Tumor-Detection/blob/master/Brain%20Tumor%20Detection.ipynb

[8] "Google Colaboratory." https://colab.research.google.com/github/trekhleb/machine-learning-experiments/blob/master/experiments/digits_recognition_cnn/digits_recognition_cnn.ipynb#scrollTo=Bt8lDaSfsC93

[9] B. Neo, "Building an image classification model from scratch using PyTorch," Medium, Feb. 07, 2022. [Online]. Available: https://medium.com/bitgrit-data-science-publication/building-an-image-classification-model-with-pytorch-from-scratch-f10452073212

[10] Daviddmc, "GitHub - daviddmc/NeSVoR: NeSVoR is a package for GPU-accelerated slice-to-volume reconstruction.," GitHub. https://github.com/daviddmc/NeSVoR

[11] "Predicting the Energy-Consumption of MPI applications at scale using only a single node," IEEE Conference Publication | IEEE Xplore, Sep. 01, 2017. https://ieeexplore.ieee.org/document/8048921

[12] "Predicting the Energy-Consumption of MPI applications at scale using only a single node," IEEE Conference Publication | IEEE Xplore, Sep. 01, 2017. https://ieeexplore.ieee.org/document/8048921

A Appendix : Application

A.1 Installation and preparation

If you're working on a mesocentre server or INT server, you don't need to work on this first step, as these files have already been installed for you.
If this is not the case, the first step is to download the files available on github, under this link: `https://github.com/nathanlemmers/EnergyTracker`.
Once these files have been installed, it's time to download all the libraries needed to run the code. The "setup_module.sh" file is available for this purpose. We advise you to use the command "source setup_module.sh" to be sure of their initialization. Libraries will not be reinstalled if they are already on your server/computer. By adding argument 1, python module for the mesocentre will be automatically uploaded.
Next, if you don't want to have to fill in certain values in the GreenAlgorithm site by hand to get a result, you'll need to add certain data to the "data_GA.csv" file.
This file contains all the reference values with which our most precise function works. To do this, you need to add the data from your processor. Here's how it works:

Launch a Python terminal, and run these commands:

```
import cpuinfo
cpu_name = cpuinfo.get_cpu_info()['brand_raw']
print(cpu_name)
```

Write down the exact name of your CPU in this form and add it to the data column of the "data_GA.csv" file. Next, find the associated TDP value on the Internet. For example, if you have an Intel CPU, you can check on : `https://www.intel.fr/content/www/fr/fr/products/overview.html`.
Then divide the TCP value by the number of **physical** cores. And put this value in the « value » column.
After that, your data is now up to date, and an adapted version of *GreenAlgorithm* is available.

To make the code as efficient as possible, you'll need an API key to access live data about the gco2/kWH equivalent. If you decide not to install it, the default average value will be generated, representing the average value in France. To generate an API key, you need to create an account and make a request at:
https://opendata.reseaux-energies.fr/
Then, you have to run this command on your terminal:

```
export RESEAUX_ENERGIES_TOKEN= your_key
```

Once you've done all these installations, the program is ready to use.

A.2 Launching an operation

To measure the carbon impact of your file, run:

```
./Monitor/monitor.py "your_linux_command"
```

The co2 measurement will be taken. Make sure that the setup file has been executed first.
If you want to add the value of your PUE, you can add pue=value as another argument after your "linux command". The default value is set to 1.67, so you must add pue=1 if you want to run the code on your own computer.
If your code has a short duration, we also advise you to change the frequency at which measurements are taken. The default value is 10 sec, but you can change it by adding the argument freq=*time* in seconds.
Here is an example on a command on my laptop:

```
./Monitor/monitor.py "sleep␣15s" pue=1 freq=5
```

A.3 Result file

After running the previous command, a new folder named *co2* should be created. In this folder, some important information.
Firstly, you should have a new folder for each command you've run. This is recognizable by its name, which should be the date of your code launch. Inside, you have 3 files, representing the 3 measurements that have been made independently.
If you want to see and compare the effectiveness of the different methods, go to the last section.
To quickly summarize, the most accurate result is that of the "GreenAlgorithm.txt" file. However, it's important to note that some functions require CUDA on your GPU. The measurements will be distorted, and there's no point in using the results from the "GreenAlgorithm.txt" file if your code is using a GPU without CUDA.
If you have filled in the "data_GA.csv" file correctly, then the result should be the most accurate. However, if this is not the case, you may need to go to http://calculator.green-algorithms.org/
to enter the information written in this file by hand. In this case, the *eco2ai* values will be used as a reference for calculating your total emissions.
These are found in the file "total.csv" in the co2 folder and contains the total of your co2 emissions, accumulating all the results of your measurements.

NB: If you want to free up space but keep your total consumption, simply never delete the "total.csv" file. All other files and folders can be deleted.