Project 4: Numc

Deadline: Friday, August 6, 11:59:59 PM PT

Overview

This project is designed to be both a C project as well as a performance project. In this project you will be implementing a slower version of numpy, a very useful Python library for performing mathematical and logical operations on arrays and matrices. Your version of numpy, `numc` (how exciting!), is most likely to be slower than numpy, but much faster than the naive implementations of matrix operations. The steps of this project are as follows:

1. You will first complete a naive solution for some matrix functions in C
2. You will gain a deeper understanding of the Python-C interface by overloading some operators and defining some instance methods for `numc.Matrix` objects
3. Finally, you will speed up your naive solution, thus making `numc.Matrix` operations faster.

Do not expect your final completed `numc` module to be as good as numpy, but you should expect a very large speedup compared to the naive solution, especially for matrix multiplication and exponentiation!

Tips and Guidelines

- Please start early! Because there are many more 61C students than Hive machines, you will likely share resources with your classmates. This might affect the measurement of your code's speedup. We encourage you to use Hivemind to help balance the load amongst the hive machines.

    - You will have 6 tokens every 6 hours for the Gradescope assignment. So again, please start early!
- For this project, **we strongly suggest working on Hive machines under your cs61c account**. We have set up a few environment settings that only work if you use your cs61c account. If you run into issues related to not working on the Hive machines, we will **NOT** be able to help you.
- If you would like to run the reference solution to compare your solution to, you can import the `dumbpy` library on hive as we have already installed it there for you!
- We will not be directly testing your C code. All tests will be in Python!
- You may change the skeleton code in `src/numc.c`, especially if you are not using a row-major setup for your matrices.
- You may **NOT** add/remove any additional imports.
- You may change the function signatures in the following files:
    - `src/matrix.h` and `src/matrix.c`
- You may not change the function signatures in the following files:
    - `numc.h` and `src/numc.c`
- You will get negative points up to twice the number of points it is worth if you fail to complete task 4, so you should do it as much as you can even if you do not complete the entire project!

Getting Started

> **Warning**: this assignment allows groups of 2 students! If you have a group, one member should create the repo and invite the other group member to that repo. **The other group member should NOT create their own repo.** If you haven't decided on a group yet, don't create the repo yet. You will not be able to change groups after creating a repo!

Visit Galloc. Log in and start the Project 4 assignment. A GitHub repo will be created for your team; this will be your repo for any Project 4 work you do. Then, clone your repository locally and add the starter remote:

```
$ git clone YOUR_REPO_NAME
$ cd YOUR_REPO_NAME
$ git remote add starter https://github.com/61c-teach/su21-proj4-
$ git pull starter main
```

If we publish changes to the starter code, retrieve them using `git pull starter main`.

To be able install the modules that you will complete in this project, you must create a virtual environment with by running

```
$ python3.6 -m venv .venv
```


Note that you **MUST** use python 3.6 as our reference module `dumbpy` only supports this specific version of python. Finally, run the following command to activate the virtual environment (a tool used to create isolated Python environments such that our project can have its own dependencies, regardless of what dependencies every other project has).


```
$ source .venv/bin/activate
```


This will put you in a virtual environment needed for this project. Please remember that if you exit the virtual environment and want to return to work on the project, you must re-run `source .venv/bin/activate`. This also means every time you re-ssh into the hive, you will have to re-run `source .venv/bin/activate`.

Then, run

```
pip3 install -r requirements.txt
```

in the virtual environment. This will install all python packages you need for running your custom python tests.

Finally, if you have to exit out of the virtual environment, you can do so by running:

```
$ deactivate
```

We already have the reference library `dumbpy` installed for you on Hive machines. You can import it with or without the virtual environment while using python3.6, and all object and function names are the same as the `numc` module that you will use (please refer to using the setup file). **You will only be able to access the dumbpy package on hive as we will not be directly releasing it**. You can use it as a reference for both correctness and speed.

Again, for this project, **we strongly suggest working on Hive machines in your cs61c class account**. **You will not be able to import dumbpy if you are using other class accounts**. We will be unable to help you with issues caused by working outside of the Hive.

Task 1: Matrix functions in C

Before contacting the staff about any issues that you encounter, please have a look through the FAQ here.

For this task, you will need to complete all functions in `src/matrix.c` labelled with `/* TODO: YOUR CODE HERE */`. The comments above each function signature in `src/matrix.c` contain instructions on how to implement the functions and the comments next to each variable of the `matrix` struct in `src/matrix.h` contain details about each variable, so read them carefully before you start coding.

The function `allocate_matrix_ref` is called from `src/numc.c`'s `Matrix61c_subscript` function and and is used for getting a row of the `from` matrix (see Info: numc.Matrix indexing for an example). Currently, `Matrix61c_subscript` and `allocate_matrix_ref` assume a row-major setup. If you choose to implement your matrices as column-major, you will have to change the implementation of `Matrix61c_subscript`, and you might also want to change the function signature of `allocate_matrix_ref`.

Again, you may change any function signature in `src/matrix.h` and `src/matrix.c`.

Important notes:

- The `deallocate` function as well as the `ref_cnt` field in the `matrix` struct have caused a lot of confusions in the past semesters. It is important to remember that this `ref_cnt` is **NOT** Python's internal reference count. It is simply a field that will help you implement the `deallocate` function. It does not have to reflect the true reference count if you deem that setting it to other values will simplify your implementation of `deallocate`.
- For the `deallocate` function, since there can be multiple matrices that refer to the same data array in the memory, you must not free the data until you call `deallocate` on the last existing matrix that refers to that data. If you are having some difficulties implementing this, here's a hint: you can keep the matrix struct in the memory even if you have already called `deallocate` on that matrix. **You only need to make sure to that the struct is freed once the last matrix referring to its data is deallocated.**
    - If this explanation does not make sense now, don't worry! It will make more sense after you implement the indexing section of task 2.

Throwing Errors

For the following errors, please have a look at `PyExc_ValueError` and `PyExc_RuntimeError` in `PyErr_SetString` mentioned here to throw errors correctly. Make sure to test these errors and not to confuse the error types being thrown.

Return value	Error (later thrown in Task 2)	When to return value
`-1`	`PyExc_ValueError`	If you are trying to allocate matrices with non-positive dimensions.
`-2`	`PyExc_RuntimeError`	If `allocate_matrix` or `allocate_matrix_ref` fails to allocate space.

Since you will use these return values to later throw these errors in `src/numc.c`, you would have to make sure that a runtime/value error will be

thrown in `src/numc.c` whenever we run out of memory. **This includes throwing an error in Matrix61c_init**.

Testing for Correctness

We've provided some sanity in `tests/mat_test.c`. **These tests make several assumptions:**

- They assume that all result matrices are already pre-allocated with the correct dimensions and that all input dimensions are valid.
- They assume that you have not modified the `matrix` struct in `src/matrix.h`
- All tests except the tests for `get` and `set` assume that your `get` and `set` are correct
- They assume that you have not modified the function signatures. **Note that you may still change the function signatures of `matrix.c` and `matrix.h` if you'd like to, as mentioned here.** These tests are different from the autograder tests because they only test `matrix.c` which is why you need to have the same function signatures here.

Violation of one or more of these assumptions may not cause your tests to fail, but please keep this in mind if your tests are failing and you are violating at least one of these assumptions.

To run the CUnit tests, run

```
$ make test
```

in the root folder of your project. This will create an executable called `test` in the root folder and run it.

By default, CUnit will run these tests in Normal mode. When debugging a specific issue, it may be helpful to switch to Verbose mode, which can be done by commenting and uncommenting the relevant lines in `mat_test.c`:

```
// CU_basic_set_mode(CU_BRM_NORMAL);
CU_basic_set_mode(CU_BRM_VERBOSE);
```

Make sure that one line is uncommented at a time.

Please keep in mind that these tests are not comprehensive, and passing all the sanity tests does not necessarily mean your implementation is correct. This is especially true with the memory functions `allocate_matrix`, `allocate_matrix_ref`, and `deallocate_matrix`. Also keep in mind that the autograder will be using our own set of sanity tests, and will not be running your CUnit tests.

Another thing to note is that the **`Makefile`** is written for compilation on the hive machines. If you wish to run it locally, you will have to modify the **`Makefile`** by replacing the path to your CUnit/Python libraries in your `CUNIT` and `PYTHON` variables. You will also need to make sure that your local computer supports AVX extensions and OpenMP.

Finally, you are welcomed to modify the **`tests/mat_test.c`** file in the `tests` directory to implement your custom test cases. To add your own custom tests, you will need to define your own test function and possibly use any of the `CU_ASSERT_EQUAL`, `CU_ASSERT_NOT_EQUAL`, or `CU_ASSERT_PTR_EQUAL` CUnit test cases to compare any value that you would like (Suggestion: there are more possible CUnit Test Cases linked here. Lastly, you will need to call `CU_add_test` to the main function to run your newly created function! A good place to start is to look at some of the provided tests and use the general approach to your own specific tests.

Using the setup file

The `setup.py` file is used for installing your custom-built modules and has been already provided to you. Have a look at the code and make sure to understand what is being done. You should be able to install `numc` by simply running:

```
$ make
```

This will uninstall your previously installed `numc` module if it existed and reinstall `numc`. We have written `src/numc.c` so that `numc.Matrix` will be initialized and ready to import upon succesful installation of the `numc` module. You should rerun `make` every time you make changes and want them to be reflected in the `numc` module.

You can uninstall your `numc` module by running

```
$ make uninstall
```

You will likely get a lot of warnings about functions being defined but not used, and that's ok! You should ignore these warnings for now, and they will be gone after you finish writing Task 2.

Remember that **you must be in the virtual environment that you set up in order to install the modules**, otherwise you will get a "Read-only file system" error.

Task 2: Writing the Python-C interface

Before contacting the staff about any issues that you encounter, please have a look through the FAQ here.

Now that you have successfully installed your `numc` module, you can import your `numc.Matrix` objects in Python programs! Here are some ready-to-use features already implemented for `numc.Matrix` objects. You might find them helpful when debugging Task 2.

Info: Importing `numc.Matrix`

Here are several ways of importing `numc.Matrix`

```
from numc import Matrix

import numc
numc.Matrix
```

```
import numc as nc
nc.Matrix
```

Info: `numc.Matrix` initialization

The code block below lists all the different ways of creating a `numc.Matrix` object.

```
>>> import numc as nc
CS61C Summer 2021 Project 4: numc imported!
>>> nc.Matrix(3, 3) # This creates a 3 * 3 matrix with entries al
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
>>> nc.Matrix(3, 3, 1) # This creates a 3 * 3 matrix with entries
[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]
>>> nc.Matrix([[1, 2, 3], [4, 5, 6]]) # This creates a 2 * 3 matr
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
>>> nc.Matrix(1, 2, [4, 5]) # This creates a 1 * 2 matrix with er
[[4.0, 5.0]]
```


More specifically:

- `nc.Matrix(rows: int, cols: int)` will create a matrix with `rows` rows and `cols` cols. All entries in this matrix are defaulted to 0.
- `nc.Matrix(rows: int, cols: int, val: int/float)` will create a matrix with `rows` rows and `cols` cols. All entries in this matrix will be initialized to `val`.
- `nc.Matrix(rows: int, cols: int, lst: List[int/float])` will create a matrix with `rows` rows and `cols` cols. `lst` must have length `rows * cols`, and entries of the matrix will be initialized to values of `lst` in a row-major order.
- `nc.Matrix(lst: List[List[int/float]])` will create a matrix with the same shape as the 2D `lst` (i.e. each list in `lst` is a row for this matrix).

Info: `numc.Matrix` indexing

You can index into a matrix and change either the value of one single entry or an entire row. More specifically, `mat[i]` should give you the `i`th row of `mat`. If `mat` has more than 1 column, `mat[i]` should also be of type `numc.Matrix` with (`mat`'s number of columns, 1) as its shape. In other words, `mat[i]` returns a row vector. This is to support 2D indexing of `numc` matrices.

If `mat` only has one column, then `mat[i]` will return a double. `mat[i][j]` should give you the entry at the `i`th row and `j`th column. If you are setting one single entry by indexing, the data type must be `float` or `int`. If you are setting an entire row of a matrix that has more than one column, you must provide a 1D list that has the same length as the number of columns of that matrix. Every element of this list must be either of type `float` or `int`.

Please note that if `mat[i]` has more than 1 entry, it will share data with `mat`, and changing `mat[i]` will result in a change in `mat`.

The example given below assumes the matrices are initialized from the code block above.

```
>>> import numc as nc
CS61C Summer 2020 Project 4: numc imported!
>>> mat = nc.Matrix([[1, 2, 3], [4, 5, 6]]) # This creates a 2 *
>>> mat
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
>>> slice = mat[0]
>>> slice
[[1.0], [2.0], [3.0]]
>>> slice[0]
1.0
>>> slice[1] = 10.0 # Change a value in slice
>>> slice
[[1.0], [10.0], [3.0]]
>>> mat # Mat is changed as well
[[1.0, 10.0, 3.0], [4.0, 5.0, 6.0]]
```


Partial slices, however, are not supported. For example,

```
mat[1:3] # not allowed
mat[0][1:3] # not allowed
```

Info: instance attributes

The matrices and vectors have an attribute shape, which is a tuple of `(rows, cols)`. Example is given below.

```
>>> mat.shape
(3, 3)
```


Info: Python/C API Reference

Here is the link to the full reference manual: https://docs.python.org/3.6/c-api/index.html. If you ever find anything confusing in the skeleton code or are at a lost on how to implement `src/numc.c`, this is a great resource.

Quick Overview of skeleton code

We define the `Matrix61c` struct in **`numc.h`**. It is of type `PyObject` (this means you can always cast `Matrix61c` to `PyObject`, but not vice versa), which according to the official documentation, "contains the information Python needs to treat a pointer to an object as an object". Our `Matrix61c` has the `matrix` struct we defined in **`src/matrix.h`**.

Then we define a struct `PyTypeObject` named `Matrix61cType` to specify the intended behaviors of our Python object `Matrix61c`. This struct will then be initialized to be our `numc.Matrix` objects.

```
static PyTypeObject Matrix61cType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "numc.Matrix",
    .tp_basicsize = sizeof(Matrix61c),
    .tp_dealloc = (destructor)Matrix61c_dealloc,
```

```
    .tp_repr = (reprfunc)Matrix61c_repr,
    .tp_as_number = &Matrix61c_as_number,
    .tp_flags = Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE,
    .tp_doc = "numc.Matrix objects",
    .tp_methods = Matrix61c_methods,
    .tp_members = Matrix61c_members,
    .tp_as_mapping = &Matrix61c_mapping,
    .tp_init = (initproc)Matrix61c_init,
    .tp_new = Matrix61c_new
};
```

For example, `.tp_dealloc` tells Python which function to call to destroy a `numc.Matrix` object when its reference count becomes 0, and `.tp_members` tells Python what instance attributes `numc.Matrix` objects have. You can take a look at the official documentation if you are curious.

Useful functions:

Here is a list of some functions and Python objects from `<Python.h>` that you may find useful. You can also choose any other functions at this link.

- PyObject_TypeCheck
- PyErr_SetString
- Py_BuildValue
- PyTupleObject
- PyLongObject
- PyFloatObject
- PyListObject

Now you are ready to complete **`src/numc.c`**, the Python-C interface! As before, you will need to fill out all functions and variables labeled `/* TODO: YOUR CODE HERE */`. The code for initializing the module `numc` and the object type `numc.Matrix` is already done for you. Although not required, we

encourage you to take a look at the existing code to better understand the interface.

Below are the two main parts for this task.

Note: Don't forget to use the return values from task 1 to throw `PyExc_ValueError` and `PyExc_RuntimeError` in `src/numc.c`! Here is the error table for your convenience:

Return value	Error	When to return value
`-1`	`PyExc_ValueError`	If you are trying to allocate matrices with non-positive dimensions.
`-2`	`PyExc_RuntimeError`	If `allocate_matrix` or `allocate_matrix_ref` fails to allocate space.

Number Methods

For this part, we ask you to overload operators for `numc.Matrix` objects. **For the following operations and functions, please have a look at `PyExc_TypeError`, `PyExc_ValueError`, and `PyExc_IndexError` in `PyErr_SetString` to throw errors correctly. Make sure to test these errors and not to confuse the error types being thrown.** We have made subtraction and negation optional to reduce the amount of redundant work for the project. Feel free to implement these functions if you would like. We have provided autograder tests for you to test their functionality. Extra points will NOT be awarded for implementing the optional functions. Here are the expected behaviors of overloaded operators:

Operator	Function	Description	Throwing Errors

Operator	Function	Description	Throwing Errors
+	`Matrix61c_add`	Element-wise sum of `a` and `b`. Returns a `numc.Matrix` object.	• `TypeError` if either `a` or `b` is not of type `numc.Matrix`. • `ValueError` if `a` and `b` do not have the same dimensions.
- (subtraction)	`Matrix61c_sub`	**(Optional)** Element-wise subtraction of `a` and `b`. Returns a `numc.Matrix` object. It is optional and will not be graded since its implementation is very similar to `Matrix61c_add`.	• `TypeError` if either `a` or `b` is not of type `numc.Matrix`. • `ValueError` if `a` and `b` do not have the same dimensions.

Operator	Function	Description	Throwing Errors
*	`Matrix61c_multiply`	Matrix multiplication of `a` and `b`. Returns a `numc.Matrix` object. Remember that this is a matrix multiplication, not an element-wise multiplication.	• `TypeError` if either `a` or `b` is not of type `numc.Matrix`. • `ValueError` if `a`'s number of columns is not equal to `b`'s number of rows.
- (negation)	`Matrix61c_neg`	**(Optional)** Element-wise negation of `a`. Returns a `numc.Matrix` object. It is optional since the implementation is simple when using other helper operators within the current list.	`TypeError` if `a` is not of type `numc.Matrix`.
abs()	`Matrix61c_abs`	Element-wise absolute value of `a`. Returns a `numc.Matrix` object.	`TypeError` if `a` is not of type `numc.Matrix`.

Operator	Function	Description	Throwing Errors
**	`Matrix61c_pow`	Raise `a` to the `pow`th power. `a` to the 0th power is the identity matrix (1 on the top left to bottom right diagonal and 0 everywhere else). Returns a `numc.Matrix` object. This operator is defined in terms of matrix multiplication, not element-wise multiplication.	• `TypeError` if `a` is not of type `numc.Matrix` or `pow` is not an integer. • `ValueError` if `a` is not a square matrix or if `pow` is negative.

Please note that for all these operations above, you never directly modify the matrix that you pass in. You always make a new `numc.Matrix` object to hold your result, so **make sure you set the `shape` attribute of the new `numc.Matrix`.** You can use `Matrix61c_new` to create new `numc.Matrix` objects. Take a look at the implementation of `Matrix61c_subscript` for an example.

For all the functions above, throw a runtime error using `PyExc_RuntimeError` (similarly to `PyExc_ValueError`, `PyExc_TypeError`, and `PyExc_IndexError` as mentioned above) if any error occurs (such as matrix allocation failure) and causes the operation to fail. Moreover, for any operations that involve two instances of `numc.Matrix`, you will have to make sure that both `a` and `b` are

indeed of type `numc.Matrix` as we do not support operations between `numc.Matrix` and other data/object types. **Please read the comments in `src/numc.c` carefully**.

After you implement all the functions above, you will need to fill out the struct `Matrix61c_as_number` in `src/numc.c`, which is used to define the object type `numc.Matrix`. **Remember to cast your functions to the correct types when assigning them to `Matrix61c_as_number`'s fields!** Here is the link to the official documentation of a `PyNumberMethods` struct: https://docs.python.org/3/c-api/typeobj.html#c.PyNumberMethods

Instance Methods

You will implement two instance methods for `numc.Matrix`:

Method	C Function	Description	Throwing Errors
`set`	`Matrix61c_set_value`	Set `self`'s entry at the `i`th row and `j`th column to `val`.	• `TypeError` if the number of arguments parsed from args is not 3, if `i` and `j` are not integers, or if `val` is not a `float` or `int`. • `IndexError` if `i` or `j` or both are out of range.

Method	C Function	Description	Throwing Errors
`get`	`Matrix61c_get_value`	Returns the entry at the `i`th row and `j`th column. Return value is a Python `float`.	• `TypeError` if the number of arguments parsed from args is not 2 or if either `i` or `j` is not an integer. • `IndexError` if `i` or `j` or both are out of range.

After you implement all the functions above, you will need to fill out the array of `PyMethodDef` structs `Matrix61c_methods` in `src/numc.c`, which is used to define the object type `numc.Matrix`.

This link tells you what goes into a `PyMethodDef` struct:
https://docs.python.org/3/c-api/structures.html

Indexing

As mentioned in task 1, if you are storing your matrix data in a non-row-major order, you might want to change your `Matrix61c_subscript`.

Regardless of how you are storing you matrices, now is a good time to check if your `allocate_matrix_ref` in **`src/matrix.c`** is working as intended. A correct implementation of `allocate_matrix_ref` and `Matrix61c_subscript` should result in behaviors specified in the indexing info section above. **More importantly, please take some time to make sure that you don't have memory leaks!** In other words, you need to make sure that when all references to a matrix is gone, that matrix's data need to be free'd immediately. However, when the parent itself is gone but there are still existing slices of this matrix, you can delay the freeing of the parent's data until all those slices are also gone. Here's an example to what is meant by the above.

```
import numc as nc
a = nc.Matrix(2, 2)
b = a[0] # b is referencing a's data
del a # a is gone, but you don't have to free a's data right now
del b # b is gone, now you need to free a's data
```

How to Debug

To debug the Python-C interface, we suggest that you write your test files in Python, and use gdb or both gdb and pdb to debug.

Using only gdb

You don't have to use pdb if you do not wish to set breakpoints in your Python test file. Open your terminal and run


```
$ gdb python3
```

Then you can set breakpoints in your C files using the normal gdb commands. gdb will warn you with


```
No source file named {your c file}
Make breakpoint pending on future shared library load? (y or [n])
```

Press 'y' (without the quotes) to proceed.

After that, you can run `run {your python test file name}.py` in gdb, and gdb will break at the breakpoints that you just set.

Using both gdb and pdb

You will have to use pdb if you wish to set breakpoints in your Python file. Here's how it works. Start gdb by running

```
$ gdb python3
```

and set your breakpoints in C (see previous section). Then you will need to run in gdb

```
run -m pdb {your python file}.py
```

After this step, you can set breakpoints in your Python file using gdb syntax (for example, `b test.py:5`). With this approach, your debugger will switch between pdb and gdb depending on whether you are stepping through a Python file or a C file. If you are seeing values being optimized out on gdb, try changing `O3` to `O0` in `setup.py`, then reinstalling your module.

Task 3: Speeding up matrix operations

Before contacting the staff about any issues that you encounter, please have a look through the FAQ here.

Now that you have completed the two steps above and successfully installed your naive version of `numc`, it's time to speed up your matrix functions in `src/matrix.c`! Below we outline some steps for boosting performance.

Please note that cache blocking, a technique to rearrange data access to pull subsets (blocks) of data into cache and to operate on this block to avoid having to repeatedly fetch data from main memory, is not necessary to meet the benchmark speedups.

Step 1: Algorithmic and Other Optimizations

You should first try to speed up the computation by trying to apply conventional code optimizations (i.e. without using SSE or OpenMP). While we won't tell you the exact steps, here are some hints that should help you get started:

1. Function calls are expensive since they involve setting up a stack frame and jumping to a different part of code. See if there are any functions that are frequently called that don't necessarily need to be.

2. Are there any places where you can apply algorithmic optimizations? A good place to begin is by viewing the various exponentiation techniques described here (and few of the following pages as well).
3. Is there any unnecessary computation being done?
4. Are there any places where you could do manual loop unrolling?

Note that the above hints relate to general optimization practices. You do not necessarily need to do all of these to achieve a good speedup.

Once you have improved performance using these optimizations, you can start applying vectorization and parallelization to make the program even faster. Note that you have considerable freedom to apply any of these optimizations, and there is more than one correct solution. Try to experiment with different approaches and see which one gives you the best performance.

Step 2: SIMD Instructions

From lectures, you learned how to apply SIMD instructions to improve performance. The processors in the hive machines support the Intel AVX extensions, which allow you to do SIMD operations on 256 bit values (not just 128 bit, as we have seen in the lab). You should use these extensions to perform four operations in parallel (since all floating point numbers are doubles, which are 64 bit in size). If you are unfamiliar with SIMD instructions, lab 9 can be a good warmup.

As a reminder, you can use the Intel Intrinsics Guide as a reference to look up the relevant instructions. You will have to use the `__m256d` type to hold 4 doubles in a YMM register, and then use the `_mm256_*` intrinsics to operate on them.

Here is a list of AVX instructions that you may find helpful, although you are also allowed to use other AVX instructions not on the list.

```
void _mm256_storeu_pd (double * mem_addr, __m256d a)
__m256d _mm256_set1_pd (double a)
__m256d _mm256_set_pd (double e3, double e2, double e1, double e0
```

```
__m256d _mm256_loadu_pd (double const * mem_addr)
__m256d _mm256_add_pd (__m256d a, __m256d b)
__m256d _mm256_sub_pd (__m256d a, __m256d b)
__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c)
__m256d _mm256_mul_pd (__m256d a, __m256d b)
__m256d _mm256_cmp_pd (__m256d a, __m256d b, const int imm8)
__m256d _mm256_and_pd (__m256d a, __m256d b)
__m256d _mm256_max_pd (__m256d a, __m256d b)
```

Step 3: OpenMP

Finally you should use OpenMP to parallelize computation. Note that you will need to make sure that none of the different threads overwrites each others' data. Just adding a `#pragma omp parallel for` may cause errors. Here are a few useful links to use as a starting point: Using OpenMP with C and OpenMP Functions.

Note that the Hive machines have 4 cores with two hyperthreads each. This means that you should expect a speed-up of 4-8x (note that hyperthreads mean that two different threads execute on the same physical core at the same time; they will therefore compete for processor resources and as a result, you will not get the same performance as if you were running on two completely separate cores).

Loop Unrolling

Please note that depending on your implementation, loop unrolling might not increase your performance. This is due to the fact that within large code with large inputs, loop unrolling does not work if the compiler can't predict the exact amount of iterations of the loop at compile time. However, manual unrolling may indeed increase your speedup values. This might not work for some students due to their implementation, however, it may work for some. Due to this, the staff solutions used to determine the benchmark speedups did not include loop unrolling. You may have a read here and test it out by viewing the assembly dump by running gcc with the `-S` flag on your `src/matrix.c` file.

Task 4: Tell us what you did!

Write up what you did in your README.md! We have provided a template for you to use where you should discuss what you did as a whole, the different python functions you implemented, what performance improvements you had, what were you surprised about, etc. More specifically, we want you to document your `numc` module. **We expect a minimum of 1000 characters (1000 characters from each partner if working in pairs).** Again, failure to complete this task may result in negative points, so make sure you do it or you will lose points!

Testing

We will not be grading your tests but we will NOT help you debug unless you have written a test which shows how your code is failing. This means just using the autograder to figure out your issues will not be acceptable for office hours.

We use **`tests/unittests`** as the framework for testing and have provided a `tests/unittests` folder that contains this framework for testing your python module. You should be familiar with `unittest` by now as you have had experience with it in project 2. **Here is the official documentation for the standard Python unittest library.** `tests/unittests/unittests.py` contains all the skeleton code for both correctness and performance tests, and `tests/unittests/utils.py` has all the functions that you might need in order to write tests. We have provided some sample tests for you, but it is up to you to design and code up the tests. We will not be grading your tests.

As mentioned in Tips and Guidelines and Getting Started, we have installed the naive solution which we will be comparing against on hive! The python package is called `dumbpy` and you can import it like any other python library (so long as you are on hive)! Please note we will not be distributing this binary which means you must work on hive if you want to test with it. You should use this and the **`time`** package to determine how much you sped up your code.

Functions in `tests/unittests/utils.py`

- **`dp_nc_matrix(*args, **kwargs)`**
  - This function will return a `dumbpy` matrix and a `numc` matrix that are identical. `*args` and `**kwargs` will be used as the arguments to instantiate both matrices in the exact same format as you would instantiate a `numc.Matrix`. We provide some examples below.

```
>>> dp_mat, nc_mat = dp_nc_matrix(2, 2, 0)
>>> dp_mat
[[0.0, 0.0], [0.0, 0.0]]
>>> nc_mat
[[0.0, 0.0], [0.0, 0.0]]
>>> dp_mat, nc_mat = dp_nc_matrix([[1, 2, 3]])
>>> dp_mat
[[1.0, 2.0, 3.0]]
>>> nc_mat
[[1.0, 2.0, 3.0]]
```


- **`rand_dp_nc_matrix(rows, cols, low=0, high=1, seed=0)`**
  - This function will instantiate a random `dumbpy` matrix and a random `numc` matrix with `seed` where each element is in the range [low, high). The two matrices are identical with `rows` rows and `cols`. `seed` is defaulted to 0.


```
>>>> dp_mat, nc_mat = rand_dp_nc_matrix(2, 2, seed=5)
>>> dp_mat
[[0.27474559623503386, 0.046467764790387715], [0.99275522
>>> nc_mat
[[0.27474559623503386, 0.046467764790387715], [0.99275522
```

- **`cmp_dp_nc_matrix(dp_mat: dp.Matrix, nc_mat: nc.Matrix)`**
  - This function will return True if `dp_mat` has the same size of `nc_mat` and all corresponding elements are equal (within a margin of error).
- **`compute(dp_mat_lst: List[Union[dp.Matrix, int]], nc_mat_lst: List[Union[nc.Matrix, int]], op: str)`**

- This function takes in a list of dumbpy matrices and a list of numc matrices, then applies the specified operation `op` on these matrices. Note that `op` is a string. `"add"`, `"sub"`, `"mul"`, `"neg"`, `"abs"`, and `"pow"` correspond to the operations `+`, `-`(subtraction), `*`, `-`(negation), and `abs`, respectively. This function will return whether the computed `dumbpy` matrix is equal to the computed `numc` matrix, as well as the speedup for this specific computation.
  - For unary operations like `"neg"` and `"abs"`, each matrix list must only contain 1 matrix. For the binary operations, they must contain more than 1 matrix.
  - Note that to compute `"pow"`, `dp_mat_lst` and `nc_mat_lst` should look like something like `[Matrix(2, 2), 1]`.
  - You can also do chained operations like `compute([a, b, c], ["add", "mul"])` where `a, b, c, d, e, f` are matrices, and the function will compute `a + b * c` and `d * e * f`.

More about `unittest`

- Running

```
$ python -m unittest unittests.py -v
```

  will run all tests in `unittests.py`.
- If you only want to run tests of one particular class, run

```
$ python -m unittest unittests.{classname} -v
```

  For example, running

```
$ python -m unittest unittests.TestAdd -v
```

  will only run the tests under the `TestAdd` class.
- If you want to run a specific test under a specific class, run

```
$ python -m unittest unittests.{classname}.{testname} -v
```

  For example, running

```
$ python -m unittest unittests.TestAdd.test_small_add -v
```

will only run the `test_small_add` test.

Frequently Asked Questions

Task 1

- **Q 1.1**: Will we be graded on what types of errors we threw? Or do we just have to throw some error?
  A 1.1: Yes we expect you to throw the correct error types. We do not care about the error strings tho.

- **Q 1.2**: I'm getting this when I run `make test`, what is happening?


```
rm -f test
gcc -g -Wall -std=c99 -fopenmp -mavx -mfma -pthread -O3 mat_t
./test
     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/
Makefile:25: recipe for target 'test' failed
make: *** [test] Illegal instruction (core dumped)
```

  A 1.2: This means your code segfaulted before any assert statements was reached and none of the tests passed. Use gdb to locate your bug.

- **Q 1.3**: Do we assume rows and cols start at 1 or 0?
  A 1.3: They are both zero-indexed.

- **Q 1.4**: Do our functions need to support self-referential operations? For example, `mul_matrix(mat, mat, mat)`.
  A 1.4: No.

- **Q 1.5**: Is it reasonable to run valgrind on `./test`?
  A 1.5: You can try, but would not recommend. Setting up valgrind for C-

Python interface can be difficult.

- **Q 1.6**: When we throw an error, is it just a fprintf call to stderr?
  A 1.6: Nope, you should throw errors in a python context. i.e., when your python code calls this underlying C function, it should error. Take a look at the Python/C API's reference manual for exception handling.

- **Q 1.7**: Can we make a slice on slice? for example, `mat1 = [[1,2,3], [4,5,6]]`, `mat2 = mat1[0]`. Can we do `mat3 = mat2[0]`?
  A 1.7: No a "slice" on slice is just a number and is pass by value. In this case, `mat3` will just be a number and changing its value will not change `mat2`.

- **Q 1.8**: Does our solution need to exactly match the reference?
  A 1.8: No, we allow an error margin of 1e-6.

- **Q 1.9**: The values in my matrices change after I print their slices.
  A 1.9: This is likely a memory leak. Double check your `deallocate_matrix` implementation. What happens when you call `print(mat[0])` is that it will create a slice, then immediately deallocate it after this line.

Task 2

- **Q 2.1**: Getting `<class 'AttributeError'> -- shape`.
  A 2.1: Make sure the relevant method in src/numc.c is setting `mat->shape`

- **Q 2.2**: Getting `<class 'SystemError'> -- <built-in method set of numc.Matrix object at 0x7f175faffef0> returned NULL without setting an error!`
  A 2.2: You need to throw an error before returning `NULL` in any interface

function.

- **Q 2.3**: `UnicodeDecodeError: 'utf-8' codec can't decode byte 0x89 in position 5: invalid start byte` upon import. What's wrong?
  A 2.3: `{NULL, NULL, 0, NULL}` from starter code must be last element of `Matrix61c_methods`, so make sure you don't accidentally remove that!

- **Q 2.4**: Are we allowed to call methods we wrote in **`src/matrix.c`**?
  A 2.4: Yes please do!

- **Q 2.5**: How do we know what gets passed into `PyObject* args`?
  A 2.5: Read the official documentation! (You can also take a look at the starter code for examples.)

- **Q 2.6**: Is it safe to cast `PyLong_AsLong` as int?
  A 2.6: Yes! You can assume that all integer values in this project will never overflow.

- **Q 2.7**: Does the sign of 0.0 matter when negating matrices?
  A 2.7: No.

- **Q 2.8**: Should `Matrix61c_multiply` cover both `nb_matrix_multiply` and `nb_multiply`, or just `nb_multiply`?
  A 2.8: Just `nb_multiply`. `nb_matrix_multiply` corresponds to the operator `@`.

- **Q 2.9**: My `dumbpy` and `numc` matrices are exactly the same, however, comparing them using `==` returns False.
  A 2.9: You cannot compare them like that as they are of different object types. Please use `cmp_dp_nc_matrix`, which is provided in

`unittests/utils.py`.

Task 3

- **Q 3.1**: `#pragma omp parallel for` above a simple for loop slows down my add_matrix drastically?
  A 3.1: You need big matrices to actually see the effect of speedup (something around 1000 * 1000).

- **Q 3.2**: I'm getting `warning: ignoring #pragma omp parallel [-Wunknown-pragmas]` but I have not changed the Makefile?
  A 3.2: It might be your `setup.py` file. Take a look at the compiler and linker flags we provided you.

Autograder-related Questions

- **Q 4.1**: I'm passing my local correctness tests but failing autograder. What might be the case?
  A 4.1: Here are some common sources of bugs:
  - Make sure you test your matrix operations on non-integer floating point values (such as 1.2, 1.3). In the past semester, some students' code was only able to generate correct results when every floating point value in their matrices were integer-valued floating points (such as 1.0, 2.0).
  - Make sure you are setting the `shape` attribute of every returned matrix!
  - Please do NOT modify anything related to generating random matrices! Do NOT try to speed them up! We use those to generate matrices to test on the autograder. We don't include the time for generating matrices when testing for performance.
  - Please make sure that you have removed all your debug print statements.

- **Q 4.2**: Performance tests are failing but correctness tests all pass.
  A 4.2: Our performance tests test on much larger matrices than the correctness tests, so make sure you handle those cases correctly!

- **Q 4.3**: Failing tests that are related to throwing errors.
  A 4.3: Make sure you are throwing the correct type of error, not just any error.

- **Q 4.4**: Failing abs correctness test, even though my `abs_matrix` logic is very simple.
  A 4.4: Check if you are using C's built-in `abs` function. If so, you probably want to manually write another one since the built-in `abs` function will round off all decimal places.

Testing Questions

- **Q 5.1**: Why are there `TODO: YOUR CODE HERE` comments in the test functions in `unittests.py` even though these functions have already been implemented for us?
  A 5.1: The tests we have provided are VERY simple and they are purely examples to help you write more tests. Please modify these tests or add more tests for more comprehensive testing.
- **Q 5.2**: My tests are failing inside `rand_md5`, is this a bug from the starter code?
  A 5.2: No. Make sure your implementations `Matrix61c_get_value` and `Matrix61c_get_value` are correct! Otherwise `rand_md5` might error.

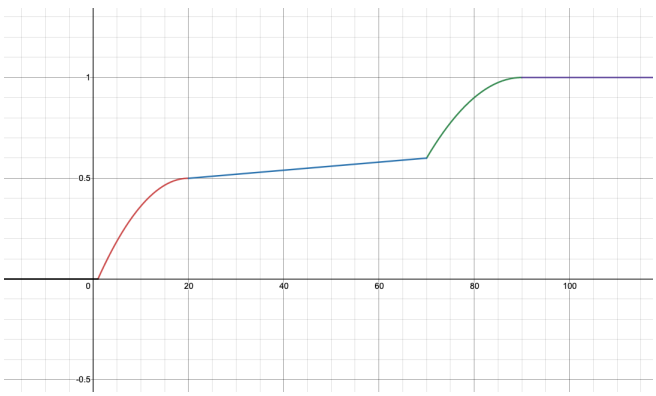
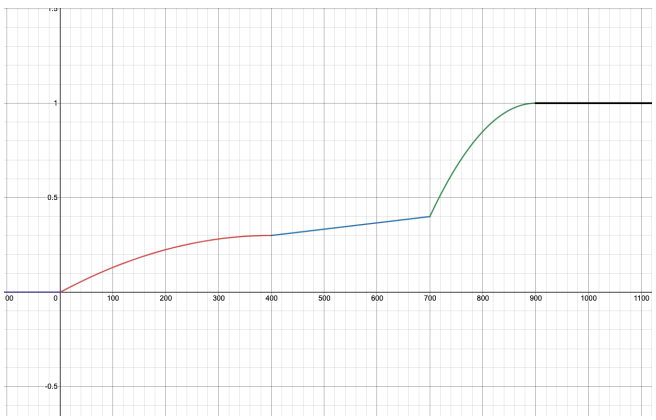
Grading

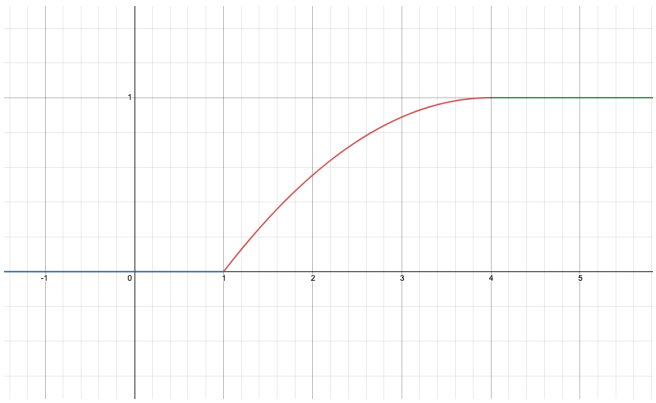
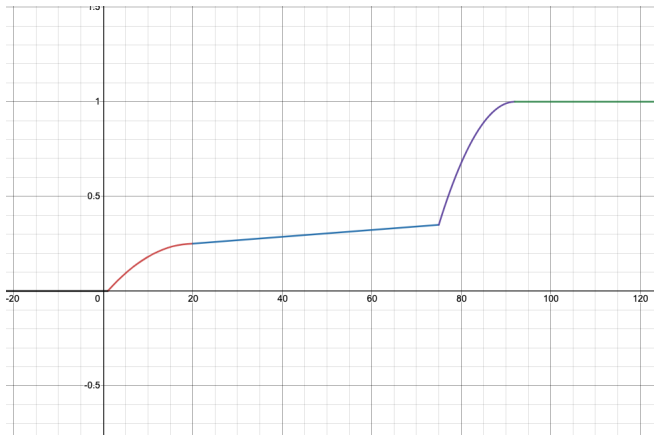
The grading breakdown for Project 4 is as follows:

- Correctness: 55%
  - Visible (**25 pts**):
    - Provided Basic Correctness Tests (25 **pts**)
  - Hidden (**30 pts**):
    - Wrong Args Test (6 **pts**)
    - Memory Leak Test (7 **pts**)
    - Dimension Test (4 **pts**)
    - Slice After Matrix Test (2 **pts**)
    - Matrix After Slice Test (2 **pts**)
    - Indexing Test (3 **pts**)
    - Get/Set Index Error Test (2 **pts**)
    - Function not in-place test (4 **pts**)
- Speedup [**Visible**]: 40%
  - Multiplication: 20%
  - Power: 25%
  - Simple: 12.5%
  - Comprehensive: 42.5%
- README.md [**Hidden**]: 5%

**Updated on August 3rd: ** Here are the graphs for the speedup tests. The x-axis is your speedup times and the y-axis indicates what percentage of that test's total score you will receive.

** Updated on August 6th: Graphs are still being updated **

Multiplication	**Power**
	

Simple	Comprehensive
	

[Benchmark update : 08/06] Minimum speedup for 100% on each test:

- Multiplication: 83x
- Power: 850x
- Simple: 4x
- Comprehensive: 85x

Since we are running your submissions on hive, albeit reserved, speedup times may fluctuate a bit. You should try to go above the speedup value as we will rerun the ag after the deadline and your speedup may go up or down. We will not rerun submissions if they went down unless we made another change to the autograder.

We will only be using your `src/matrix.h`, `src/matrix.c`, `src/numc.c`, and `README.md` for grading. Gradescope only shows you all the core correctness tests that are needed to run performance tests. Other tests will stay hidden until the late due date. **Keep in mind that your `Matrix61c_get` method in `src/numc.c` as well as the shapes you set for your matrices need to be correct in order for you to pass any of those correctness tests.**

Submitting Your Code

Please submit using Gradescope to Project 4, using the GitHub submission option to ensure that your files are in the right place.

Warning: If you have a partner, you **MUST** add your partner to every submission on Gradescope or it may be flagged for cheating. The person submitting can change, but both partners must be added to every submission.