PhysiCL: An OpenCL-Accelerated Python Physics Simulator

Benjamin Warner

Department of Computer Science, Pacific Lutheran University, Tacoma, WA 98447, USA

[a]Corresponding author: warnerbc@plu.edu, b.c.warner@wustl.edu

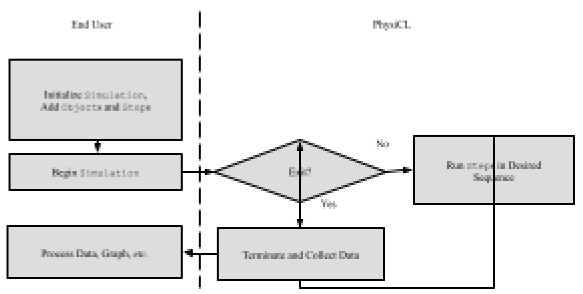
Abstract. Numerical methods of physics analysis require specialized forms of programming as well as attention to issues of implementation. PhysiCL is a Python package that aims to provide general-purpose tools for performing OpenCL-accelerated physics simulations with ease. PhysiCL contains a Numpy-based code units system, a set of generic simulation tools, built-in tools for photon scattering, tools for measuring light behavior, and tools for writing new OpenCL-based simulation features. This package can be installed via PyPI using `pip install physicl`, and found on GitHub with source code and examples at `https://github.com/bcwarner/physicl.`

Introduction

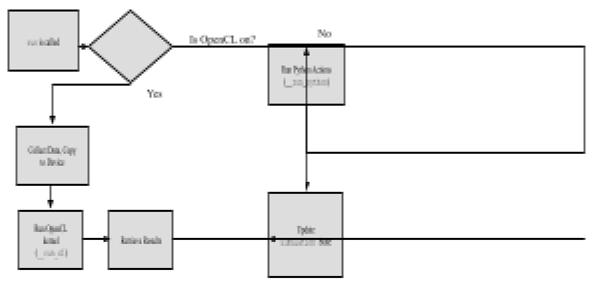
PhysiCL is a Python library that utilizes OpenCL to accelerate physics simulations, and it is intended to make writing physics simulations easier for both students and researchers. Currently, the feature set is designed primarily to work with simulations involving light scattering, and future work may expand it beyond this. We shall examine the basic usage of PhysiCL, its OpenCL metaprogramming tools, its code units system, and its base light scattering system.

Features

Basic Simulation Operation


End User
PhysiCL
Initialize Simulation, Add Objects and Steps
Begin Simulation
End?
No
Yes
Run steps in Desired Sequence
Terminate and Collect Data
Process Data, Graph, etc.

(a)



(b)

FIGURE 1. Outline of a basic simulation conducted using PhysiCL. (a) Outline of a general simulation, with the area left of the dashed line showing what the end user controls and the right showing what is abstracted away from the end user. (b) Outline of the operation of a `Step` in PhysiCL. A Python or OpenCL is run depending on whether OpenCL is turned on by the user.

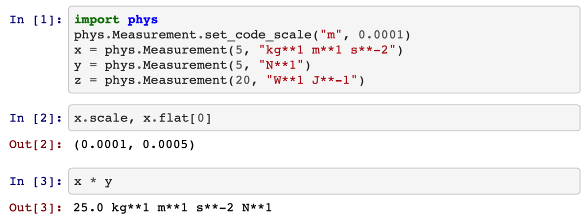
At the root of the PhysiCL package is `Simulation`. It is primarily defined by an *exit condition*, two values representing Δt and t, an OpenCL context and command queue, a list of `Objects` and a list of `Steps`. When we call `start`—which will call `run` on a separate thread—some setup is performed. We set t and Δt to zero, get the starting timestamp, and initialize a list for keeping track of ts that were simulated. Then, while the exit condition returns false, we call `run` on each `Step`. After the exit condition returns true, we call `terminate` on each `Step` for it to perform any clean up. An overview of this process appears in Fig. 1a. Each `Step` utilizes three main methods, `__init__`, `run`, and `terminate`, as well as `__run_cl` and `__run_python`. The first is used to initialize the `Step`, the second one is used when the simulation runs each `Step`, and the third one is called when the simulation is finished running. The last two are called to run the simulation using a parallelized OpenCL implementation or native Python implementation, which allows for the comparison of their relative performance. Users who wish to write new `Steps` will primarily modify `__init__` and `run`, in addition to both `__run_cl` and `__run_python` if such a comparison is desired. An overview of a typical process the latter three functions are used in is shown in Fig 1b.

There are two primitive steps in the root module of PhysiCL. The first, `UpdateTimeStep`, is initialized with a function that takes the simulation as an argument, determines what Δt should be and updates t accordingly. The second, `MeasureStep`, is a generic class for measuring the states of simulations that subclasses will override as needed. In general, users will extend `MeasureStep` or one of its subclasses to measure behavior, and user extensions upon `Step` will represent state-altering behavior. Another `Step` worth noting is `physicl.newton.NewtonianKinematicsStep`, which updates the position of objects according to their velocity.

OpenCL and Metaprogramming

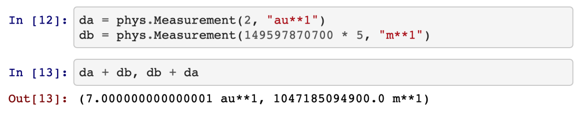
PhysiCL relies on OpenCL and PyOpenCL[3] to achieve accelerated computation, and uses metaprogramming techniques—where new code is generated dynamically—in concert to provide increased speed in developing and executing simulations. OpenCL is a library that allows for parallel computing on a variety of devices, using programs known as *kernels.*[1,2] Kernels may be metaprogrammed using PhysiCL's `CLProgram`. `CLProgram` represents a partially written OpenCL kernel as well as the Python code needed to provide it input and retrieve output. There are two stages to metaprogramming using `CLProgram`. OpenCL kernels must be compiled before they are used, and when we first need to build our kernel, we call `build_kernel`. When this function is called, a completed OpenCL kernel is generated. From there, we can call `run`—which is done from within a `Step`—and the appropriate Python code to collect input data and run the kernel will be generated. After the kernel finishes, the resulting output of the kernel is then used to update the simulation as needed.

Measurements



```
In [1]: import phys
        phys.Measurement.set_code_scale("m", 0.0001)
        x = phys.Measurement(5, "kg**1 m**1 s**-2")
        y = phys.Measurement(5, "N**1")
        z = phys.Measurement(20, "W**1 J**-1")

In [2]: x.scale, x.flat[0]

Out[2]: (0.0001, 0.0005)

In [3]: x * y

Out[3]: 25.0 kg**1 m**1 s**-2 N**1
```

(a)



```
In [12]: da = phys.Measurement(2, "au**1")
         db = phys.Measurement(149597870700 * 5, "m**1")

In [13]: da + db, db + da

Out[13]: (7.000000000000001 au**1, 1047185094900.0 m**1)
```

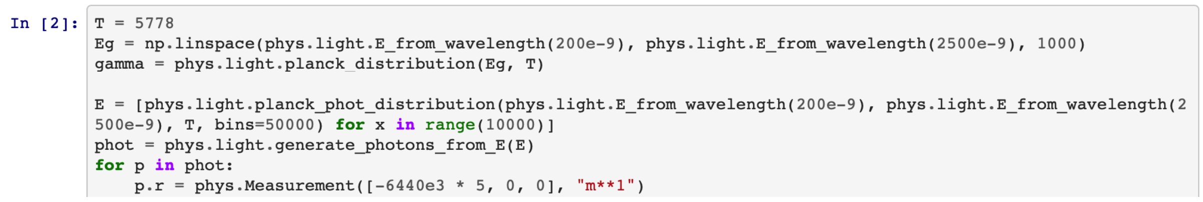
(b)

FIGURE 2. Three examples of the `Measurement` class demonstrating dimensional analysis and unit conversions. (a) Examples of binary and unary operations. (b) Example of two automatic unit conversions.

This subclass of Numpy's `ndarray`[7] represents an array of numbers that follows the *code units* scheme, which is where we scale units up or down to avoid floating point precision loss. Instances of this class are initialized with two parameters, `raw_value`, and `units`. `raw_value` can be another `ndarray`, another `Measurement`, or a list mixed with numbers and other `Measurements`. `units` is a string representing the units of the value, which are written as their corresponding symbol, either the Python power symbol (**) or a caret (^), and the dimension of the unit.

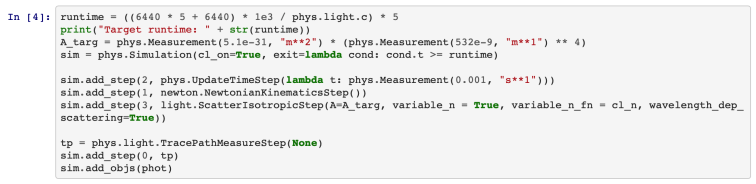
The scaling process starts by determining what units are being used. After isolating these units, they are recursively converted to their defining units until they have been reduced to the 7 fundamental SI units. After this, we convert these fundamental units to the code scale units.[4] Each of these code scale units carries a scaling factor, which is used to affect the scaling of all `Measurements` that rely on the same fundamental units. We multiply each element of our array by this scale. We store these dimensions, as well as the dimensions of the original units used for use whenever a string representation is needed. Each of the 7 fundamental SI units can be scaled up or down using `set_code_scale`, as seen in Fig. 2a. Currently, it is designed so that it can only be done once, before any related modules are imported. This is to optimize for speed, as performing checks to see if two `Measurements` have the same scale can be costly; however, it may be possible to use something akin to a counter to keep track of the current scale. After it is set up, it may be operated on by directly calling one of Numpy's `ufuncs`—such as `numpy.square`[5]—or by utilizing the corresponding Python operator, generating results with the appropriate underlying units and scale, as seen in Fig. 2. It will not cancel or reduce any original units passed to it for speed; however, a future implementation may reduce units lazily when a string representation is needed.

Photon Scattering, Measurement, and Generation



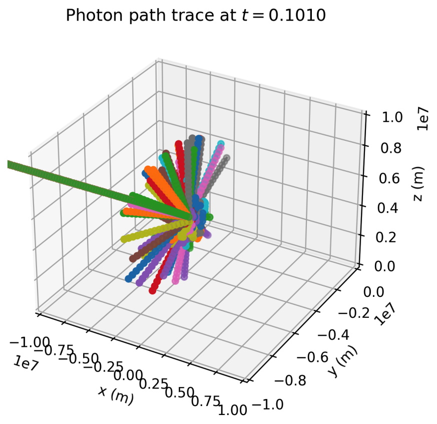
```
In [2]: T = 5778
        Eg = np.linspace(phys.light.E_from_wavelength(200e-9), phys.light.E_from_wavelength(2500e-9), 1000)
        gamma = phys.light.planck_distribution(Eg, T)

        E = [phys.light.planck_phot_distribution(phys.light.E_from_wavelength(200e-9), phys.light.E_from_wavelength(2
        500e-9), T, bins=50000) for x in range(10000)]
        phot = phys.light.generate_photons_from_E(E)
        for p in phot:
            p.r = phys.Measurement([-6440e3 * 5, 0, 0], "m**1")
```

(a)



```
In [4]: runtime = ((6440 * 5 + 6440) * 1e3 / phys.light.c) * 5
        print("Target runtime: " + str(runtime))
        A_targ = phys.Measurement(5.1e-31, "m**2") * (phys.Measurement(532e-9, "m**1") ** 4)
        sim = phys.Simulation(cl_on=True, exit=lambda cond: cond.t >= runtime)

        sim.add_step(2, phys.UpdateTimeStep(lambda t: phys.Measurement(0.001, "s**1")))
        sim.add_step(1, newton.NewtonianKinematicsStep())
        sim.add_step(3, light.ScatterIsotropicStep(A=A_targ, variable_n = True, variable_n_fn = cl_n, wavelength_dep_
        scattering=True))

        tp = phys.light.TracePathMeasureStep(None)
        sim.add_step(0, tp)
        sim.add_objs(phot)
```

(b)



(c)

FIGURE 3. An example simulation involving isotropic scattering around an approximation of the upper half of Earth's atmosphere with a photon distribution resembling that of the Sun, as well as output displayed in matplotlib. (a) A segment of the necessary code to set up a simulation involving a beam of photons being scattered, namely the creation of `PhotonObjects` drawn from `planck_phot_distribution`. (b) Another segment of the necessary code required to set up the simulation, including the addition of `Steps` and `PhotonObjects`. (c) The resulting paths of a beam of photons as they travel towards an object with arbitrarily high density.

Currently, the main feature of this package are the tools for photon scattering and measurement, which are in the `physicl.light` module. It includes `ScatterDeleteStep` and `ScatterIsotropicStep`, which represent photon absorption and isotropic scattering, respectively. It also includes `ScatterMeasureStep`, which measures photons passing through specified planes, `ScatterSignMeasureStep`, which measures the quantity of photons with a positive/negative sign along any axis, and `TracePathMeasureStep`, which tracks the path a photon takes.

The first class, `ScatterDeleteStep`, represents the scattering of photons as if they were being absorbed into a medium. It assumes that the medium has a specified uniform number density, n_{targ}, and cross-sectional area, A_{targ}, throughout the entire simulation space. When this step is run, it finds all `PhotonObjects` within the simulation, and collects the Δrs for all photons. It also generates a random number in $[0, 1)$ for each photon, and allocates two lists, one pointing to the photons examined, and another for the results of the OpenCL kernel it will run. The kernel will find $P_{coll} = n_{targ}A_{targ}\Delta r$, and if P_{coll} is greater than the random number generated for a particular photon, then it will be marked for removal. After the OpenCL kernel returns, the marked photons are deleted from the simulation.

`ScatterIsotropicStep` represents the scattering of photons as if they were being refracted into a random direction. Like `ScatterDeleteStep` it assumes a consistent n_{targ} and A_{targ} throughout the entire simulation, however the n_{targ} may be varied with an OpenCL expression, and if the user desires, scattering may also be dependent on the wavelength, as occurs with Rayleigh scattering. When this `Step` is run it first collects the Δr for each photon. It then generates a random $\theta \in [0, 2\pi), \phi \in [0, \pi)$ that will be used to derive a new direction for the photon to go, and a random number `rand` $\in [0, 1)$. Next, it collects the original $\vec{v}$ for the photons; if wavelength-dependent scattering is on, it collects each E_γ; and if variable n_{targ} is on, it collects the current r of each photon. Then when the kernel is completed, `ScatterIsotropicStep` will apply the changes calculated in the kernel. If the velocity was changed, the photon will have its $\Delta v \leftarrow v_{new} - v_{old}$. An example of this in use can be seen in Fig. 3.

`ScatterMeasureStep` measures the total quantity of photons within a simulation, as well as the number and energies of photons that pass through a plane at a given point in time. When initialized, the user may decide whether they want the total quantity of objects to be measured, coordinates for the planes where we should measure photons passing through, and whether the `ScatterDeleteStep` should also record the energies of the photons passing through. `ScatterSignMeasureStep` measures the number of objects within a simulation as well as the number of objects whose v_x, v_y, v_z have values greater than zero. `TracePathMeasureStep` tracks the position of each object throughout a simulation. When this step is run, it iterates through each object in the simulation and performs several steps. If the current object in the iteration does not have a unique identifier, it assigns one. `TracePathMeasureStep` then records the starting time, creates a list to store positions, and if the user desires, the frequency with which the photon changed velocity. Then, `TracePathMeasureStep` records the current position of the object, and if the velocity changes, increments the accumulator representing the frequency of velocity changes. After the simulation is complete, `TracePathMeasureStep` compiles the data collected into a two-dimensional array representing the ts that were recorded, with each row represents an individual object, the number of times its velocity changed if desired, and finally all positions that were recorded for each time. An example of `TracePathMeasureStep` can be seen in Fig. 3, where its output data is graphed to show atmospheric refraction.

In addition to these tools to simulate and measure photons, there is `planck_phot_distribution`, which randomly generates a series of photon energies according to a desired segment of the Planck distribution. `planck_phot_distribution` works by finding the total area under a Planck distribution curve for a desired number of bins, normalizing the total area under these bins so that it equals 1, and finally randomly picking an energy bin using our normalized distribution. There are also two other ways to generate photons. `generate_photons_from_E` takes a list of E_γ and generates new photons for each energy given with a velocity of c in the $+x$ direction. `generate_photons`, takes a function that generates random numbers, minimum and maximum energies, and a desired number of photons, and returns a list of `PhotonObjects`.

ACKNOWLEDGMENTS

I would like to acknowledge my physics professor, Dr. Sean O’Neill, for his guidance and feedback on this project. I would also like to acknowledge my capstone advisor, Dr. David Wolff, for providing feedback and facilitating this project. In addition, I would like to acknowledge Megan Longstaff, a fellow physics student who connected me with Dr. O’Neill. Finally, I would like to acknowledge Dr. Katrina Hay, Dr. William Greenwood, and

Professor Laurie Murphy for advising my academic career so far, without them this project would not have happened.

References

1. *Heterogeneous Computing with OpenCL*, edited by B. Gaster, (Morgan Kaufmann, Waltham, 2012), p. 277.
2. *OpenCL Programming Guide*, edited by A. Munshi (Addison-Wesley. Upper Saddle River, 2012), p. 603.
3. A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, Parallel Computing **38**(3):157–74 (2012).
4. Bureau International des Poids et Mesures, *The International System of Units 9th ed.* (2019), available from https://www.bipm.org/utils/common/pdf/si-brochure/SI-Brochure-9.pdf.
5. S. van der Walt, S.C. Colbert, and G. Varoquaux, Comput Sci Eng. **13**(2):22–30 (2011).