# PySPH: a reproducible and high-performance framework for smoothed particle hydrodynamics

Prabhu Ramachandran[‡§*]

https://youtu.be/6UnuPhTPdnM

**Abstract**—Smoothed Particle Hydrodynamics (SPH) is a general purpose technique to numerically compute the solutions to partial differential equations such as those used to simulate fluid and solid mechanics. The method is grid-free and uses particles to discretize the various properties of interest (such as density, fluid velocity, pressure etc.). The method is Lagrangian and particles are moved with the local velocity.

PySPH is an open source framework for Smoothed Particle Hydrodynamics. It is implemented in a mix of Python and Cython. It is designed to be easy to use on multiple platforms, high-performance and support parallel execution. Users write pure-Python code and HPC code is generated on the fly, compiled, and executed. PySPH supports OpenMP and MPI for distributed computing, in a way that is transparent to the user. PySPH is also designed to make it easy to perform reproducible research. In this paper we discuss the design and implementation of PySPH.

## Background and Introduction

SPH (Smoothed Particle Hydrodynamics) is a general purpose technique to numerically compute the solutions to partial differential equations used to simulate fluid and solid mechanics. The method is grid-free and uses particles to discretize the various properties of interest. The method is Lagrangian and particles are moved with the local velocity. The method was originally developed for astrophysical problems [Luc77], [GM77] (compressible gas-dynamics) but has since been extended to simulate incompressible fluids [Mon94], solid mechanics [GMS01], free-surface problems [Mon94] and a variety of other problems. Monaghan [Mon05], provides a good review of the method.

The SPH method is relatively easy to implement. This has resulted in a large number of schemes and implementations proposed by various researchers. SPH schemes differ in the details of how the governing equations are approximated. It is often difficult to reproduce published results due to the variety of implementations. While a few standard packages like SPHysics [devb], DualSPHysics [deva], JOSEPHINE [CPR12], GADGET-2 [Spr05] etc. exist, they are usually tailor-made for particular applications and are not general purpose. They are all implemented in FORTRAN (77 or 90) or C, and do not have a convenient Python interface.

* *Corresponding author: prabhu@aero.iitb.ac.in*
‡ *Department of Aerospace Engineering*
§ *IIT Bombay, Mumbai, India*



Our group has been developing PySPH (http://pysph.bitbucket.org) over the last 5 years. PySPH is open source, and distributed under the new BSD license. Our initial implementation was based on Cython [BBC+11] and also featured some parallelization using MPI. This was presented at SciPy 2010 [RK10]. Unfortunately, this previous version of PySPH proved difficult to use as users were forced to implement most of their code in Cython. This was not a matter of simply writing a few high performance functions in Cython. The PySPH library is object oriented and supporting a new SPH formulation would require subclassing one or more classes and this would need to be done with Cython. This made the design more rigid as all the types needed to be pre-defined. Writing all this in Cython meant that users had to manage compilation and linking the Cython code during development. This made development with PySPH inconvenient.

It was felt that we might as well have implemented the core library in C++ and exposed a Python interface to it. A traditional compiled language has more developer tooling around it. For example debugging, performance tuning, profiling would all be easier if everything were written in C or C++. Unfortunately, such a mixed code-base would not be as easy to use, extend or maintain as a largely pure Python library. In our experience, a pure Python library is a lot easier for say an undergraduate student to grasp and use over a C/C++ code. Others are also finding this to be true [Per15]. Many of the top US universities are teaching Python as their first language [Guo14]. This means that a Python library would also be easier for relatively inexperienced programmers. It is also true that a Python library would be easier and shorter to write for the other non-high-performance aspects (which is often a significant amount of code). So it seemed that our need for performance was going against our desire for an easy to use Python library that could be used by programmers who were not C/C++ developers.

In early 2013, we redesigned PySPH so that users were able to implement an entire simulation using pure Python. This was done by auto-generating HPC code from the pure Python code that users provided. This version ended up being faster than our original Cython implementation! Since we were auto-generating code, with a bit of additional effort it was possible to support OpenMP as well. The external user API did not change so users did not have to modify their code at all to benefit from this development. PySPH has thus matured into an easy to use, yet high-performance framework where users can develop their schemes in pure Python and yet obtain performance close to that of a lower-level language implementation. PySPH has always supported running on a cluster

of machines via MPI. This is seamless and a serial script using PySPH can be run with almost no changes using MPI.

PySPH features a reasonable test-suite and continuous integration servers are used to test it on Linux and Windows. The documentation is hosted at http://pysph.readthedocs.org. The framework supports several of the standard SPH schemes. A suite of about 30 examples are provided. These are shipped as part of the sources and installed when a user does a pip install. The examples are written in a way that makes it easy to extend and also perform comparisons between schemes. These features make PySPH well suited for reproducible numerical work. In fact one of the author's recent papers [RP16] was written such that every figure in the paper is automatically generated using PySPH.

In this paper we discuss the use, design, and implementation of PySPH. In the next section we provide a high-level overview of the SPH method.

### Smoothed Particle Hydrodynamics

The SPH method works by approximating the identity:

$$f(x) = \int f(x')\delta(x-x')dx',$$

where, $\delta$ is the Dirac Delta distribution. This identity is approximated using:

$$f(x) \approx \int f(x')W(x-x',h)dx', \tag{1}$$

where $W$ is a smooth and compact function and is called the kernel. It is an approximate Dirac delta distribution that is parametrized on the parameter $h$ and $W \to \delta$ as $h \to 0$. $h$ is called the smoothing length or smoothing radius of the kernel. The kernel typically will need to satisfy a few properties if this approximation is to be accurate. Notably, its area should be unity and if it is symmetric, it can be shown that the approximation is at least second order in $h$. The above equation can be discretized as,

$$f(x) \approx \langle f(x) \rangle = \sum_{j \in \mathcal{N}(x)} W(x-x_j,h) f(x_j) \Delta x_j, \tag{2}$$

where $x_j$ is the position of the particle $j$, $\Delta x_j$ is the volume associated with this particle. $\mathcal{N}(x)$ is the set of particle indices that are in the neighborhood of $x$. In SPH each particle carries a mass $m$ and associated density $\rho$ with it and the particle volume is typically chosen as $\Delta x_j = m_j/\rho_j$. This results in the following SPH approximation for a function,

$$< f(x) >= \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j} W(x-x_j,h) f(x_j). \tag{3}$$

Derivatives of functions at a location $x_i$ are readily approximated by taking the derivative of the smooth kernel. This results in,

$$\frac{\partial f_i}{\partial x_i} = \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j}(f_j - f_i)\frac{\partial W_{ij}}{\partial x_i}. \tag{4}$$

Here $W_{ij} = W(x_i - x_j)$. Similar discretizations exist for the divergence and curl operators. Given that derivatives can be approximated one can solve differential equations fairly easily. For example the conservation of mass equation for a fluid can be written as,

$$\frac{d\rho}{dt} = -\rho \nabla \cdot \vec{v}, \tag{5}$$

where $v$ is the velocity of the fluid and the LHS is the material or total derivative of the density. The equation 5 is in a Lagrangian form, in that it represents the rate of change of density as one is moving locally with the fluid. If an SPH discretization of this equation were performed we would get,

$$\frac{d\rho_i}{dt} = -\rho_i \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j} \vec{v}_{ji} \cdot \nabla_i W_{ij}, \tag{6}$$

where $\vec{v}_{ji} = \vec{v}_j - \vec{v}_i$. This equation is typical of most SPH discretizations. SPH can therefore be used to discretize any differential equation. This works particularly well for a variety of continuum mechanics problems. Consider the momentum equation for an inviscid fluid,

$$\frac{d\vec{u}}{dt} = -\frac{1}{\rho}\nabla p \tag{7}$$

A typical SPH discretization of this could be written as,

$$\frac{d\vec{u}_i}{dt} = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) \nabla W_{ij} \tag{8}$$

More details of these and various other equations can be seen in the review by Monaghan [Mon05]. It is easy to see that equations 6 and 8 are ordinary differential equations that govern the rate of change of the density and velocity of a fluid particle. In principle, one can integrate these ODEs to obtain the flow solution given a suitable initial condition and appropriate boundary conditions.

### Numerical implementation

As discussed in the previous section, in an SPH scheme, the field properties are first discretized into particles carrying them. Partial differential equations are reduced to a system of coupled ordinary differential equations (ODEs) and discretized using an SPH approximation. This results in a system of ODEs for each particle. These ODEs need to be integrated in time along with suitable boundary and initial conditions in order to solve a particular problem. To summarize, a typical SPH computation proceeds as follows,

- Given an initial condition, the field variables are discretized into particles carrying the various properties.
- Depending on the scheme used to integrate the ODEs, the RHS of the ODEs needs to be computed (see equations 6 and 8). These RHS terms are called "accelerations" or "acceleration terms".
- Once the RHS is computed, the ODE can be integrated using a suitable scheme and the fluid properties are found at the next timestep.

The RHS is typically computed as follows:

- Initalize the particle accelerations (i.e. the RHS terms).
- For each particle in the flow, identify the neighbors of the particle which will influence the particle.
- For each neighbor compute the acceleration due to that particle and increment the acceleration.

Given the total accelerations, the ODEs can be readily integrated with a variety of schemes. Any general purpose abstraction of the SPH method must hence provide functionality to:

1) Easily represent the discretized properties of particles. This is easily done with `numpy` arrays representing the property values in Python.
2) Given a particle, identify the neighbors that influence the particle. This is typically called Nearest Neighbor Particle Search (NNPS) in the literature.

3) Define the interactions between the particles, i.e. an easy way to specify the inter particle accelerations. In PySPH these are called "Equations".
4) Define how the ODEs should be integrated.

Of the above, the NNPS algorithm is usually a well-known algorithm. For incompressible flows where the smoothing radius of the particles, $h$, is constant, a simple bin-based linked list implementation is standard. For cases where $h$ varies, a tree-based algorithm is typically used. Users usually do not need to experiment or modify the NNPS. PySPH allows the rest of the tasks to be all implemented in pure Python.

### The PySPH framework

PySPH allows a user to specify the inter-particle interactions as well as the ODE integration in pure Python with a rather simple and low-level syntax. This is described in greater detail further below. As discussed in the introduction, with older versions of PySPH as discussed in [RK10], these interactions would all need to be written in Cython. This was not very easy or convenient. It was also rather limiting.

The current version of PySPH supports the following:

- Define a complete SPH simulation entirely in Python.
- High-performance code is generated from this high-level Python code automatically and called. The performance of this code is comparable to hand-written FORTRAN solvers.
- PySPH can use OpenMP seamlessly. Users do not need to modify their code at all to use this. This works on Linux, OS X, and Windows, and produces good scale-up.
- PySPH also works with MPI and once again this is transparent to the user in that the user does not have to change code to use multiple machines. This feature requires mpi4py and Zoltan to be installed.
- PySPH provides a built-in 3D viewer for the particle data generated. The viewer requires Mayavi [RV11] To be installed.
- PySPH is also open-source and currently hosted at http://pysph.bitbucket.org

Currently, PySPH supports the simulation of compressible and incompressible fluid flows (with and without free-surfaces), simple rigid-body motion, and elastic dynamics for solids. It does not support astro-physical simulations since it lacks the tree-code needed to simulate graviational forces. This can be added but is not the current focus.

In the following subsection we provide a high-level overview of PySPH and see how it can be used by a user. Subsequent subsections discuss the design and implementation in greater detail.

#### *High-level overview*

PySPH is tested to work with Python-2.6.x to 2.7.x and also with Python 3.4/3.5. PySPH is a typical Python package and can be installed fairly easily by running:

```
$ pip install pysph
```

PySPH will require a C++ compiler. On Linux, this is trivial to get and usually pre-installed. On OS X, clang will work as will gcc (which can be easily installed using brew). On Windows the Visual C++ Compiler for Python will need to be installed. Detailed instructions for all these are available from the PySPH documentation.

If one wishes to use OpenMP,

- On Linux one needs to have libgomp installed.
- On OS X one needs to install OpenMP for clang or one could use GCC which supports OpenMP via brew.
- On Windows, just having the Visual C++ computer for Python will work.

If one wishes to use MPI for distributed computing, one must install Zoltan which is typically easy to install. PySPH provides a simple script for this. mpi4py is also needed in this case. Zoltan is used for load-balancing and distributing the particles efficiently on distributed machines. Unfortunately, MPI is not tested on Windows by us currently. PySPH also provides an optional 3D viewer and this depends on Mayavi.

In summary, PySPH is easy to install if one has a C++ compiler installed. MPI support is a little involved due to the requirement to install Zoltan.

Once PySPH is installed an executable called `pysph` is available. This is a convenient entry point for various tasks. Running `pysph -h` will provide a listing of these possible tasks. For example, the test suite can be run using:

```
$ pysph test
```

This uses nose internally and can be passed any arguments that `nosetests` accepts.

PySPH installs about 30 useful examples along with the sources and any of these examples can be readily run. For example:

```
$ pysph run
1. cavity
   Lid driven cavity using the Transport Velocity
   formulation. (10 minutes)
[...]
Enter example number you wish to run:
```

Provides a listing of the examples available and prompts for a particular one. Each example also provides a convenient (but rough) time estimate for the example to run to completion in serial. If the name of the example is known, one may directly specify it as:

```
$ pysph run elliptical_drop
```

The examples will accept a large number of command line arguments. To find these one can run:

```
$ pysph run elliptical_drop -h
```

`pysph run` will execute the standard example. Note that internally this is somewhat equivalent to running:

```
$ python -m pysph.examples.elliptical_drop
```

The example may therefore be imported in Python and also extended by users. This is by design.

When the example is run using `pysph run`, the example documentation is first printed and then the example is run. The example will typically dump the output of the computations to a directory called `example_name_output`, in the above case this would be `elliptical_drop_output`. This output can be viewed using the Mayavi viewer. This can be done using:

```
$ pysph view elliptical_drop_output
```

This will start up the viewer with the saved files dumped in the directory. Figure 1 shows the viewer in action. The viewer

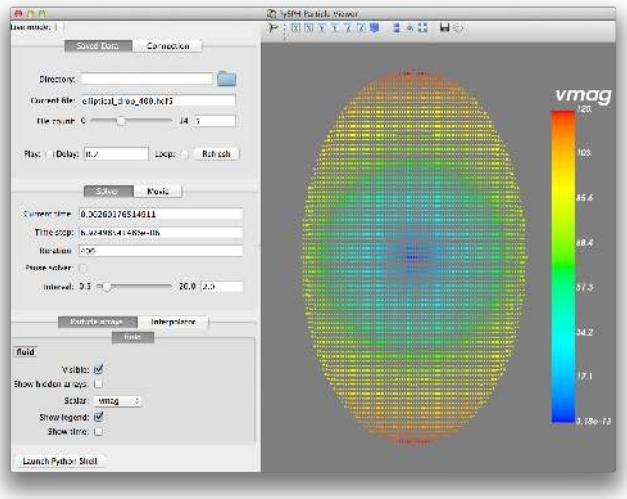


***Fig. 1:*** *The viewer provides a convenient interface to view data dumped by simulations.*

provides a very convenient interface to view the data. On the right side, one has a standard Mayavi widget which also features a Mayavi icon on the toolbar. Clicking this will open the Mayavi UI with which one can easily change the visualization. On the left pane there are three sub panels. On the top, one can see a slider for the file count. This can be used to move through the simulation in time. This can be also animated by checking the "Play" checkbox which will iterate over the files. The "Directory" button allows one to view data from a different output directory. Hitting the refresh button will rescan the directory to check for any new files. This makes it convenient to visualize the results from a running simulation. The "Connection" tab can be used when the visualization is in "Live mode" when it can connect to a running simulation and view the data live. While this is very useful in principle, it is seldom used in practice as it is a lot more efficient to just view the dumped files and use the "Refresh" button is convenient. Regardless, it does show another feature of PySPH in that one can actually pause a running simulation and query it if needed. Below this pane is a "Solver" pane which shows the various solver parameters of interest. The "Movie" tab allows a user to dump screenshots and easily produce a movie if needed. At the bottom of the interface are two panels called "Particle arrays" and "Interpolator". The particle arrays lists all the particles and different scalar properties associated with the SPH simulation. Selecting different scalars will display those scalars. The interpolator tab allows a user to specify a rectilinear region on which the particle properties may be interpolated and visualized -- for example if one wishes to see a contour of velocity magnitudes this would be useful. Right at the bottom is a button to launch a Python shell. This can be used for advanced scripting and is seldom used by beginners. This entire viewer is written using about 1024 lines of code and ships with PySPH.

PySPH output can be dumped either in the form of `.npz` files (which are generated by NumPy) or HDF5 files if h5py is installed. These files can be viewed using other tools or with Python scripts if desired. The HDF5 in particular can be viewed more easily. In addition, the `pysph dump_vtk` command can be used to dump VTK output files that can be used to visualize the output using any tool that supports VTK files like ParaView etc. This can use either Mayavi or can use pyvisfile which has no dependency on VTK. Finally, the saved data files can be loaded in Python very easily, for example:

```
from pysph.solver.utils import load
data = load('elliptical_drop_100.hdf5')
# if one has only npz files the syntax is the same.
data = load('elliptical_drop_100.npz')
```

This provides a dictionary from which one can obtain the particle arrays and solver data:

```
particle_arrays = data['arrays']
solver_data = data['solver_data']
fluid = particle_arrays['fluid']
p = fluid.p
```

where `particle_arrays` is a dictionary of all the PySPH particle arrays. `solver_data` is another dictionary with solver properties and `p` is a NumPy array of the pressure of each particle. Particle arrays are described in greater detail in the following sections. Our intention here is to show that the dumped data can be very easily loaded into Python if desired.

As discussed earlier, PySPH supports OpenMP and MPI. To use multiple cores on a computer one can simply run an example or script as:

```
$ pysph run elliptical_drop --openmp
```

This will use OpenMP transparently and should work for all the PySPH examples. PySPH will honor the `OMP_NUM_THREADS` environment variable to pick the number of threads. If PySPH is installed with MPI support through Zoltan, then one may run for example:

```
$ mpirun -np 4 pysph run dam_break_3d
```

This will run the `dam_break_3d` example with 4 processors. The amount of scale-up depends on the size of the problem and the network. OpenMP will scale fairly well for moderately sized problems. Note that for a general PySPH script written by the user, the command to run would simply be:

```
$ mpirun -np 4 python my_script.py
```

Similarly when using OpenMP:

```
$ python my_example.py --openmp
```

This provides a very high-level introduction to PySPH in general. The next section discusses some essential software engineering used in the development of PySPH. This is followed by details on the underlying design of PySPH.

### *Essential software engineering*

PySPH follows several of the standard software development practices that most modern open source implementations follow. For example:

- Our sources are hosted on bitbucket (http://pysph.bitbucket.org). We are thinking of shifting to GitHub because GitHub has much better integration with continuous integration services and this is a rather frustrating pain point with bitbucket.
- We use pull requests to review all new features and bug fixes. At this point there is only a single reviewer (the author) but this should hopefully increase over time.
- PySPH has a reasonable set of unit tests and functional tests. Each time a bug is found, a test case is first created

(when possible or reasonable), and then fixed. nose is used for discovering and executing tests. One of our functional tests runs one time step of every single example that ships with PySPH. tox based tests are also supported. This makes it easy to test on Python 2.6, 2.7 and 3.x.

- We use continuous integration services from http://shippable.com for Linux, http://appveyor.com for Windows and http://codeship.com for faster Linux builds.
- Our documentation is generated using Sphinx and hosted online on http://pysph.readthedocs.io.
- Releases are pushed to the Python Package Index (PyPI).
- The pysph-users mailing list is also available where users can post their questions. Unfortunately, the response time is currently slow as the author does not have the time for this but we are hoping this will improve as more graduate students start getting involved with PySPH.

These greatly improve the quality, reliability and usability of the software and also encourage open collaboration.

### *Design overview*

In the previous sections a high-level description of the project was provided. This section provides more design details of how PySPH works internally. The general approach used in PySPH is as follows:

1) Create particles: discretize the initial materials into particles with suitable properties.
2) Choose an appropriate kernel for the SPH approximation.
3) Create equations: write out the equations that specify the inter-particle interactions.
4) Setup the integrator and specify the integration steps, for example one could use an Euler scheme or a predictor-corrector scheme and each of these involve slightly different integration steps. These need to be specified explicitly.

PySPH allows a user to do all of these from pure Python.

1) In PySPH, particles of a particular kind are managed by a `ParticleArray` instance. A particle array is assigned a unique name and manages a collection of properties. Each property is internally represented as a contiguous block of memory. All properties have the same number of elements. A particle array may also have any number of "constants" associated with it. Each constant can be a scalar or an array but its size is independent of the number of particles.
2) The kernels are implemented in pure Python and a default collection of kernels is available in `pysph.base.kernels`. A new kernel class would implement the following methods, note that the default arguments have no meaning except that they help the code generator use the correct types:

```
class MyKernel(object):
    def __init__(self, dim):
        # ...
    def kernel(self, xij=[0., 0, 0], rij=1.0,
               h=1.0):
        # ...
    def gradient(self, xij=[0., 0, 0], rij=1.0,
                 h=1.0, grad=[0, 0, 0]):
        # ...
```

3) In PySPH, the equations can also be created in pure Python and this is discussed in detail in the following.
4) The integrators are split into two parts, an integrator and an integrator step. This is also written in pure Python and discussed with an example further below.

A typical example is considered first to illustrate the design. Consider the example `pysph/pysph/examples/elliptical_drop.py`. When installed, this may be imported as `import pysph.examples.elliptical_drop`. This example simulates the evolution of a fluid drop that is initially circular and imposed an initial velocity field of the form $\vec{V} = -100x\hat{i} + 100y\hat{j}$. This problem is a simple benchmark problem that was first solved in the context of SPH by [Mon94]. The key parts of the example are shown below:

```
from numpy import array, ones_like, mgrid, sqrt

# PySPH base and carray imports
from pysph.base.utils import get_particle_array
from pysph.base.kernels import Gaussian

# PySPH solver and integrator
from pysph.solver.application import Application
from pysph.sph.integrator import EPECIntegrator
from pysph.sph.scheme import WCSPHScheme

class EllipticalDrop(Application):
    def initialize(self):
        # ...
    def create_particles(self):
        # ...
    def create_scheme(self):
        # ...
    def post_process(self, info_file_or_dir):
        # ...

if __name__ == '__main__':
    app = EllipticalDrop()
    app.run()
    app.post_process(app.info_filename)
```

This illustrative example deliberately excludes several details to focus on the general structure and API. There are a few common imports at the top starting with NumPy specific imports first. The next imports are PySPH specific:

- `get_particle_array` is a convenient function that helps create a `ParticleArray` instance.
- The `Gaussian` kernel is used for the SPH simulation.
- The `Application` class is subclassed to create the new example.
- The `WCSPHScheme` encapsulates a particular scheme, in this case this class abstracts out the requirements for a weakly-compressible scheme applied to incompressible flows. Internally the WCSPH scheme is responsible to setup the equations and the integrator. By abstracting this into a scheme it becomes easy to reuse this instead of spelling out the equations for each example.

The typical entry point for a user is to subclass `Application` to solve their particular problem. The methods listed above are:

- `initialize`, this is automatically called by `Application.__init__` and is typically not used but sometimes useful when one wishes to have some common attributes setup.

- `create_particles` generates the initial particle distribution and returns a sequence of `ParticleArray` instances.
- `create_scheme` creates the particular scheme. A `SchemeChooser` is also available which can be given multiple schemes and allows the user to switch between them via command line arguments.
- the `post_process` method is run in the end to compute any useful quantities that may be used to check the accuracy of the simulation or facilitate comparisons between different schemes.

The `if __name__` block is listed to just illustrate how this application can be used. When `run` is called, the command line arguments are parsed, the various objects involved are suitably configured and the simulation executed. At the end, the `post_process` method is called. This also shows that a user could potentially rewrite the post processing code and simply rerun that part instead of re-running the simulation (which can sometimes run for days).

We next look inside the `create_particles` and `create_scheme` methods:

```
1  def create_particles(self):
2      x, y = mgrid[-1.:1.05:dx,-1.:1.05:dx]
3      x, y = x.ravel(), y.ravel()
4      m = ones_like(x)*dx*dx
5      h = ones_like(x)*hdx*dx
6      # ...
7      u = -100*x
8      v = 100*y
9
10     # remove particles outside the circle
11     indices = []
12     for i in range(len(x)):
13         dist = sqrt(x[i]*x[i] + y[i]*y[i])
14         if dist - 1 > 1e-10:
15             indices.append(i)
16
17     pa = get_particle_array(
18         x=x, y=y, m=m, rho=rho, h=h, p=p,
19         u=u, v=v, cs=cs, name='fluid')
20     pa.remove_particles(indices)
21     self.scheme.setup_properties([pa])
22     return [pa]
23
24 def create_scheme(self):
25     s = WCSPHScheme(
26         ['fluid'], [], dim=2, rho0=self.ro, c0=co,
27         h0=self.dx*self.hdx, hdx=self.hdx,
28         gamma=7.0, alpha=0.1, beta=0.0
29     )
30     kernel = Gaussian(dim=2)
31     dt = 5e-6; tf = 0.0076
32     s.configure_solver(
33         kernel=kernel,
34         integrator_cls=EPECIntegrator,
35         dt=dt, tf=tf, adaptive_timestep=True,
36         cfl=0.3, n_damp=50,
37     )
38     return s
```

The `create_particles` method above is straightforward. NumPy arrays are created that set the position, mass, smoothing radius $h$, the velocity etc. The arrays are all one dimensional. The indices that are outside the circle are identified between lines 11 and 14 and these are removed in line 20. This could have also been done with pure NumPy indexing. In Line 17 the particle array instance is created and is called `'fluid'`. Line 22 delegates to the `scheme` to setup any additional properties for the particle array and finally a list of particle arrays is returned.

The `create_scheme` method is fairly simple. A `WCSPHScheme` is instantiated and passed arguments as defaults. The kernel is created and this is all passed to a scheme method called `configure_solver`, this also specifies the integrator to use, the timestep to use, the time for which the simulation is to be run etc. To someone who is familiar with SPH, these are fairly obvious parameters. The scheme may also allow a user to set these parameters via command line arguments. This can be found by simply running:

```
$ pysph run elliptical_drop -h
```

The `post_process` method is also fairly straightforward and is entirely optional. With just this code, one may run the example. As soon as this is done, PySPH will generate high-performance code, compile it, and use that code to run the example.

The scheme in this case is really doing a lot of work because it encapsulates the creation of the equations and the integrators. In order to understand this better, we look at a lower-level implementation of the same example. This example also ships with PySPH and is called `elliptical_drop_no_scheme.py`. Unsurprisingly, this example can be run as:

```
$ pysph run elliptical_drop_no_scheme
```

This implementation does not use a scheme but instead creates the equations and the `Solver` instance directly. The example differs from the `elliptical_drop` in that there is no `create_scheme` method but instead there are two additional methods: - `create_equations` which explicitly creates the equations. - `create_solver` which sets up the solver, stepper and integrators. The `create_particles` and `post_process` etc. are all identical. The code is listed below:

```
def create_equations(self):
    equations = [
        Group(equations=[
          TaitEOS(
              dest='fluid', sources=None,
              rho0=self.ro, c0=self.co, gamma=7.0),
        ], real=False),

        Group(equations=[
            ContinuityEquation(
                dest='fluid', sources=['fluid',]),

            MomentumEquation(
                dest='fluid', sources=['fluid'],
                alpha=self.alpha, beta=0.0,
                c0=self.co),

            XSPHCorrection(dest='fluid',
                           sources=['fluid']),
        ]),
    ]
    return equations
```

As can be seen, the equations are simply instantiated. We look closer at equations further below but at this stage it can be seen that:

- Each equation has a destination `dest` and a list of sources. A destination is a particle on which the acceleration is to be computed a source is one that influences the particle. In this problem there is only one destination and source, `"fluid"`. Note that the names of the arrays are used here to determine the appropriate particle array.
- The `TaitEOS` is an equation of state, i.e. it does not depend on any neighbors and is simply an equation of

the form $p = (\rho - \rho_0)c^2$ or something along those lines. This does not require any "sources".

- Equations can be "grouped" using a `Group`. Each time the acceleration is computed, all equations in a group are evaluated for all the particles before the next group is considered. This is important in the above case as an equation of state is needed to compute the pressure. The pressure must be found for all particles before the other accelerations are evaluated.
- The other equations describe the physics of the problem, namely, continuity and momentum. The `XSPHCorrection` is an SPH-specific correction (see [Mon05]).
- The group containing `TaitEOS` has an additional argument `real=False` this is only used when the example is run via MPI and specifies that the equation of state be computed for all particles local and remote.

```
def create_solver(self):
    kernel = Gaussian(dim=2)

    integrator = EPECIntegrator(fluid=WCSPHStep())

    dt = 5e-6; tf = 0.0076
    solver = Solver(
        kernel=kernel, dim=2, integrator=integrator,
        dt=dt, tf=tf, adaptive_timestep=True,
        cfl=0.3, n_damp=50,
        output_at_times=[0.0008, 0.0038])

    return solver
```

The `create_solver` method simply instantiates a `EPECIntegrator` and asks that the fluid particles be stepped with the `WCSPHStep` stepper. A solver is then constructed which combines the kernel, integrator, and any integration parameters. The scheme automatically creates the equations and solver. Specifying equations directly can be error prone and schemes make this task a lot easier. Schemes also support command line arguments which the direct example would require additional code for.

The only thing that remains is to see how the equations and steppers are actually implemented. Let us consider the continuity equation (6) and see how the `ContinuityEquation` class is implemented.

```
class ContinuityEquation(Equation):
    def initialize(self, d_idx, d_arho):
        d_arho[d_idx] = 0.0

    def loop(self, d_idx, d_arho, s_idx,
             s_m, DWIJ, VIJ):
        vijdotdwij = DWIJ[0]*VIJ[0] + \
            DWIJ[1]*VIJ[1] + DWIJ[2]*VIJ[2]
        d_arho[d_idx] += s_m[s_idx]*vijdotdwij
```

In this class there are two methods:

- `initialize`: this is called first for every destination particle with index `d_idx`.
- `loop`: this is called for every destination source pair. Thus, internally all the nearest neighbors of the destination particle are identified and looped over.

There are some simple conventions followed with the variable names.

- `d_*` indicates a destination array. The name that follows `d_` is the same as the property name of the array.
- `s_*` indicates a source array.
- `d_idx` is a destination index and `s_idx` the source index.
- A method can take any arguments in arbitrary order and these are automatically passed in the right order.

Clearly this seems rather low-level, however, it is simple to write and maps almost exactly with the actual SPH discretized equation (see equation 6).

The integrator and integrator stepper code is similarly quite simple and low level. It is written entirely in pure Python. More details are available in the online PySPH design overview document.

This approach allows a user to specify new equations and integration schemes very easily and use them to perform SPH simulations. The `Application` class also has several other convenient methods that can be overridden by the user to perform a variety of tasks. For example:

- `add_user_options` can be overridden to add any user-defined command line arguments. The argument parsing is done using argparse. Once processed, the options are available in `self.options`.
- `consume_user_options` is used to use any of the parsed options. This is called after the command line arguments are parsed but before the `create_particles` etc.
- `create_domain` can be used to create a periodic domain.
- `configure_scheme` can be used to configure a created scheme based on command line arguments. This is also useful in conjunction with user-defined command line arguments.
- `pre_step`, `post_step`, `post_stage` are convenient methods which will be called before each timestep, after each timestep and after each integration stage if these are defined. These are convenient for a variety of user defined actions including debugging, adaptive refinement, checking for errors etc.

Together, these features are extremely powerful and allow a user a great deal of flexibility.

### *High performance*

While PySPH allows a user to write the code in pure Python, internally, high-performance Cython code is generated, compiled, and used to extract as much performance as possible. This is done using Mako templates. A general Mako template is written to compute the accelerations, this is in `pysph/sph/acceleration_eval_cython.mako`. The main module is `pysph.sph.acceleration_eval` which is implemented in pure Python. A helper class `pysph.sph.acceleration_eval_cython_helper` uses all the high-level information from the user code and provides several methods that are called from the mako template.

The user Python code is already implemented in a low-level allowing us to directly inject the sources into the Cython code. The `pysph.base.cython_generator` module helps with the generation of Cython code from Python code. The `pysph.base.ext_module` takes the generated Cython and compiles this. The extension modules are stored in `~/.pysph/source` in a Python version and architecture specific directory. The md5sum of the Cython code is checked and

if an extension for that md5sum exists the code is not recompiled. Care is taken to look for changes in dependencies of this generated source.

As a result of this, the code performs almost as well as a hand-written FOTRAN code. We have compared running both 2D and 3D problems with the SPHysics serial code. In 2D our code is about 1.5 times slower. This is in part because by default the PySPH implementation is 3D. In 3D, PySPH is about 1.3 times slower. SPHysics symmetrizes the inter-particle computations, i.e. while computing the interaction of a source on a destination, they also compute the opposite force and store it. This appears to provide additional performance gains. Regardless, it is clear that PySPH is comparable in performance with SPHysics. However, PySPH is a lot easier to use and much easier to extend.

PySPH also displays good scale-up with OpenMP. Consider the cube example which considers a cube of a user-defined number of particles (100000 by default), and takes 5 timesteps. One can run `pysph run cube --disable-output` and compare the time taken to run this with `--openmp`. On a quad-core Macbook Pro this produces a speedup of about 4.16. This shows that the scale up is excellent. Good scale up has been observed in the distributed case but is not discussed here.

### *Reproducibility*

The object-oriented API of PySPH makes it easy to extend and use. The design allows for a large amount of code reuse.

We have found that it is extremely important to treat our examples to be as important as the source itself and that these should be shipped with the installation as part of the sources. This forces us to design our examples to be reusable. This is extremely important as:

- it forces a clean API for an end-user. This drives us to minimize repetitive code, and simplify the API.
- the examples are all reusable. If a user wishes to try a new scheme they need to just focus on the new scheme.
- it makes the library easier to use.

While post-processing results, the post-processed data is dumped into a separate file. This makes it trivial to compare the output of different schemes. Some simple tools in `pysph.tools.automation` are provided which make it easy to use PySPH in an automation framework.

Recently, we have used these features to make an entire publication [RP16] completely reproducible. Every figure produced in the paper (a total of 23 in number) is produced with a single driver script making it possible to rerun all the simulations with a single command. This will be described in a future publication. However, it is important to note that PySPH allows for reproducible computation with the SPH method.

## Plans

In the future, the plan is to develop the following features:

- A GPU backend which should allow effective utilization of GPUs with minimal changes to the API.
- Cleanup and potential generalization of the parallel code.
- Implement more SPH schemes.
- Better support for variable $h$.
- Cleanup of many of the current equations implemented.
- Support for implicit SPH schemes and other related particle methods.
- Advanced algorithms for adaptive resolution.

## Conclusions

In this paper a broad overview of the SPH method was provided. The background and context of the PySPH package was discussed. A very high-level description of the PySPH features were provided followed by an overview of the design. From the description it can be seen that PySPH provides a powerful API and allows users to focus on the specifics of the SPH scheme which they are interested in. By abstracting out the high-performance aspects even inexperienced programmers can use the high-level API and produce useful simulations that run quickly and scale well with multiple cores and processors. The paper also discusses how PySPH facilitates reproducible research.

## Acknowledgments

I would like to thank Kunal Puri, Chandrashekhar Kaushik, Pankaj Pandey and the other PySPH developers and contributors for their work on PySPH. I thank the department of aerospace engineering, IIT Bombay for their continued support, excellent academic environment and academic freedom that they have extended to me over the years.

## REFERENCES

[BBC+11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March-April 2011. URL: http://www.cython.org, doi:10.1109/MCSE.2010.118.

[CPR12] J.M. Cherfils, G. Pinon, and E. Rivoalen. JOSEPHINE: A parallel {SPH} code for free-surface flows. *Computer Physics Communications*, 183(7):1468 – 1480, 2012. doi:http://dx.doi.org/10.1016/j.cpc.2012.02.007.

[deva] DualSPHysics developers. Dualsphysics home page. URL: http://www.dual.sphysics.org/.

[devb] SPHysics developers. Sphysics home page. URL: https://wiki.manchester.ac.uk/sphysics/index.php/SPHYSICS_Home_Page.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[GMS01] J.P. Gray, J. J. Monaghan, and R.P. Swift. SPH elastic dynamics. *Computer Methods in Applied Mechanics and Engineering*, 190:6641–6662, 2001.

[Guo14] Philip Guo. Python is now the most popular introductory teaching language at top u.s. universit ies, 2014. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

[Luc77] L. B. Lucy. A numerical approach to testing the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.

[Mon94] J. J. Monaghan. Simulating free surface flows with SPH. *Journal of Computational Physics*, 110:399–406, 1994.

[Mon05] J. J. Monaghan. Smoothed Particle Hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

[Per15] Jefferey M. Perkel. Pickup Python. *Nature*, 518:125–126, February 2015.

[RK10] Prabhu Ramachandran and Chandrashekhar Kaushik. PySPH: A python framework for smoothed particle hydrodynamics. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 16 – 20, 2010.

[RP16] Prabhu Ramachandran and Kunal Puri. Entropically damped artificial compressibility for SPH. *Under review*, 2016.

[RV11] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science and Engineering*, 13(2):40–51, 2011.

[Spr05] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.