

# A Python Based Wrapper for SPIFFE

J. P. Edelen

September 21, 2017

## Abstract

This paper documents the python based wrapper for the SPIFFE code that is maintained by the Advanced Photon Source.

## 1 Introduction

SPIFFE is a 2.5-D electromagnetic particle in cell field solver. The code is open source and developed and maintained by the Advanced Photon Source at Argonne National Laboratory. Currently the code is built to interface with SDDSToolkits and other APS software however due to the broad use of Python in the scientific community for scripting of simulations, optimization, and for data analysis, we have developed a wrapper for SPIFFE that allows for the construction of simulations inside of python functions. This is a convenient way to not only script simulation scans but also to interface with the broad range of optimization algorithms available in python and its associated packages.

In this paper we describe the basic structure of the wrapper and provide sample code to demonstrate its use. The wrapper is available upon request via email to the authors and a published version of the code will hopefully be made available shortly.

## 2 Python wrapper of SPIFFE

In order to facilitate the use of SPIFFE in optimizations using the python toolboxes we constructed a wrapper that can interpret python commands that will build and run a SPIFFE simulation. The only python libraries needed in the wrapper are os and os.path. The initialization command will check to see if the simulation file exists, if it does it will remove the file in order to allow for the creation of a new simulation file. If no input to the initialization is specified the default file will be removed.

```
import os
import os.path
```

```
def initialize(
    simulation_file='simulation.spiffe'):
    if os.path.isfile(simulation_file):
        os.system('rm '+simulation_file)
    return
```

---

Each name-list variable in SPIFFE is represented by a python function whose inputs mirror those expected by SPIFFE. Additionally each function has default values that align with the default values expected by SPIFFE. The specification of default values removes the need to create exceptions for each parameter in the name-list variable. These defaults also make it easier for the user as they do not need to specify each parameter every time. The functions build a string that contains all the information for SPIFFE to properly interpret the simulation block. Once the string is defined it is written to the simulation file that is specified in the initialization. Each function returns the string as well if desired. Note that if the default simulation file is used, no filename is needed. An example of this code is given here.

```
def set_constant_fields(
    simulation_file='simulation.spiffe',
    Ez=0,Er=0,Ephi=0,Bz=0,Br=0,Bphi=0):
    line1 = '&set_constant_fields\n'
    line2 = '\t Ez = '+str(Ez)+'\n'
    line3 = '\t Er = '+str(Er)+'\n'
    line4 = '\t Ephi = '+str(Ephi)+'\n'
    line5 = '\t Bz = '+str(Bz)+'\n'
    line6 = '\t Br = '+str(Br)+'\n'
    line7 = '\t Bphi = '+str(Bphi)+'\n'
    line8 = '&end \n'
    output =
        line1+line2+line3+line4+line5+line6+line7+line8
    fsim = open(simulation_file,'a')
    fsim.write(output)
    fsim.close()
    return output
```

---

After each of the simulation blocks have been defined the user must then run the simulation using the simulate command in the wrapper.

```
def simulate(
```

---

```
simulation_file='simulation.spiffe'):
command = 'spiffe '+simulation_file
os.system(command)
```

Note that if no simulation file is given the default simulation will be run. Next we provide an example of how the simulation is constructed in python. We begin by importing the wrapper then define the simulation parameters to be used in the simulation. This is followed by calling each command that we wish SPIFFE to execute. Finally the simulate command is issued which will run the simulation. This can be collected into a python function that is called as part of a parameter scan or an optimization.

---

```
from subSpiffe import *

## Here the simulation parameters are defined.
nZ = 200
zMin = 0
zMax = 0.1
nR = 200
rMax = 0.1
fn_geo = 'simulation.geo'
dt_integration = 1.1e-13
start_time = 0.0
finish_time = 2.0e-13
status_interval = 100
space_charge = 0

## Initialize the simulation
initialize()

## Make geometry block
define_geometry(nz=nZ,zmin=zMin,zmax=zMax,
nr=nR,rmax=rMax,boundary=fn_geo,
boundary_output=fn_geo+'.bnd')

## Save fields for analysis
define_field_output(filename='simulation.fld',
time_interval=dt_integration,
z_interval=5,r_interval=5)

## The poisson correction is added
poisson_correction(step_interval=1)

## Add external fields (0.1 T magnetic field to
confine the beam)
set_constant_fields(Bz=-0.2)

## Define the emitter
define_emitter(temperature=1000.0,
material_id=2,
number_per_step=10,work_function=2,
electrons_per_macroparticle=1.0e5,
stop_time=1.0e-6)

## Get beam snapshots
```

```
define_snapshots(filename='simulation.snap',
time_interval=dt_integration*100)

## Screen
define_screen(filename='simulation.screen',
z_position=0.01,start_time=2.5e-9)

## The integration block is created...
integrate(dt_integration=dt_integration,
start_time=start_time,finish_time=finish_time,
status_interval=status_interval,
space_charge=space_charge,
lost_particles='simulation.lost')

## The full simulation is defined by the
following line and then written to file.
simulate()
```

---

### 3 Setting up SPIFFE in a python function

Implementing the simulation inside a python function is then a fairly straightforward task. For example if one wanted to study the effects of mesh intervals on an electrostatic simulation it could be constructed inside a python function and then scripted using loops and numpy arrays. The following code snippet shows an example of how one might accomplish this by taking advantage of the the wrapper.

---

```
from subSpiffe import *

# Global parameters
zMin = 0
zMax = 0.1
rMax = 0.1
fn_geo = 'simulation.geo'
dt_integration = 1.1e-13
start_time = 0.0
finish_time = 2.0e-13
status_interval = 100
space_charge = 0

def test_simulation(x):
    ## Here the simulation parameters are
    defined.
    nZ = x[0]
    nR = x[1]
    output_file =
        'simulation'+str(x[0])+'+'+str(x[1])+'.fld'

    ## Initialize the simulation
    initialize()

    ## Make geometry block
    define_geometry(nz=nZ,zmin=zMin,zmax=zMax,
```

```

nr=nR,rmax=rMax,boundary=fn_geo,
boundary_output=fn_geo+'.bnd')

## Save fields for analysis
define_field_output(filename=output_file)

## The poisson correction is added
poisson_correction(step_interval=1)

## The integration block is created...
integrate(dt_integration=dt_integration,
start_time=start_time,finish_time=finish_time,
status_interval=status_interval)

## The full simulation is defined by the
    following line and then written to file.
simulate()

return

# Define mesh parameters for the loop
nZ = numpy.linspace(100,500,10)
nR = numpy.linspace(100,500,10)

# Loop over mesh parameters
for i in range(0,len(nZ)):
    for j in range(0,len(nR)):
        test_simulation([nZ[i,j],nR[i,j]])

```

---

## 4 Conclusions

In this paper we have described the development of a wrapper for SPIFFE in python. This wrapper makes for easy scripting of a fairly well documented and easy to use electromagnetic PIC solver. This provides a platform for using python optimization tools directly with SPIFFE simulations and also creates an easy way to interact with SPIFFE through python for users who are quite familiar with python as opposed to SDDSTools.

## 5 References