Simulation and Visualisation of electron column in the IOTA ring using Paraview

Diletta Milana

under the supervision of

Jayakar Charles Tobin Thangaraj (FAST-IOTA)

and

Chong Shik Park (IOTA)

Giulio Stancari (IOTA)

September 23rd, 2016

Fermi National Accelerator Laboratory, Batavia (IL)

Abstract

During my nine-week internship at Fermilab, I had the opportunity to take part in the IOTA[1] ring experiment, that is currently being developed in the FAST[2] facility. The IOTA ring will be of use for studying some of the instabilities that affect current accelerators, such as the space-charge effect, that is the focus of this work.

In particular, I have focused on the simulation of one space-charge compensation method: the electron column. This method has shown to have potential to improve the performance of circular accelerators.

In the first phase of my internship, I focused on the visualisation of data using Python and Paraview, an open-source, multi-platform data analysis tool for extremely large datasets. Starting from the output of the electron column simulation, I was able to produce a number different visualisations that were very useful in detecting and understanding the most relevant features of the process. These visualisations can be launched by running the relative python script directly from Paraview's GUI: in fact, anyone can potentially make small modifications in the code (setting specific values for the parameters), run the script and see the result.

In the second phase of my internship, I started to focus on how to reduce the bottlenecks in the simulation using HDF5. More work has to be done in the future in this direction.

[1] Integrable Optics Test Accelerator

[2] Fermilab Accelerator Science and Technology

Table of Contents

The context

In this section I will report some key hints concerning the physics behind the electron column simulation.

High-power accelerators and high-brightness beams are needed in many areas of particle physics, such as the study of neutrinos and of rare processes. The performance of these accelerators is limited by tolerable losses, beam halo and instabilities in general, including the **space-charge effect**. Nonlinear integrable optics, self-consistent or compensated dynamics with self fields and beam cooling beyond the present state of the art are being studied to address these issues (G. Stancari† s.d.).

The IOTA ring

The Integrable Optics Test Accelerator (IOTA) is a research storage ring with a circumference of 40 m being built at Fermilab (A. V. S. Nagaitsev s.d.) (D. B.-v. S. Nagaitsev s.d.). Its main purposes are the practical implementation of nonlinear integrable lattices in a real machine and the study of space-charge compensation in rings. IOTA is designed to study single-particle linear and nonlinear dynamics with pencil beams of 150-MeV electrons. For experiments on space-charge dynamics, 2.5-MeV protons will be injected (G. Stancari† s.d.).

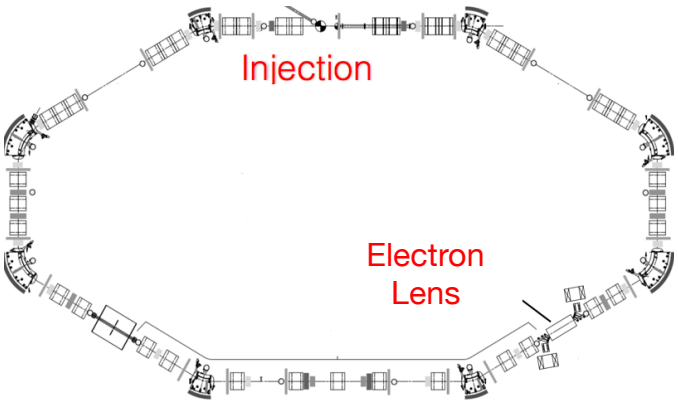


Figure 1: Schematics of the IOTA ring

The space-charge effect

In a beam, particles with the same charge are forced to remain extremely close. The resulting mutual coulomb repulsion creates an internal electric field[3]: this process is called the space-charge effect and, as it was mentioned earlier, it might cause beam losses and emittance growth that limit the current and hence cannot be tolerated. In fact, transverse space-charge effects have long been

[3] In a moving beam, space-charge forces are also partially mitigated by magnetic attraction.

recognized as a fundamental intensity limitation in synchrotrons and storage rings[4] (D. Mohl s.d.). The usual figure of merit of the space-charge effect in circular accelerators is the incoherent tune shift, that is defined as the frequency change of a single particle's betatron oscillations caused by the space charge of a stationary beam. As the RF voltage is turned on in the ring, the injected beam is gradually bunched and experiences the largest space-charge tune shift (W. T. Weng s.d.). If the tune shift is larger than the free space between dangerous (lower order) resonances, then the beam suffers from coherent (e.g., quadrupole breathing modes of the beam envelope) and incoherent (e.g., parametric resonances in a single particle's motion) instabilities (D. Mohl s.d.) (A. V. Burov 2000).

Space-charge compensation

The main idea of space-charge compensation is based on the long-known fact that the negative effect of Coulomb repulsion can be mitigated if beams are made pass through a plasma column of the opposite charge (D. Garbor s.d.). This idea has been successfully applied to transport high-current low- energy proton and H- beams into the RFQ in many linacs[5]. In circular machines, partial neutralization by ionized electrons was attempted with remarkable improvements in proton beam intensity, namely one order of magnitude higher than the space-charge limit. However, the beam-plasma system was subject to strong transverse electron-proton (e-p) instability. In principle, this difficulty can be overcome if protons and electrons are immersed in a strong enough longitudinal magnetic field (Y. A. V. Shiltsev s.d.). Further, the nonlinear optics adopted in the IOTA ring are expected to suppress the e-p instability (V. Dudnikoiv s.d.), and minimize the space-charge driven halo formation (S. Webb s.d.). These synergistic mechanisms can be readily studied by injecting low-energy protons into the IOTA ring. Therefore, the necessary conditions for the effective space-charge compensation in rings will be (G. I. Dimov s.d.), (A. V. Burov 2000) :

1. The impact of electrons is equal to the total impact of beam space charge over the (G. I. Dimov s.d.) ring.
2. The Transverse profile of the electron density $n_e(r)$ is the same as that of the proton beam.
3. The system of electrons and protons is dynamically stable.

Although space-charge compensation is commonly used in linacs, its implementation in rings is still an active field of research. Charge neutralization over the circumference of the ring is

[4] As a side note, the space-charge forces become negligible at high beam energies. So increasing the injection energy, either by making the linac longer or by adding a small booster, would be a straightforward solution, but very costly. In synchrotrons, fast acceleration would help the beam to cross resonances of order 3 and higher, but still there is enough time for lower order modes to develop.

[5] Linear accelerators

usually not practical. Local compensation schemes require high charge densities, which in turn can cause beam scattering, distortions of the lattice, and beam- plasma instabilities.

The electron lens

"Electron lens" is the term used to indicate the area of the ring where space-charge compensation will be applied[6]. In particular, two compensation methods will be investigated, at different times, in the IOTA ring: the electron column and the electron gun The **electron gun** generates the required charge distribution in transverse space and in time, to reproduce the bunch shape of the circulating beam (A. V. Burov 2000). This method is technically challenging, particularly, it requires a large number of expensive lenses for high periodicity machines (M. Chung s.d.). In the other scheme, the so-called **electron column**, the electrons are generated by ionization of the residual gas and trapped axially by electrodes and transversely by the solenoidal field, in a configuration similar to a Penning-Malmberg trap (V. Shiltsev 2007) where the electron gun and collector are not necessary.

The electron column

This method of space-charge compensation aims to achieve very intense and stable beams in circular accelerators through trapping and controlling of electrons generated from beam-induced residual gas ionization. As mentioned earlier, in the electron column, electrons are generated internally through beam-induced residual gas ionization without special electron sources and optics. For this reason, its realization would be more effective and technically feasible (M. Chung s.d.) .

In the following figure, the electron column is shown in detail: a beam of protons is made flow from left to right inside a horizontal pipe. As the protons bump into the H_2 ions contained in the pipe, new electrons are generated: these electrons, that will remained focused towards the center of the pipe due to the solenoid and to the two electrodes, located at the two ends of the pipe, will compensate the space-charge effect.

$$p + H_2 \rightarrow p + H_2^+ + e$$

[6] Because the electron lens is based upon magnetically confined electron beams, some of the space-charge effects can already be mitigated.

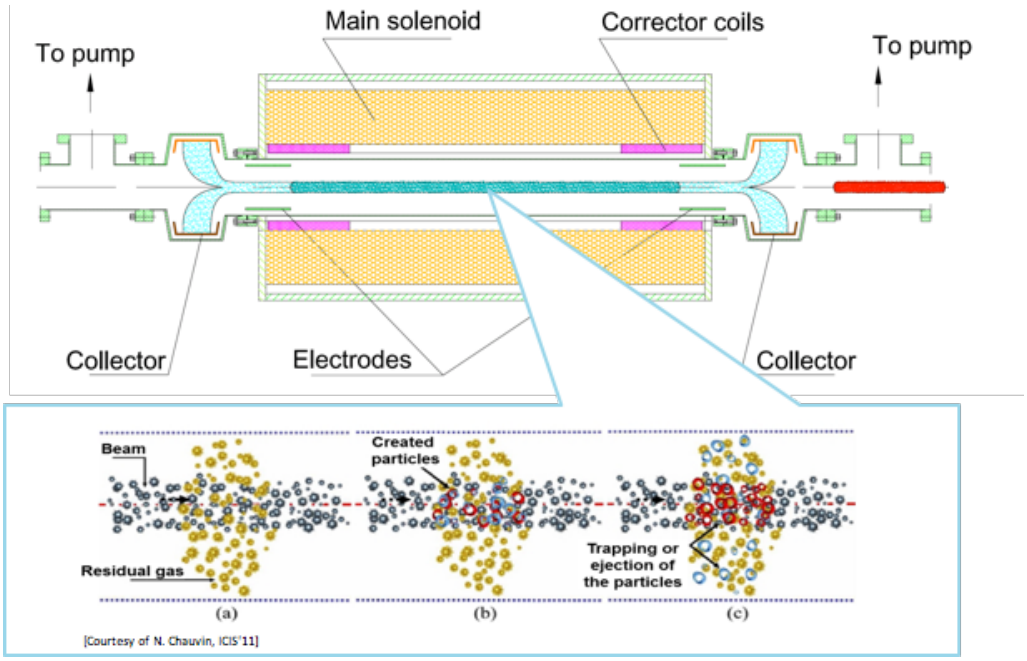


Figure 2: Shiltsev (2007) – Electron Column Proposed Experimental Setup

Installation and setting

Because the IOTA ring is under construction, a simulation of the electron column experiment is being run. To be able to run this simulation script, parallel computation is required. I took advantage of the Accelerator Simulations ***Wilson*** cluster [7] (http://tev.fnal.gov), a joint acquisition by the Accelerator Physics Center, Computing Sector and Technical Division. At the link http://wilsonweb.fnal.gov/cluster/status.html it is possible to monitor in real time the situation of the cluster. To run the simulation, it is necessary to connect to it using ssh, which means that two steps must always be performed[8]:

1. `$ kinit username@FNAL.GOV #to request a Kerberos ticket on the connecting machine. This command will in return ask for the password corresponding to that user`[9]
2. `$ ssh -k username@tev.fnal.gov #to connect to the cluster. Once connected, the current directory will be /home/username`

[7] This cluster is currently being used for development and testing of accelerator and radio frequency simulation codes.

[8] in the following, username will always need to be replaced with the individual username of the active user.

[9] Depending on the local machine, this ticket might not be forwarded to the remote machine once the ssh process starts. This might cause the user to request another Kerberos ticket (with command number 1) once logged in the remote machine, for instance in case he/she needs to push to a remote repository. To understand whether the remote machine already has a ticket or not, it is sufficient to check the output of the command:

`$klist`

After having created a personal directory on the Wilson Cluster (home/username), I proceeded with the build of Warp, an extensively developed open-source particle-in-cell code designed to simulate charged particle beams with high space-charge intensity. Warp takes advantage of a number of different libraries that must be installed as well. To do this, the following steps required, in this order.

At first, I created a "warp" directory, and in that directory the "build" and "install" sub-directories:

```
$ mkdir warp
$ cd warp
$ mkdir build
$ mkdir install
```

All packages are built under "/PATH/TO/BUILD" directory and installed in the "/PATH/TO/INSTALL" directory.

In my case,

```
# /PATH/TO/BUILD   = /home/dmilana/warp/build
# /PATH/TO/INSTALL = /home/dmilana/warp/install
```

Before the installation begins, and also every time a new ssh session is established[10], three environmental variables will need to be sourced. Two of these, PATH and LD_LIBRARY_PATH, can already be set now:

```
$ LD_LIBRARY_PATH="/PATH/TO/INSTALL/lib:$LD_LIBRARY_PATH"
$ export LD_LIBRARY_PATH
$ PATH="/PATH/TO/INSTALL/bin:$PATH"
$ export PATH
```

A config script is suggested to automate this operations at the beginning of each session.

The following steps cover the installation of all the packages needed for the Warp simulation to run.

Python

We will be installing Python-2.7.8. A version 2.7 or greater is required to import "subprocess" module. Python 3.4 is tested, but its syntax is not compatible with Warp.

```
$ cd /PATH/TO/BUILD/
$ wget https://www.python.org/ftp/python/2.7.8/Python-2.7.8.tgz
$ tar zxf Python-2.7.8.tgz
$ cd Python-2.7.8
```

[10] It is suggested to create a single bash script to be launched at the beginning of every session in order to perform all these actions automatically. The one I used will be included in the final documentation.

```
$ ./configure --prefix=/PATH/TO/INSTALL
$ make altinstall prefix=/PATH/TO/INSTALL exec-prefix=/PATH/TO/INSTALL
```

At this point, it is necessary to source the third environment variable, PYTHONPATH

```
$ PYTHONPATH="/PATH/TO/INSTALL/lib/python2.7/site-packages:$PYTHONPATH"
$ export PYTHONPATH
```

Which will be included in the script mentioned above.

Numpy-1.7.1

```
$ cd /PATH/TO/BUILD/
$ wget http://downloads.sourceforge.net/project/numpy/NumPy/1.7.1/numpy-1.7.1.tar.gz
$ tar zxf numpy-1.7.1.tar.gz
$ cd numpy-1.7.1
$ python2.7 setup.py build
$ python2.7 setup.py install --prefix=/PATH/TO/INSTALL
```

Forthon

```
$ cd /PATH/TO/BUILD/
$ git clone https://github.com/dpgrote/Forthon.git
$ cd Forthon
$ python2.7 setup.py install --prefix=/PATH/TO/INSTALL
```

Pygist

```
$ cd /PATH/TO/BUILD/
$ git clone https://bitbucket.org/dpgrote/pygist.git
$ cd pygist
$ python2.7 setup.py config
$ python2.7 setup.py install --prefix=/PATH/TO/INSTALL
```

Warp/Pywarp90

Two alternatives can be used for installation[11].

```
$ cd /PATH/TO/BUILD/
$ git clone https://bitbucket.org/berkeleylab/warp/git/warp.git
$ cd warp/pywarp90
$ git pull
$ cd pywarp90
$ make install
```

[11] The first one is suggested, as it guarantees that the version installed is always up-to-date.

or

```
$ cd /PATH/TO/BUILD/
$ wget
https://bitbucket.org/berkeleylab/warp/downloads/Warp_Release_4.2.tgz
$ tar zxf Warp_Release_4.2.tgz
$ cd warp/
```

Now, a Makefile.local has to be created (for example using vi editor). The following command will create and open up a file in command mode:

```
$ vi Makefile.local
```

At this point, press *i to enter the edit mode. Now, paste the following line

```
PYTHON = python2.7
```

press esc to exit the edit mode and enter the command mode, then :x (will show up at the bottom of the page). Then:

```
$ make install
```
[12]

For the parallel version:

Openmpi13

```
$ cd /PATH/TO/BUILD/
$ wget http://www.open-mpi.org/software/ompi/v1.6/downloads/openmpi-
1.6.5.tar.gz
$ tar zxf openmpi-1.6.5.tar.gz
$ cd openmpi-1.6.5
$ ./configure --prefix=/PATH/TO/INSTALL/
$ make all install
```

PyMPI

```
$ cd /PATH/TO/BUILD/
$ git clone http://portal.nersc.gov/project/warp/git/pyMPI.git
$ cd pyMPI
$ ./configure --prefix=/PATH/TO/INSTALL
$ make
$ make install
```

[12] in case this doesn't work, try:
`make -f Makefile.Forthon install`

[13] Default openmpi version in the Wilson cluster is 1.8.1 but its mpif lib has a different name from old openmpi (libmpi_f77.so -> libmpi_mpifh.so). Fix mpif lib name of warp Makefile.local.pympi or compile openmpi-1.6.5.

Warp/pywarp90

```
$ cd /PATH/TO/BUILD/
$ cd warp/pywarp90
```

Now, create Makefile.local.pympi with:

```
$vi Makefile.local.pympi
```

Now enter edit mode, pressing *i and paste:

```
FARGS = --farg "-DMPIPARALLEL -I/PATH/TO/INSTALL/include -
L/PATH/TO/INSTALL/lib/"
PYTHON = python2.7
```

Now, create setup.local.py with:

```
$vi Makefile.local.pympi
```

Now enter edit mode, pressing *i, and paste (for openmpi-1.6.5):

```
if parallel:
      library_dirs = library_dirs + ['/PATH/TO/INSTALL/lib/']
      libraries = fcompiler.libs + ['mpi', 'mpi_f77']
```

Press esc and then :x to close and save. Now:

```
$ make pinstall
```

Optional Packages:

Scipy

```
$ cd /PATH/TO/BUILD/
$ wget
https://sourceforge.net/projects/scipy/files/scipy/0.14.0/scipy-
0.14.0.tar.gz
$ tar zxf scipy-0.14.0.tar.gz
$ cd scipy-0.14.0
$ python2.7 setup.py install –prefix=/PATH/TO/INSTALL
```

HDF5

```
$ cd /PATH/TO/BUILD/
$ wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-
1.8.17.tar.bz2
$ tar jxf hdf5-1.8.16.tar.bz2
$ cd hdf5-1.8.16
$ ./configure --prefix=/PATH/TO/INSTALL
```

```
$ make
$ make install
```

Pytables

```
$ cd /PATH/TO/BUILD
$ wget
https://sourceforge.net/projects/pytables/files/pytables/2.1.2/table
s-2.1.2.tar.gz
$ tar zxf tables-2.1.2.tar.gz
$ cd tables-2.1.2
$ python2.7 setup.py build --hdf5=/PATH/TO/INSTALL
$ python2.7 setup.py install --prefix=/PATH/TO/INSTALL –
hdf5=/PATH/TO/INSTALL
```

Now that the installation is complete, to make sure it works properly, here are some test steps to perform:

```
$ cd /PATH/TO/BUILD/
$ cd warp/warp_test
$ python2.7 runalltests.py
```

If no errors are returned, the installation was successful.

Running the simulation

To run the electron column simulation, two files are necessary:

1. **ecolumn.py** : A python file that contains the electron column simulation script written using Warp libraries by Moses Chung and Chong Shik Park. For the data visualisation and analysis part of this report, this simulation script will be taken for granted and not discussed any further.
2. **ecolumn.sh**: A bash file that launches the simulation on the Wilson Cluster.

Simple bash file

This is an example of the most simple bash file used to run the simulation: ecolumn.sh

```
#!/bin/sh
#PBS -o ecolumn.out #file produced when the simulation is over: this
is the standard output used throughout the code to print relevant
moments in the simulation.
#PBS -e ecolumn.err #file produced when the simulation is over:
contains traces of the errors occurred.
```

```
#PBS -l nodes=1:amd32 #runs on 1 node of type amd32
#PBS -l walltime=24:00:00 #maximum time allowed for a simulation to
run
#PBS -q amd32 #queue where the job will wait for its turn
#PBS -A srflinac #account on which the job time will be charged.

nproc=32          #number of processors on which the simulation runs.
This must be equal to procspernode(see next line) times nodes (see
above).
procspernode=32 #amd32 nodes have 32 processors, while intel12 have
12 processors per node.

echo "running on $nproc processors"

. /home/username/warp/setup.sh #performs the sourcing mentioned in
the previous pages

cd ${PBS_O_WORKDIR} #moves to the right directory on cluster

mpirun -np ${nproc} -npernode ${procspernode} pyMPI ecolumn.py -p 4
4 64
#launches the simulation with the parameters set above.
```

Once the parameters in the bash script have been properly set, it is possible to launch the simulation with the command

```
$ qsub ecolumn.sh
```

To check the status of the simulation:

```
$ qstat
```

To interrupt and delete a job:

```
$qdel jobID #the ID of the job can be seen using qstat command
```

Once the simulation is over, it will produce all the results in the same directory where the scripts are located.

Alternative bash file

I was able to create many different alternative bash files starting from the previous simple script. For instance, the following script runs on intel12 nodes (more efficient for input-output operations) which have 12 processors per node. The intel12_cbhat queue is a high priority queue, that also allows for a walltime[14] of 60 hours instead of 24.

What characterises this script is the fact that 3 arrays are created, each of which contains all the values for a three parameters (the magnetic field produced by the solenoid, the potential at the right electrode and the potential at the left electrode) that are of interest in a simulation.

```
#!/bin/sh
#PBS -o ecolumn.out
```

[14] time after which the job will be aborted automatically if it has not finished yet.

```
#PBS -e ecolumn.err
#PBS -l nodes=2:intel12 #running on intel12 nodes
#PBS -l walltime=60:00:00
#PBS -q intel12_cbhat #on a priority queue
#PBS -A srflinac

# Absolute value function, used to create the name of files and directories
containing the results of a specific simulation
 abs() {
   [ $1 -lt 0 ] && echo $((-$1)) || echo $1
 }

nproc=24

procspernode=12
echo "running on $nproc processors"

. /home/dmilana/warp/setup.sh

cd ${PBS_O_WORKDIR}

#creating three arrays of parameters. Each array contains all values to be
assigned to a parameter, such as Bsol (the magnetic field produced by the
solenoid), Vbias_left (potential at the left electrode) and Vbias_right.
Hence, simulation number x will be run with parameters coming from Bsol[x],
Vbias_left[x] and Vbias_right[x].

declare -a Bsol=(0 0.1 0.1 0.5 0.5 )
declare -a Vbias_left=(0 -50 -100 -50 -100 )
declare -a Vbias_right=(0 -50 -100 -50 -100)

param_length=${#Bsol[@]} #all three arrays will have the same length
echo $param_length
max=$((param_length-1))
echo $max

i="0"

while [ $i -lt $param_length ]; do
        b=${Bsol[$i]}

        vl=${Vbias_left[$i]}
        vr=${Vbias_right[$i]}

       #this value will be used later in the naming of directories
       (instead of calling a directory V_-10 we will call it V_10)
        absl=$(abs $vl)
        absr=$(abs $vr)

         #Now we use the sed command to substitute in the original
         ecolumn.py file, the 3 values just extracted from the arrays,
         creating a temporary file for each substitution and eventually a
         new complete simulation file, ecolumn_parameters.py. The
         ecolumn_parameters.py file will be placed in a new directory,
         created explicitly for that simulation and whose name will
         indicate clearly the parameters of that specific simulation.

        sed "s/\"bsol_value\"/$b/" ecolumn.py >tmp1.py
        sed "s/\"vbiasl_value\"/$vl/" tmp1.py >tmp2.py
        sed "s/\"vbiasr_value\"/$vr/" tmp2.py >ecolumn_parameters.py
```

```
          #here a new directory for that specific simulation is created and
          the ecolumn_parameters.py files is copied into it. After moving
          into that directory (cd) move into that directory, the job itself
          will be launched.

        directory="./ecolumn/B_"$b"T_V_"$absl"_"$absr"V"
        mkdir $directory
        cp ./ecolumn_parameters.py $directory
        #echo $directory
        cd $directory
        mpirun -np ${nproc} -npernode ${procspernode} pyMPI
ecolumn_parameters.py -p 4 4 64
        let i=$i+1
done
```

Once the simulation script is launched, it will start to produce some results: the output data can be analyzed. For this purpose, I used the Paraview software.

Paraview

About Paraview

ParaView is an open-source, multi-platform data analysis and visualization tool. ParaView users can build visualizations to analyze their data using qualitative and quantitative techniques. ParaView is able to visualise and animate data from a very high number of different file formats (txt, csv, vtk, hdf5…). ParaView was developed to analyze extremely large datasets using distributed memory computing resources (K. Moreland s.d.).

Why Paraview

Amongst the various software applications available, we chose Paraview for two main reasons:

- Strong Python support: users can create python scripts based on the `paraview.simple` library. These portable scripts can then be used by other users and on different datasets with different parameters (can be modified by modifying just one lines of code). All a third user needs to do to create his visualisations basing on someone else's script is work directly on the script (modifying for instance the lines where the simulation files are imported), import it in Paraview and run it to see the results.
- Strong animation support: with some hints[15], Paraview is able to understand how the data in input is structured and in particular whether a group of files located in one directory represents a unique set of data, yet sampled at different timesteps. This way, Paraview can easily create interesting animations, that allow to understand the basics of the process behind them.

[15] see next sections

How ParaView works

Visualization is the process of converting raw data into images and renderings to gain a better cognitive understanding of the data. ParaView uses VTK, the Visualization Toolkit, to provide the backbone for visualization and data processing.

Paraview's core is the so-called visualization pipeline: you bring your data into the system by creating a **reader**, the source, which depends on the file format at hand. You then apply a **filter** to either extract information (e.g., iso-contours) and render the results in a view or to save the data to disk using writers. All readers can be found in Filters, Search. Eventually, you create one or more views to **render** the data. The key point in creating the right visualisation for the specific case is, hence, finding the right filter, amongst the high number of filters available.

These three steps can be performed by a python script, that allows portability and re-usability. In the next pages, the main scripts I developed will be shown and commented.

Install ParaView

To obtain Paraview it is sufficient to visit the webpage http://www.paraview.org/download/ and download the version corresponding to the current system. To install the software, just follow the steps of the guided procedure.

Support

Paraview has an extensive support: a lot of information can be found on the web in terms of official PDF documents[16] or mailing lists[17].

Reset the current session

Before moving from one visualisation to another, it is important to reset the current session, so as to avoid any overlapping between existing readers and filters. All the scripts I developed do this automatically, as they import the `reset.py` module, that contains the `reset_all()` function that will reset the current session[18]:

```
import sys
import os
PATH_TO_SCRIPTS="/Users/diletta/Desktop/Fermiworking/paraview/"
sys.path.append(os.path.abspath(PATH_TO_SCRIPTS))
#to reset all components in the current view
```

[16] See bibliography

[17] http://www.paraview.org/mailing-lists/ . Even without subscribing to the official mailing list, a simple google.com search will likely produce very useful results.

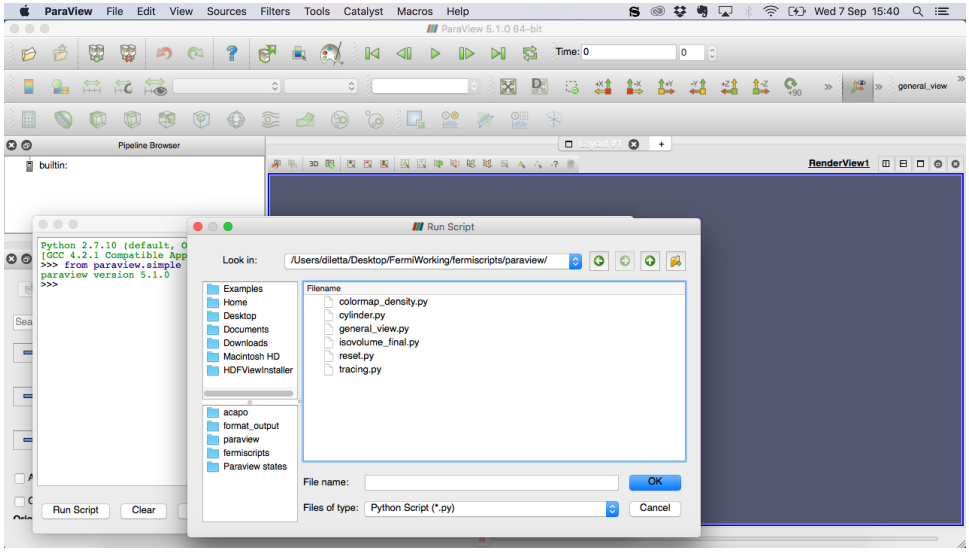
[18] For this reason it is very important that all scripts (including reset.py) are all kept in the directory indicated by the variable PATH_TO_SCRIPTS which is located at the beginning of each script. Otherwise, the import will not work

```
from reset import *
reset_all()
```

In any other case, to reset the session just click on **Edit, Reset Session**.

Run a script

It may be very useful to start from a script for Paraview that some other user has wrote. To do this, it is sufficient to click on Tools, then select Python Shell. This will open up a shell. By clicking on the Run Script button, It will be possible to select which script to run.



Hence, it is very easy to start from a script that somebody else has written and modify it according to the specific needs.

Save State

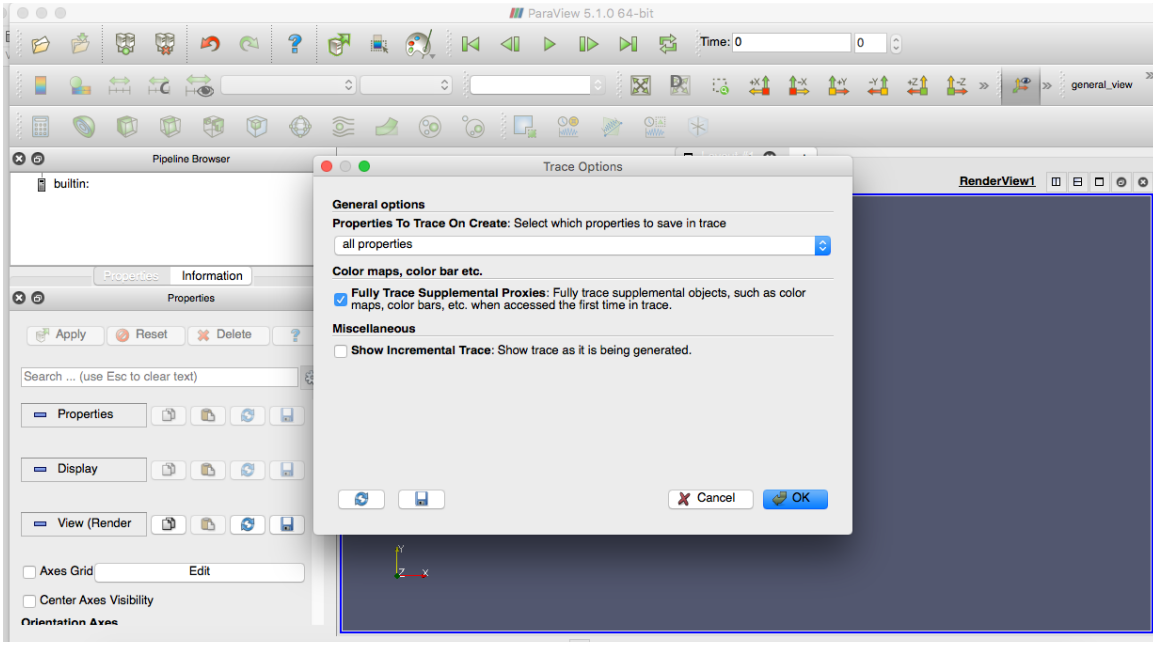
In Paraview there is no such thing as simply "save" to update an exisiting file. One of the most simple way to save the progress in Paraview is to go File, Save State. This will generate a .pvsm file that can be opened uniquely by Paraview and the reproduces completely the current situation. It is advisable to load this .pvsm file directly within Paraview (File, Load State) because most likely, simply clicking on the file and trying to open it in Paraview will cause an error[19].

Start and Stop Trace

Even though Paraview is very well documented, it might be hard to find the exact command corresponding to a desired action. This happens very often when it comes to display properties, as there are a very high number of them, depending on the active filter or view.

[19] On Mac: "no application found to open the document".

For this reason, the **Tools, Start Trace** function might be very useful. After having selected the right settings the pop-up window:



Paraview will start converting the actions performed in the GUI into python commands. To stop the trace, it is sufficient to click on Tools, Stop Trace. At this point, Paraview will visualise the script corresponding to the actions performed. The user can now decide to save the script, copy its content or close it and hence delete it permanently.

Format the output

The initial results I displayed with Paraview did not make any physical sense. In my case, with a detailed analysis I found out that these files contained some small typos: small formatting incoherencies (blank spaces, new lines), which resulted in the impossibility for Paraview to really understand the data. Since the time and resources required for the complete simulation to run again were challenging, I decided to write some simple python script that would replace the typos.

The following script replaces double spaces:

```
#!/usr/bin/python
name_temp_old="/Users/diletta/Desktop/FermiWorking/B_0.1T_V_0.005kV/";
name_temp_new="/Users/diletta/Desktop/format_output/B_0.1T_V_0.005kV/";

TOP_RANGE= 114500

#protons
part="protons_"
for i in range(0, TOP_RANGE):
```

```
      if (i==0 or i%500==0):
            count= "%06d" % (i,)
            with open(name_temp_old+part+count+".txt", "r") as f:
                  content = f.readlines()
                  newfile= open(name_temp_new+part+count+".txt", 'w+')
                  for line in content:
                        newline= line.replace("  ", ",").replace(" ", ",")

#electrons
part="electrons_"
for i in range(0, TOP_RANGE):
        if (i==0 or i%500==0): #ricordarsi i due punti in if e for
                count= "%06d" % (i,)
                with open(name_temp_old+part+count+".txt", "r") as f:
                        content = f.readlines()
                newfile= open(name_temp_new+part+count+".txt", 'w+')
                for line in content:
                        newline= line.replace("  ", ",").replace(" ", ",")

#h2plus
part="h2plus_"
for i in range(0, TOP_RANGE):
       if (i==0 or i%500==0):
                count= "%06d" % (i,)
                with open(name_temp_old+part+count+".txt", "r") as f:
                        content = f.readlines()
                newfile= open(name_temp_new+part+count+".txt", 'w+')
                for line in content:
                        newline= line.replace("  ", ",").replace(" ", “,")
```

The following script replaces blank lines:

```
#!/usr/bin/python

name_temp_old="/Users/diletta/Desktop/format_output/";
name_temp_new="/Users/diletta/Desktop/format_output/acapo/";

TOP_RANGE= 114500
part="electrons_"
```

```
for i in range(0, TOP_RANGE):
  if (i==0 or i%500==0):
    count= "%06d" % (i,)
    newfile= open(name_temp_new+part+count+".txt", 'w+')
    with open(name_temp_old+part+count+".txt", "r") as f:
      for line in f:
        if line.strip("\n"):
          newfile.write(line)

part="protons_"

for i in range(0, TOP_RANGE):
  if (i==0 or i%500==0):
    count= "%06d" % (i,)
    newfile= open(name_temp_new+part+count+".txt", 'w+')
    with open(name_temp_old+part+count+".txt", "r") as f:
      for line in f:
        if line.strip("\n"):
          newfile.write(line)

part="h2plus_"

for i in range(0, TOP_RANGE):
  if (i==0 or i%500==0):
    count= "%06d" % (i,)
    newfile= open(name_temp_new+part+count+".txt", 'w+')
    with open(name_temp_old+part+count+".txt", "r") as f:
      for line in f:
        if line.strip("\n"):
          newfile.write(line)
```

To be able to color-map the electrons by density, Paraview would need to have the same density field in the same file as the particle's location[20]. The current simulation output instead, had different files: for each time step, one file contains all densities and one file contains only the particles information. In particular, the density files can be divided in two groups: denx, carrying the density value along the x axis for all protons, electrons and h2plus, and den

[20] A programmable filter could have been considered for the same purpose, but it would have been computationally intensive anyway.

carrying the density value along the z axis. I wrote the following script to include a z-density field in the same electron[21] files:

```
#!/usr/bin/python
name_el_old="/Users/diletta/Desktop/format_output/acapo/";
name_den_old="/Users/diletta/Desktop/format_output/";
name_el_new="/Users/diletta/Desktop/format_output/new_density_field/";

TOP_RANGE= 114500
part="electrons_"
for i in range(0, TOP_RANGE):
  if (i==0 or i%500==0):
    count= "%06d" % (i,)
    print("file: "+count)

  with open(name_el_old+part+count+".txt", "r") as f: #old electron files
    content_el = f.readlines()

  with open(name_den_old+"_new_denz_"+count+".txt", "r") as densityfile:
    content_den=densityfile.readlines()

  #new electron file
  newfile= open(name_el_new+"new_"+part+count+".txt", 'w+')

  #for every line in the old electron file we pick the z coordinate
  for line in content_el:
    split= line.split(",")

    #we approximate the z coordinate by 2 digits
    coordz=round(float(split[3]), 2)
    #for every electron, we look up its density in the relative file
    for line1 in content_den:

      split1= line1.split(",")
      coordz1= round(float(split1[2]),2)
      den=float(split1[1])
      #densities are sampled by 0.001 at each step
```

[21] Since the operation was very computationally intensive per se, only electrons underwent this formatting (as they are the particles whose density is more interesting).

```
      if (float(abs(coordz1 -coordz))<0.01):

        newline=line.replace("\n","")+","+str(den)
        newfile.write(newline+"\n")
        break
```

The following script was developed to reduce the number of particles in each file, to make it easier to track them:

```
#!/usr/bin/python
name_temp_old="/Users/diletta/Desktop/format_output/";
name_temp_new="/Users/diletta/Desktop/format_output_particles/";
TOP_RANGE= 114500

#electrons
part="electrons_"
for i in range(0, TOP_RANGE):
        if (i==0 or i%500==0): #ricordarsi i due punti in if e for
                count= "%06d" % (i,)
                with open(name_temp_old+part+count+".txt", "r") as f:
                        content = f.readlines()
                newfile= open(name_temp_new+"mid_lines_"+part+count+".txt",
"w+")

                i=0
                j=0

                with open(name_temp_old+part+count+".txt", "r") as f:
                    content = f.readlines()

                for line in content:
                    if (i>301): #cosi non inizio con spazio
                        newfile.write(line)
                        j=j+1

                    if (j>7): #perche' le righe sono intervallate da spazi,
                              quindi per scrivere 3 righe ne devo leggere 6
                              (di cui 3 di spazi)
                        break
                    i=i+1
```

The main Paraview scripts

I developed five main scripts:

1. General View
2. Particle Density
3. Particle Tracing
4. Slicing
5. Compared View

In this document, only the main features of the code will be analysed[22]. First, however, the common settings within each script will be discussed.

General Settings and Annotations

It is worth to note that these scripts contain some replicated code: I chose to include the general settings and annotations in each script, instead of creating a single settings.py script to be imported by all other scripts (general_view.py, tracing.py…), because I think that it is most likely for a user to be willing to personalise its visualisations in different ways. However, in case the user believes that it might be useful to create a single script to be imported in all the others (and thus have the same settings in each visualisation), all he/she will need to do is:

1. Look for the settings-related lines of code in each python script. They can be found at the bottom of each script.

2. Copy these lines and move them into a newly created settings.py file. Inside this file, locate these lines in a dedicate function.

3. Save the file in the location saved in the PATH_TO_SCRIPTS variable. This variable is present at the beginning of every script, and it is currently only used to import the reset.py module (that resets the previous session before creating the new visualisation).

```
import sys
import os
#indicates where all imported scripts can be found
PATH_TO_SCRIPTS="/Users/diletta/ Fermiworking/fermiscripts/paraview/"
sys.path.append(os.path.abspath(PATH_TO_SCRIPTS))
```

4. Edit the new settings.py code as you prefer.

[22] Its is important to note that, since the **Start Trace** functionality was used in particular cases as a base on which to develop the traditional code-writing procedure, there might be some replicated code within the script itself. For instance, one property of a certain object might be assigned two different values in two different positions of the script. I have extensively reviewed the scripts to remove these inconveniences, but some cases might still exist.

5. Remove all settings-related and annotations-related lines of code from the scripts that you want to have common settings (in the settings.py file): instead simply call the function you created in the settings.py file that will generate them.

6. Make sure you include the `import settings` line at the beginning of each script.

Here are the most relevant settings lines, currently included in every script.

```
# set to 0 to hide the AxisGrid, 1 to show it
renderView1.AxesGrid.Visibility = 0


# set to 0 to hide OrientationAxes, 1 to show it
renderView1.OrientationAxesVisibility = 0

# set to 1 to show a gradient background, 0 otherwise
renderView1.UseGradientBackground = 0

# select the RGB colours for the background
renderView1.Background = [0.0, 0.0, 0.0]

#create a textual annotation
titleText = Text()
titleTextDisplay = Show(titleText, renderView1)
titleText.Text = 'Electron Column'
titleTextDisplay.FontSize = 16
titleTextDisplay.WindowLocation = 'AnyLocation'
titleTextDisplay.Position = [0.67, 0.884200]
renderView1 = FindViewOrCreate('RenderView1', viewtype='RenderView')


# place the view on the e-column. It is a bit cumbersome to manipulate the
view in Paraview: the slightest touch of the mouse might result in a big
change in the view. This is why I found it very useful to play a little bit
with the camera, find my preferred view and then save its coordinates[23].
renderView1.CameraPosition = [-0.13070705617372036, 0.27772719478767843, -
1.5591315229262686]
renderView1.CameraFocalPoint = [5.511098962943289e-17, 1.3644275479349438e-
16, 0.5500000059604674]
renderView1.CameraViewUp = [0.9970757747344116, -0.03729353056638097, -
0.06670151434313464]
renderView1.CameraParallelScale = 0.09921621550177633
renderView1.CameraParallelProjection = 1
```

[23] see Start Trace section

The Pipe

```
# create a 'Cylinder' representing the e-column pipe. Initially, it will
have default properties.
cylinder1 = Cylinder()
cylinder1.Resolution = 350
cylinder1.Height = 1.1
cylinder1.Radius = 0.03
cylinder1.Center = [0.0, 0.0, 0.0]
cylinder1.Capping = 1
RenameSource('Pipe', cylinder1)

 # in order to rotate the pipe and make it match the flow of particles we
will need to transform the pipe, using a Transform filter.
transform1 = Transform(Input=cylinder1)
transform1.Transform = 'Transform'
transform1.Transform.Translate = [0.0, 0.0, 0.55]
transform1.Transform.Rotate = [90.0, 0.0, 90.0]
transform1.Transform.Scale = [1.0, 1.0, 1.0]
```

1. General View

See the full tutorial at https://www.youtube.com/watch?v=TKH8VKp9zMw.

In this section we will show the main features of the python script that produces a general view of the e-column, where it is possible to see the behaviour of protons, electrons and h2plus.

```
#import source files
electrons_ = CSVReader(FileName=glob.glob(output_path+"electrons_*"))

#describe the properties of the files imported: there are no headers on top
of the columns, and each column is separated by the others by a comma.
electrons_.DetectNumericColumns = 1
electrons_.UseStringDelimiter = 1
electrons_.HaveHeaders = 1
electrons_.FieldDelimiterCharacters = ','
electrons_.MergeConsecutiveDelimiters = 0

# create a new 'Table To Points filter' to extract points from source files
tableToPoints1 = TableToPoints(Input=electrons_)
RenameSource('ElectronPoints', tableToPoints1)
tableToPoints1.XColumn = 'Field 1' #cannot place ElectronPoints here,
renaming happens only in Paraview environment
tableToPoints1.YColumn = 'Field 2'
```

```
tableToPoints1.ZColumn = 'Field 3'
tableToPoints1.a2DPoints = 0
tableToPoints1.KeepAllDataArrays = 1

# create a new 'Clip', that will cut out the "noise" points that we are not
interested in, for example those that are left before z-axis 0.
clip1 = Clip(Input=tableToPoints1)
clip1.ClipType = 'Scalar'
clip1.Scalars = ['POINTS', 'Field 3']
clip1.Value = 0.0
RenameSource('ClipElectrons', clip1)
```

2. *Particle Density*

In this section we will show the main features of the python script that produces a view of the particles, coloured by their density. Thus, a density field in the point in space where that specific particle is located at that specific moment in time, has to be added to files containing information on the single particles, it is not sufficient to have separate density and particle files. This means that in addition to the existing attributes (x-coordinate, y-coordinate, z-coordinate…), for every particle ID, also the density along the z-axis will be available in the same file.

In order to do so, since I initially did not have access to the simulation code itself and I had to take it as a black box, I just "merged" the particles and density files. This operation was computationally demanding, so it was tested only on electrons, but would be conceptually perfectly adaptable to all other particles as well:

```
#!/usr/bin/python
name_el_old="/Users/diletta/Desktop/format_output/acapo/"
name_den_old="/Users/diletta/Desktop/format_output/"
name_el_new="/Users/diletta/Desktop/format_output/new_density_field_0913/"

TOP_RANGE= 114500

#electrons
part="electrons_"
for i in range(0, TOP_RANGE):

  if (i==0 or i%500==0):
    count= "%06d" % (i,)
    print("file: "+count)

  #opening original electron files
```

```
  with open(name_el_old+part+count+".txt", "r") as f:
    content_el = f.readlines()

  #opening original density files
  with open(name_den_old+"denz_"+count+".txt", "r") as densityfile:
    content_den=densityfile.readlines()

  #opening new electron files, that will include density fields
  newfile= open(name_el_new+"new_"+part+count+".txt", 'w+')
  for line in content_el: #for every “old” electron
    split= line.split(",")
    coordz=round(float(split[3]), 2) #pick z-coordinate and round

    for line1 in content_den: #for every z-coord sampled in density files
      split1= line1.split(",")
      coordz1= round(float(split1[0]),2) #pick the z-coordinate and round
      den=float(split1[2]) #pick the density corresponding to that z coord
      if (float(abs(coordz1 -coordz))<0.01): #if the density picked is the
      that of a point on the z-axis that is close enough to the location of
      the electron chosen in the outer for loop
        newline=line.replace("\n","")+","+str(den)
        newfile.write(newline+”\n") #write the line with the new field
        break
```

An alternative to this clumsy method would be to add the density directly within the simulation code. This feature has been included in the latest ecolumn.py simulation script but has not been tested yet.

Whatever the method, once the new files are ready to be used as input into Paraview:

```
#set the color of the clip depending on the value of the density (field 5):
ColorBy(clip1Display, ('POINTS', 'Field 5’))

# get color transfer function/color map for 'Field5'
field5LUT = GetColorTransferFunction(‘Field5’)

# Rescale transfer function
field5LUT.RescaleTransferFunction(0.0, 7.017172e+14)

# create color legend/bar for field5LUT. A number of properties are listed
in the script, but they din’t have particular relevance so they will not be
listed here.
```

```
field5LUTColorBar = GetScalarBar(field5LUT, renderView1)
```

3. Particle tracing

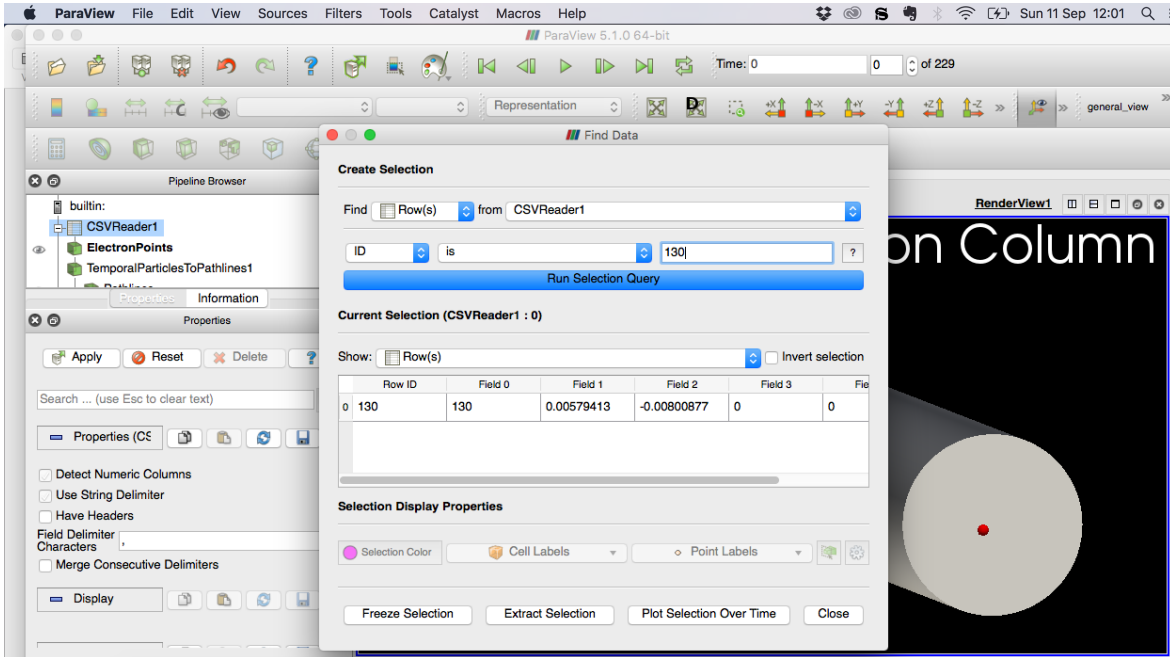
In the following script, it is shown how to trace the path of a single particle in time. The temporalParticlesToPathlines filter does not allow to select the specific particle to trace. Instead, it only allows one to pick one particle out of *many* others (this *many* depends on the MASK_POINTS variable in the next lines).

```
MASK_POINTS=10000 #indicates the sampling setting: mask_points=10000 means
only 1 particle out of 10000 will be displayed


PATH_LENGTH= 1000 #indicates the maximum length of the path of a particle.
If the path_length is too short, only the latest steps will be shown. If it
is very long, like in this case, the complet path will be shown


PARTICLE_RADIUS= 0.002 #indicates the radius of the particle displayed


# create a new 'Temporal Particles To Pathlines'
temporalParticlesToPathlines2 = TemporalParticlesToPathlines (Input=
tableToPoints2, Selection=None)
temporalParticlesToPathlines2.MaskPoints = MASK_POINTS
temporalParticlesToPathlines2.MaxTrackLength = PATH_LENGTH
temporalParticlesToPathlines2.MaxStepDistance = [1.0, 1.0, 1.0]
temporalParticlesToPathlines2.IdChannelArray = 'Global or Local IDs'


temporalParticlesToPathlines3Display =
Show(OutputPort(temporalParticlesToPathlines3, 1), renderView1)
temporalParticlesToPathlines2Display.GlyphType = 'Sphere'
temporalParticlesToPathlines2Display.GlyphType.Radius = PARTICLE_RADIUS
```

To select the single particle by its ID, it is necessary to use the GUI[24]: see a tutorial on how to extract a single particle on https://www.youtube.com/watch?v=bdLXmkev9fI . What we need to do is go **Edit, Find Data.** In the new pop-up window, it is sufficient to insert the chosen ID and press enter. At this point, the result of the query will be displayed (as shown in Figure 3). Pressing the **Extract Selection** field will create a new Filter (Extract Selection filter).

[24] There seems to be a renowned bug in ParaView concerning how to select a particle by its ID in a python script. Hence, for the time being, it will be shown only in GUI.

At this point, it is possible to click on the new filter (Extract Selection) and apply a TableToPoints filter, just as we we did with input files[25]. Now, a new TemporalParticlesToPathlines can be applied .



4. *Slicing*

```
Threshold_min= 0.54 #where the slice begins when z=0
Threshold_max=0.58 #where the slice ends when z=1

# create a new iso volume
isoVolume1 = IsoVolume(Input=tableToPoints1)
isoVolume1.InputScalars = ['POINTS', 'Field 3']
isoVolume1.ThresholdRange = [Threshold_min, Threshold_max]

# create the pipe
cylinder1 = Cylinder()
RenameSource('Pipe', cylinder1)
pipe = FindSource('Pipe')
SetActiveSource(pipe)

# Properties modified on pipe: the height of the pipe are set depending on
the dimension of the slice, whose parameters have been set at the
beginning. However, the pipe will be kept a big longer than the slice (by
0.01)
```

[25] With the extract selection filter, all we are doing is basically reading all files and taking one line in each. This is why we still need a Table To Points filter, as the type of the input (a line in the input file) is still the same.

```
pipe.Height = Threshold_max - Threshold_min +0.01
```

5. Compared view

A script producing a compared view of different simulation outputs (for instance comparing the results of a simulation running with a magnetic field of 0.1T with one with a magnetic field of 0.5T) is also included. The code of this script is not commented in this document as it does not bring anything new to the discussion.

HDF5

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data[26]. This format becomes quite handy when dealing with simulations that need to manage huge amounts of data and continuously write onto files.

HDF5 files are basically made up of two components:

- **HDF5 group:** a grouping structure containing zero or more HDF5 objects, together with supporting metadata. Working with groups is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, an object in an HDF5 file is often referred to by its full path name (also called an absolute path name). For instance, "/foo" signifies a member of the root group called foo (D. B. Michelson s.d.).
- **HDF5 dataset:** a multidimensional array of data elements, together with supporting metadata
- **HDF5 attribute** is a user-defined HDF5 structure that provides extra information about an HDF5 object. Any HDF5 group or dataset may have an associated attribute list.

To view the structure and the data contained in an HDF5 file, a visual tool like HDF5View can be very useful[27]. Another tool to keep in mind is the h5dump command, that can be used with the following options[28]:

```
-H #displays header information only (no data)
-n #displays the content (list of objects) in a file
```

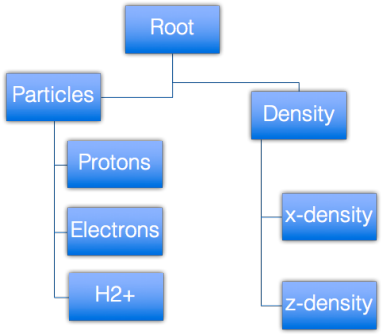
Example: `h5dump -H timestep_000000.h5`

[26] borrowed from https://www.hdfgroup.org/HDF5/

[27] https://www.hdfgroup.org/products/java/hdfview/

[28] https://www.hdfgroup.org/HDF5/Tutor/cmdtoolview.html

A possible structure would be something like the one displayed on the right.



before HDF5: one file for each type of particle for each timestep, and one file for each type of density for each timestep.

after HDF5: one file for each tilmestep containing all information on particles, and another file all densities (for all particles, along the x and z axis).

There are a number of different libraries that can be used to read and write HDF5 files. PyTables, for instance, is wrote on top of the HDF5 library, using the Python language and the NumPy package[29].

In the case of the electron-column simulation, since it was written using Warp library to be able to exploit parallel computation, pytables could not be used. Instead, PWpyt and PRpyt modules, explicitly developed to manage HDF5 in Warp, were used[30]. However, I encountered some bugs in these scripts (it wasn't possible to use the `write` function, for instance), so the script itself required a lot of testing. Eventually, once the files got written, I wasn't able to close them without causing errors and thus interrupting the simulation. Hence, more work has to be done in this direction to find out the reasons behind these errors and possibly find better ways to manage HDF5 and Warp.

The resulting ecolumn.py script included these writing lines:

```
if (iter % save_repetition == 0):
    h5file= PWpyt.PW("timestep_%06d.h5" % iter)
    ###PROTONS
    nions = ions.getn()
    xions = ions.getx()
    yions = ions.gety()
    zions = ions.getz()
    pidions = ions.getpid()

    ions_r = []
    for i in range(0, nions):
```

[29] http://www.pytables.org

[30] PWpyt.py module to write HDF5 files:
http://hifweb.lbl.gov/Warp/scripts/doc/web/html/Warp/scripts/doc/PWpyt.html
PRpyt.py module to read HDF5 files:
http://hifweb.lbl.gov/Warp/scripts/doc/web/html/Warp/scripts/doc/PRpyt.html

```
            denx=(iden[xions[i],iycenter,izcenter]+
                  iden[xions[i],iycenter,izcenter-1]+
                  iden[xions[i],iycenter,izcenter+1])/3.0
            denz=(iden[ixcenter,ixcenter,zions[i]]+
                  iden[ixcenter,iycenter,zions[i]-1]+
                  iden[ixcenter,iycenter, zions[i]+1])/3.0

            ions_r.append([i,xions[i],yions[i],zions[i],pidions[i],
            denx,denz])

        ions_np= np.array(ions_r)
        h5file.protons= ions_np
```

And same for h2plus and electrons.

HDF5 and Paraview

As mentioned earlier, HDF5 is a very complex and flexible format. This is why Paraview needs more support and information to understand the data that is contained in an HDF5 file received as input. One of the possibilities investigated was to produce an xdmf (xml-style, tree-structured) file containing the information on the structure of the file to be fed as input to Paraview, together with the output files[31]. Documentation for this process was not easily available[32]. After doing some research[33] I concluded that the quickest and probably most efficient way (for testing purposes at least) was to let Paraview and HDF5 interact was convert HDF5 files into txt files, very easy to read for Paraview.

A simple bash script took as input the HDF5 files and created the new txt files, one for each particle at each timestep:

```
#!/bin/bash
input='.'
output='./txt/'
i=0
while [ $i -lt 100000 ]; do

        if [ $i -eq 0 ]  || [ $(($i%500)) -eq 0 ]; then

        count=$(printf "%06d" $i)
        filename_in=$input'timestep_'$count'.h5'
```

[31] http://www.paraview.org/pipermail/paraview/2012-December/027049.html

[32] XDMF is extensively documented per se (see http://xdmf.org/index.php/Main_Page) but there are insufficient resources to cover all relevant aspects needed when translating files written using PWpyt.py into a simple structure. A tool like XDMF Generator (https://hpcforge.org/plugins/mediawiki/wiki/xdmfgenerator/index.php/XDMF_Generator) was considered to help in this process, but a number of errors and warnings slowed down the compilation process and led to believe that other quickest procedures had to be found.

[33] including talking to experts who have been dealing with issues like these before, such as James Amundson, from Fermilab.

```
        filename_out_p=$output'timestep_protons_'$count'.txt'
        filename_out_e=$output'timestep_electrons_'$count'.txt'
        filename_out_h=$output'timestep_h2plus_'$count'.txt'

        h5dump -o $filename_out_p -y -w 1000 -d protons $filename_in
        h5dump -o $filename_out_e -y -w 1000 -d electrons $filename_in
        h5dump -o $filename_out_h -y -w 1000 -d h2plus $filename_in
        fi
        let i=$i+1
done
```

This script turned out to be very efficient: it takes about 13 minutes to convert 200 HDF5 files into 600 txt files (one for each particle). However, these txt files needed further formatting, as the conversion process introduced blank spaces, blank lines and other characters. A very simple script was used to make these txt files understandable by Paraview.

```
#!/usr/bin/python
name_temp_old="/Users/diletta/Desktop/FermiWorking/hdf5_pytables_nogroups_0
826_1639/txt/timestep_";
name_temp_new="/Users/diletta/Desktop/FermiWorking/hdf5_pytables_nogroups_0
826_1639/txt/format/timestep_";


TOP_RANGE= 100000


part="electrons_"
for i in range(0, TOP_RANGE):
      if (i==0 or i%500==0):
            count= "%06d" % (i,)
            with open(name_temp_old+part+count+".txt", "r") as f:
                  content = f.readlines()
                  newfile= open(name_temp_new+part+count+".txt", "w+")
                  for line in content:
                        line= line.replace(",\n", "\n")
                        line= line.replace(" ", "")
                        if (line!= "\n"):
                              newfile.write(line)


part="protons_"
for i in range(0, TOP_RANGE):
        if (i==0 or i%500==0):
                count= "%06d" % (i,)
                with open(name_temp_old+part+count+".txt", "r") as f:
                        content = f.readlines()
                        newfile= open(name_temp_new+part+count+".txt",
"w+")
```

```
                        for line in content:
                                line= line.replace(",\n", "\n")
                                line= line.replace(" ", "")
                                if (line!= "\n"):
                                        newfile.write(line)

part="h2plus_"

for i in range(0, TOP_RANGE):
        if (i==0 or i%500==0):
                count= "%06d" % (i,)
                with open(name_temp_old+part+count+".txt", "r") as f:
                        content = f.readlines()
                        newfile= open(name_temp_new+part+count+".txt",
"w+")
                        for line in content:
                                line= line.replace(",\n", "\n")
                                line= line.replace(" ", "")
                                if (line!= "\n"):
                                        newfile.write(line)
```

Now these files are ready to be imported in Paraview.

Documentation

A git repository for the e-column warp simulation output and related scripts was created in the cdcvs server. It can be cloned by:

```
git clone ssh://p-ecolumns@cdcvs.fnal.gov/cvs/projects/ecolumns
```

The content is also available at https://cdcvs.fnal.gov/redmine/projects/ecolumns.

Bibliography

A. V. Burov, G. W. Foster, V. Shiltsev. 2000. "FERMILAB-TM-2125."
C. S. Park, J. Thangaraj. 2014. "Simulation of Space Charge Compensation Using e-Columns at IOTA."
D. B. Michelson, A. Henja. n.d. "A High Level Interface to the HDF5 File Format."
D. Garbor. n.d. "Nature 160 (1947) 89."
D. Mohl. n.d. "CERN/PS 93-59 (AR)."

G. I. Dimov. n.d. "Particle Accelerators 14 (1984) 155."
G. I. Dimov, V. E. Chupriyanov. n.d. "Particle Accelerators 14 (1984) 155."
G. Stancari†, A. Burov, K. Carlson, D. Crawford, V. Lebedev, J. Leibfritz, M. McGee, S. Nagaitsev, L. Nobrega, C. S. Park, E. Prebys, A. Romanov, J. Ruan, V. Shiltsev, Y.-M. Shin1, C. Thangaraj, A. Valishev. n.d. "Electron lens for the fermilab integrable optics test accelerator ."
M. Chung. n.d. "Trapped Electron Plasmas for Space-Charge Compensation in High Intensity Circular Accelerators." *DOE Early Career Research Program LAB 12-751.*
Moreland, K. n.d. "The ParaView Tutorial."
S. Nagaitsev, A. Valishev, D. Shatilov, V. Danilov. n.d. "Proceedings of IPAC12, 2012, p. 16."
S. Nagaitsev, D. Broemmelsiek, A. Burov, K. Carlson, C. Gattuso, M. Hu, T. Kroc, L. Prost, S. Pruss, M. Sutherland, C. W. Schmidt, A. Shemyakin, V. Tupikov, A. Warner, G. Kazake- vich, S. Seletskiy,. n.d. "Phys. Rev. Lett. 96 (2006) 044801."
S. Webb, D. Bruhwiler, D. Abell, K. Danilov, J. R. Cary, S. Nagaitsev, A. Valishev, V. Danilov, A. Shishlo. n.d. "Proceedings of IPAC12, 2012, p. 2961."
U. Ayachit. n.d. "The ParaView Guide - Community Edition."
V. Dudnikoiv, C. Ankenbradt. n.d. "Proceedings of PAC11, 2011, p. 1789."
V. Shiltsev, Y. Alexahin, V. Kamerdzhiev, V. Kapin, G. Kuznetsov. n.d. "AIP Conference Proceedings 1086, 2009, pp. 649–654."

Acknowledgments

To Charles Thangaraj – for the incredible supervision and mentorship;

To Chong Shik Park and Giulio Stancari – for the continuous guidance as co-supervisors;

To the **Fermilab** Accelerator Science and Technology (**FAST**) **-** Integrable Optics Test Accelerator (**IOTA**) department – for the interest and collaboration;

To Giorgio Bellettini, Emanuela Barzi, Simone Donati, CAIF– for the thoughtful support;

To Fermilab – for the truly extraordinary opportunity.

About the author

I am a Computer Science and Engineering Master's student at Politecnico di Milano, Italy. My main fields of interest are Artificial Intelligence and Machine Learning.

diletta.milana@mail.polimi.it