Controlling Hundreds of GPU-Powered Plasma-Physics Simulations with Machine Learning Algorithms

Rémi Lehe[1]

in collaboration with
Juliane Mueller[2], Manuel Kirchen[3]

[1] BELLA Center & Accelerator Modeling Group, **LBNL**, USA
[2] Computational Research Division, **LBNL**, USA
[3] Center for Free-Electron Laser Science & Department of Physics, **University of Hamburg**, Germany

Context: Laser-wakefield acceleration

Laser-wakefield acceleration

Particle accelerators bring elementary particles to high speed/energy.
Applications in fundamental science, but also medicine, industry, security.

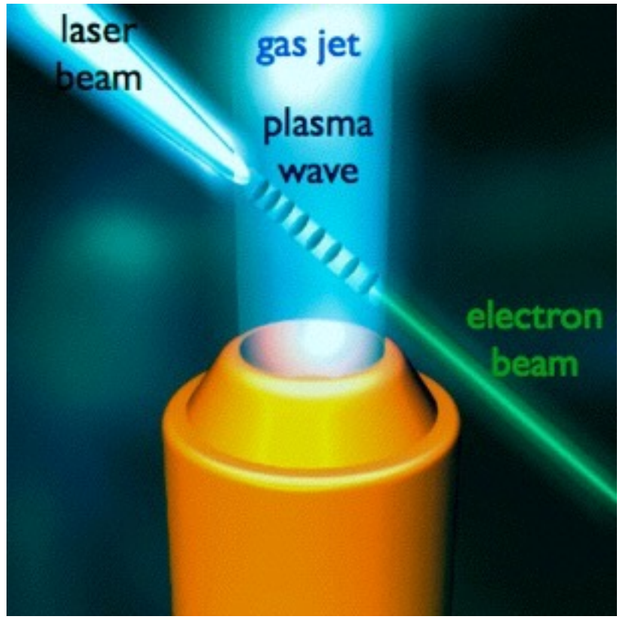
Conventional accelerators

~ 0.02 GeV/m

e.g. 8 GeV over **~500 m** of RF cavities

Laser-wakefield accelerators

~ 50 GeV/m

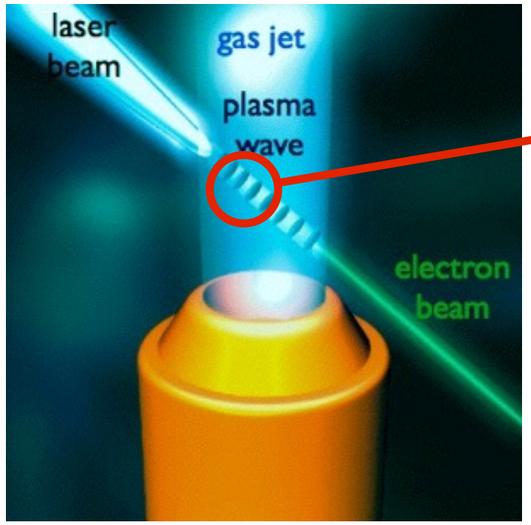


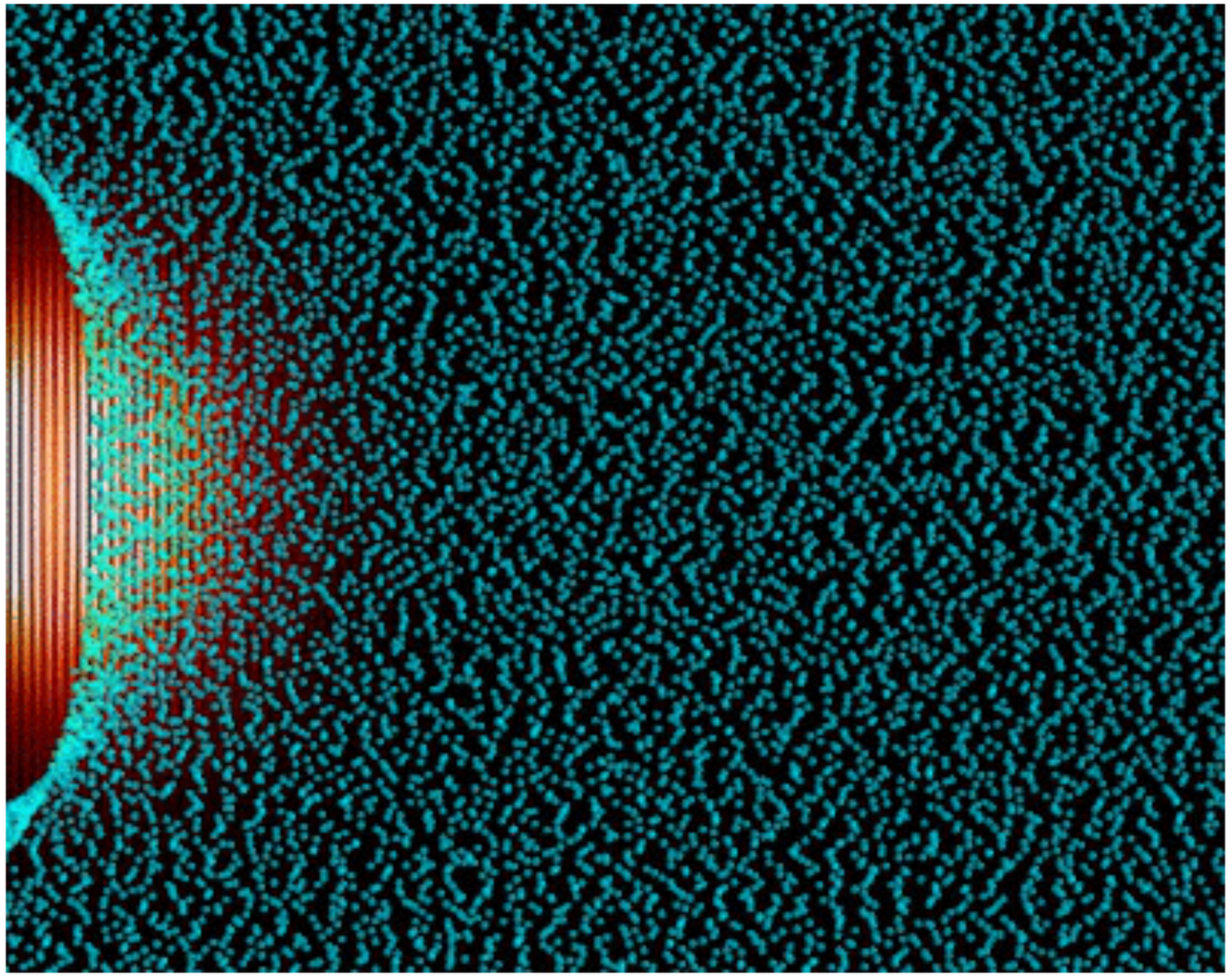
e.g. 4 GeV over **9 cm** of gas

Laser-wakefield acceleration

Physics of laser-wakefield

- The laser pulse pushes away the electrons of the gas
- This creates an accelerating structure

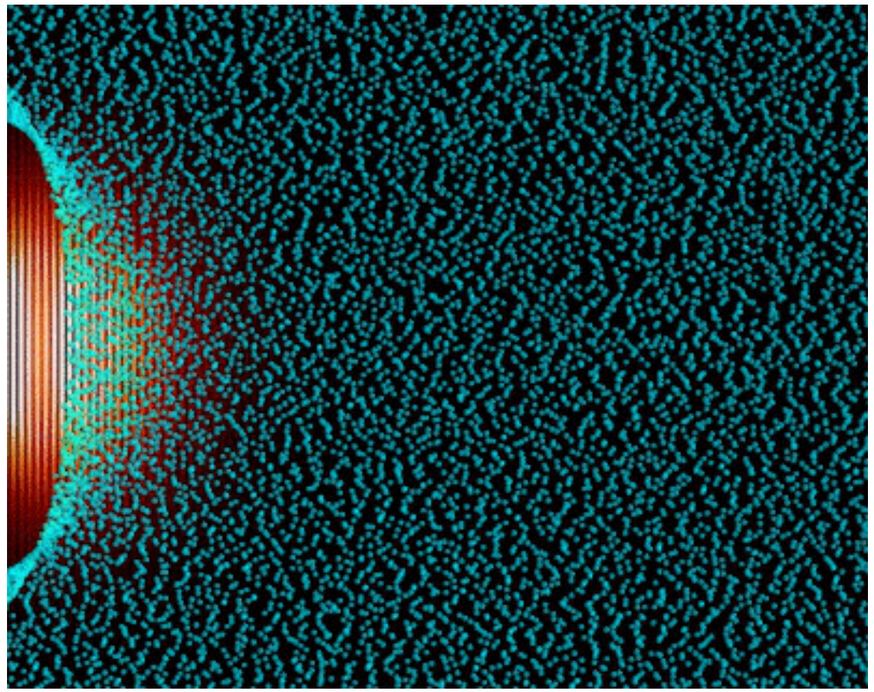


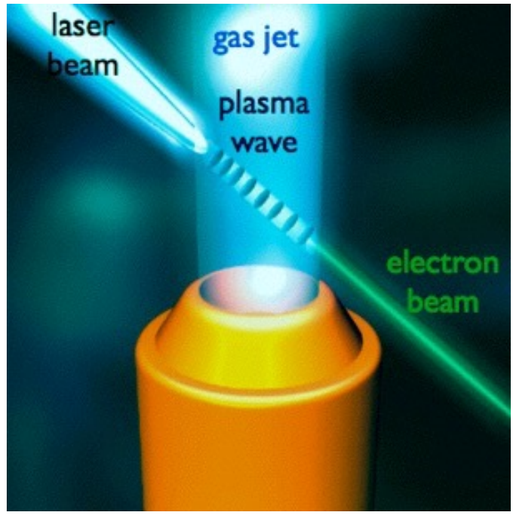


Predicting injection in laser-wakefield acceleration

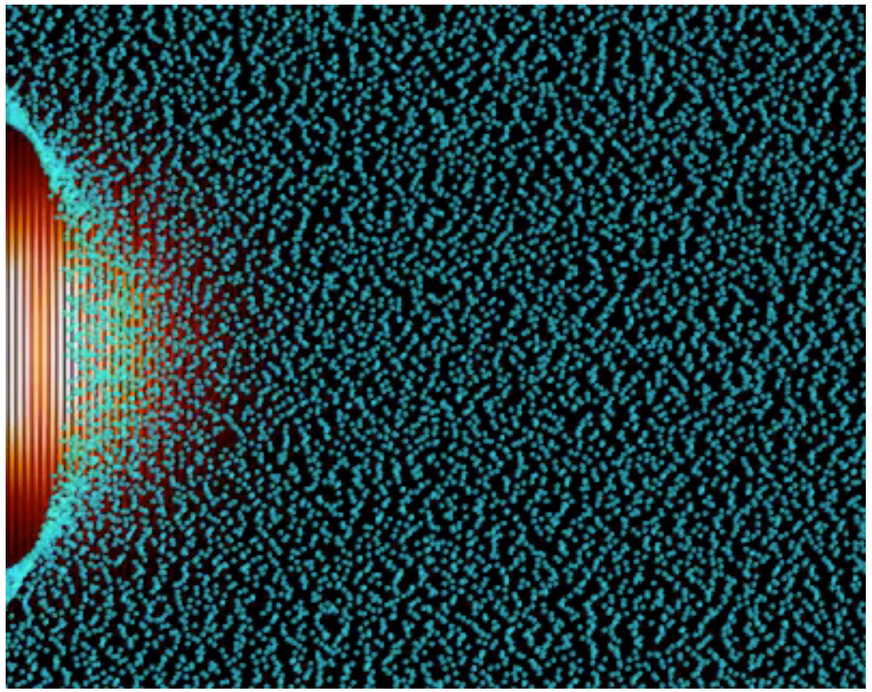
Depending on the parameters (gas density, laser intensity, …) **injection** may or may not happen.

Injection





No injection



Predicting injection in laser-wakefield acceleration

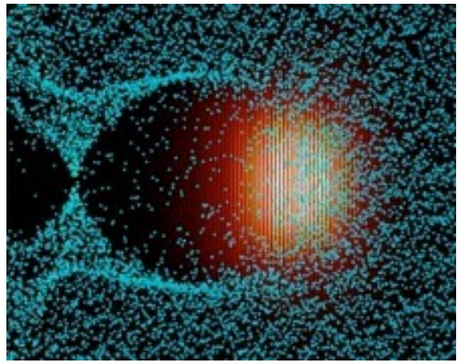
Predicting **when injection happens** requires:

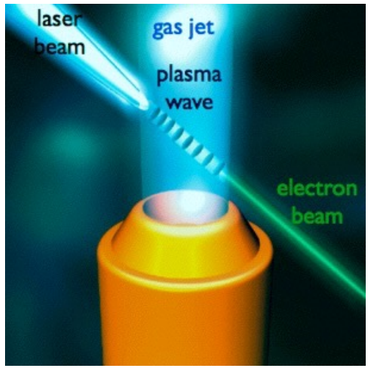
- **Numerical simulations:**
  no robust, comprehensive analytical theory available

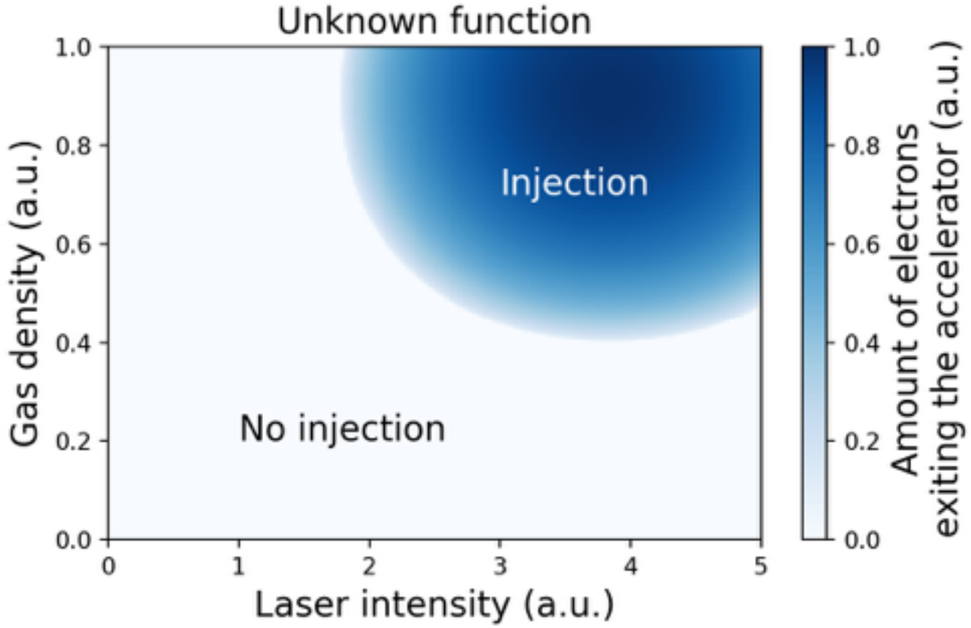
  (First part of the talk)

- **<u>Many</u> numerical simulations:**
  need to explore parameter space, esp. to identify the **threshold**

  (Second part of the talk)







Simulating injection in laser-wakefield acceleration

Particle-In-Cell simulations (PIC)

- Represent the gas/plasma by a set of macroparticles

- Solve the equations of motion for each macroparticle

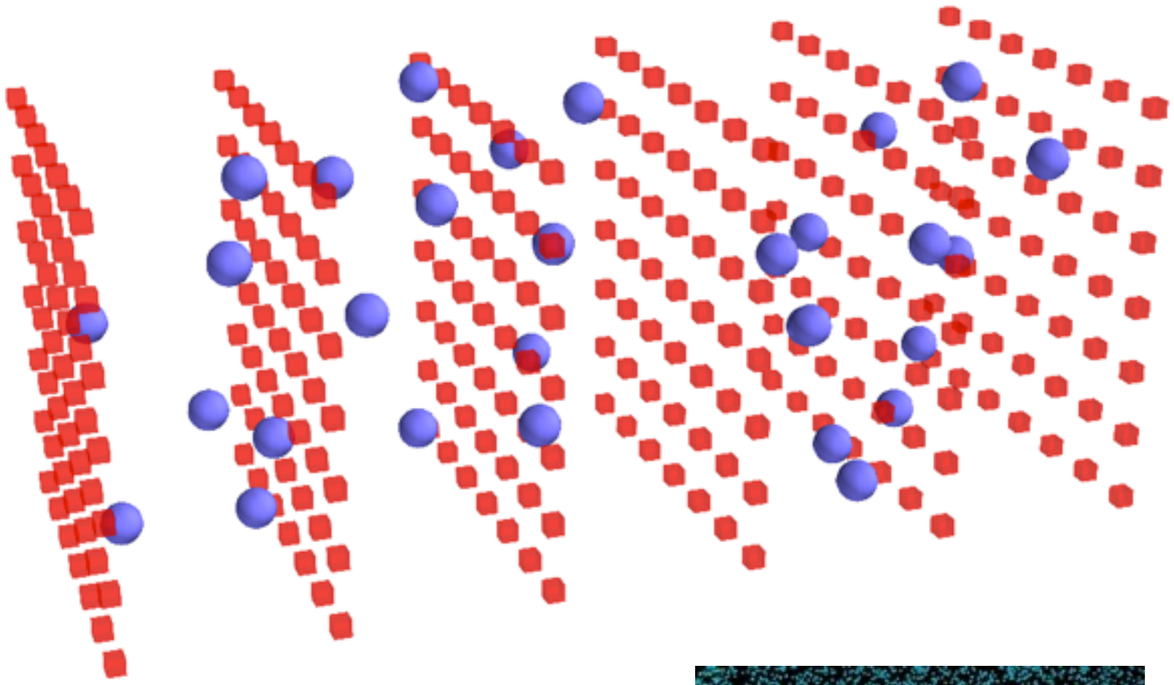
$$\frac{d\,\boldsymbol{p}}{dt} = q\boldsymbol{E} + q\boldsymbol{v} \times \boldsymbol{B}$$

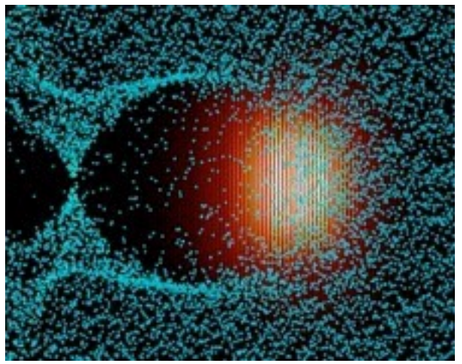
$$\frac{d\,\boldsymbol{x}}{dt} = \frac{\boldsymbol{p}}{\gamma m}$$

- Self-consistently solve the Maxwell equations on a discrete grid.

$$\frac{1}{c^2}\partial_t \boldsymbol{E} = \boldsymbol{\nabla} \times \boldsymbol{B} - \mu_0 \boldsymbol{J}$$

$$\partial_t \boldsymbol{B} = -\boldsymbol{\nabla} \times \boldsymbol{E}$$



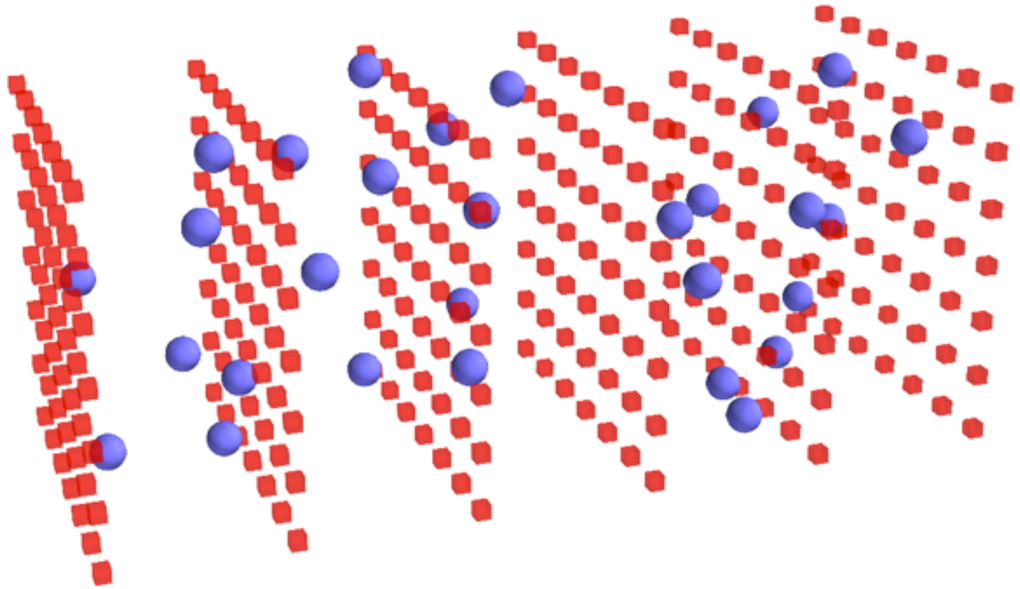


Computational cost of simulations for full-3D simulations

- For problems of injection in laser-wakefield:
  ~10^8 grid points & ~10^9 macroparticles
  over ~10^5 iterations of the PIC loop

- Large requirements in **memory**
  and **computational power**:
  typically several hours on hundreds of CPU / GPUs

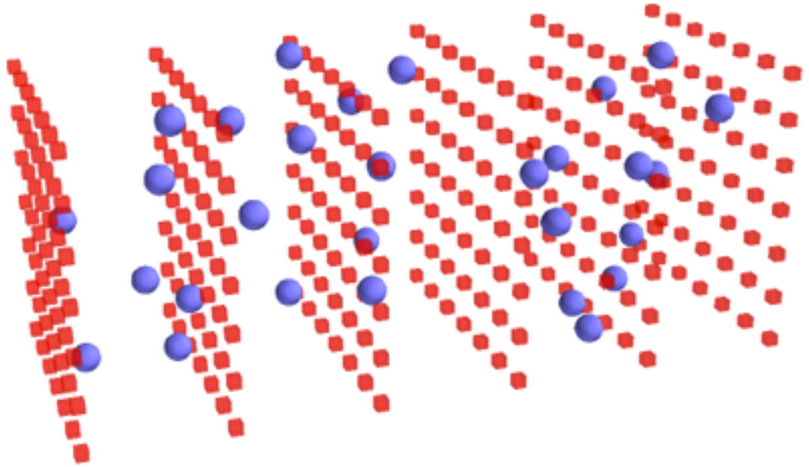
- Exploring the physical parameter space with multiple simulations is **impractical**.

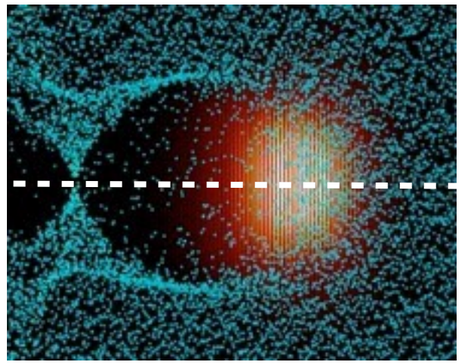


Quasi-cylindrical geometry

Traditional PIC codes

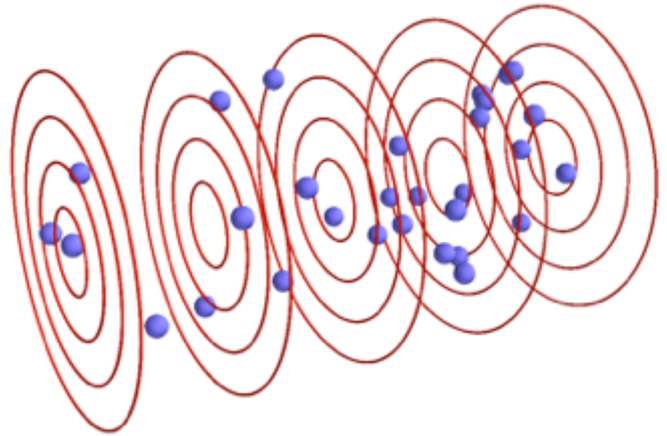
Use full **3D mesh**





Quasi-cylindrical geometry

Use a few **2D meshes** in (r, z)
(azimuthal mode decomposition;
≈ averaging over azimuthal angle)



For physical situations that have **close-to-cylindrical** symmetry:
Quasi-cylindrical simulations **vastly reduce** the **computational** and **memory** requirements.

→ Simulations can potentially fit on a **single GPU**

The code FBPIC

Fourier-Bessel Particle-In-Cell code

- Quasi-cylindrical Particle-In-Cell algorithm
- Runs on CPU and GPU
- Written in Python/Numba
- Innovative spectral algorithm based on Hankel transform
- Open-source: https://github.com/fbpic/fbpic.git

GPU kernel in Python/Numba

Example

Integrate over one timestep

$$\frac{d\,\boldsymbol{x}}{dt} = \frac{\boldsymbol{p}}{\gamma m}$$

CPU code

Function definition

```
import numba

@numba.jit()
def push_position_cpu( x, y, z, px, py, pz, gamma, m, dt):
    """
    Advance the particle positions over one timestep

    Parameters:
    -----------
    x, y, z, px, py, pz, gamma: 1darray of floats
         Position and momenta, one element per particle

    m, dt: floats
    """
    N_particles = len(x)

    # Loop over particles
    for i in range(N_particles):

        x[i] += dt * px[i]/( gamma[i]*m )
        y[i] += dt * py[i]/( gamma[i]*m )
        z[i] += dt * pz[i]/( gamma[i]*m )
```

Function call

```
push_position_cpu(x, y, z, px, py, pz, gamma, m, dt )
```

GPU code

Function definition

```
from numba import cuda

@cuda.jit()
def push_position_gpu( x, y, z, px, py, pz, gamma, m, dt):
    """
    Advance the particle positions over one timestep

    Parameters:
    -----------
    x, y, z, px, py, pz, gamma: 1darray of floats
         Position and momenta, one element per particle

    m, dt: floats
    """
    N_particles = len(x)

    i = cuda.grid(1)
    if i < N_particles:

        x[i] += dt * px[i]/( gamma[i]*m )
        y[i] += dt * py[i]/( gamma[i]*m )
        z[i] += dt * pz[i]/( gamma[i]*m )
```

Function call

```
tpb = 256
bpg = len(x)/tpb + 1

push_position_gpu[ tpb, bpg ](x, y, z, px, py, pz, gamma, m, dt )
```

Carrying out multiple simulations

Mapping the injection threshold

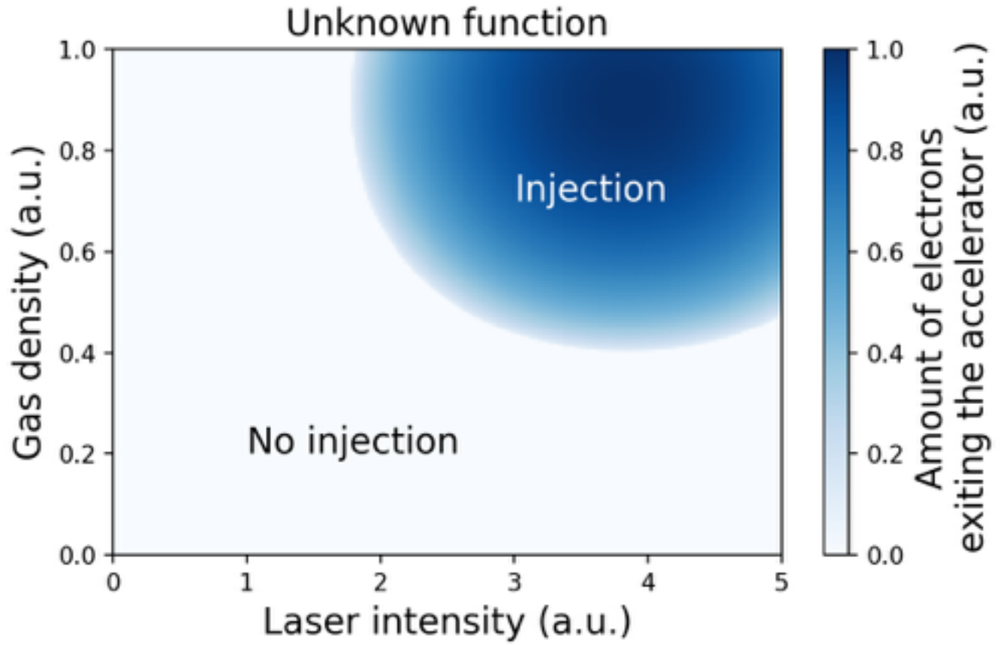
Aim

Find the location of the **injection threshold** in parameter space

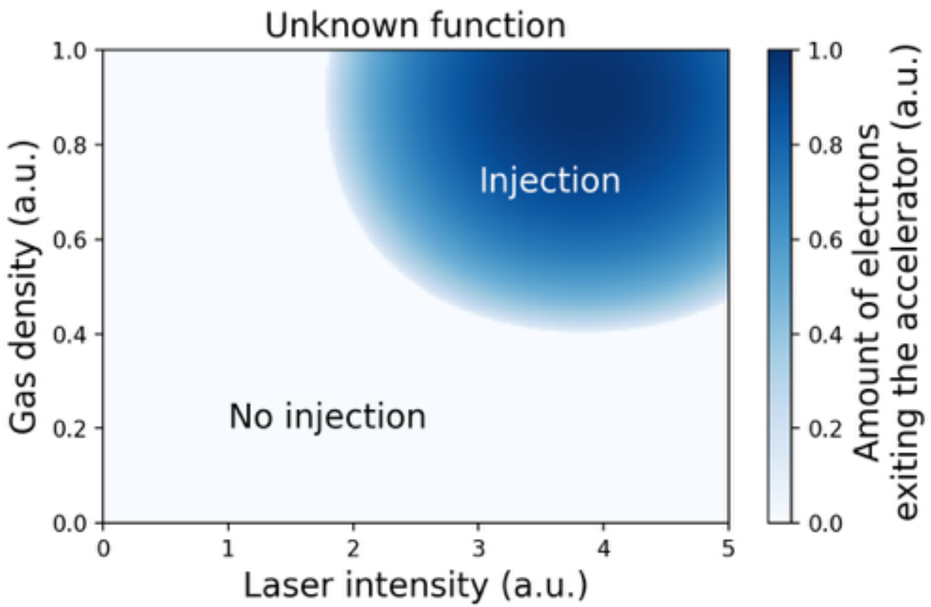
Available tools

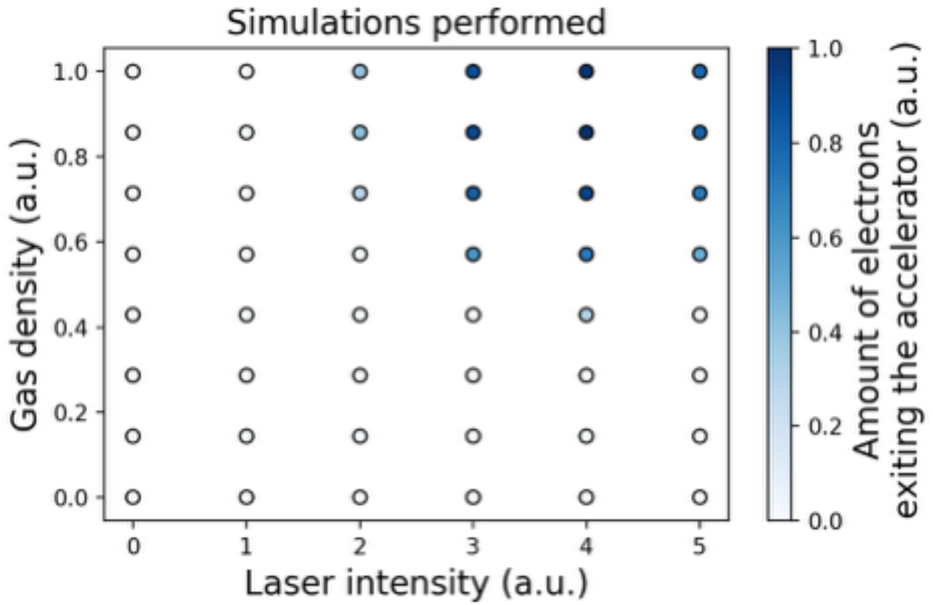
- Fast numerical simulations on **single GPU**
- Access to GPU clusters
  - Titan @ OLCF
  - Lawrencium @ LBNL

  Can carry out several single-GPU simulations in parallel



One solution: grid search





Advantages

- Simple
- Each simulation is independent and can be performed in parallel

Drawbacks

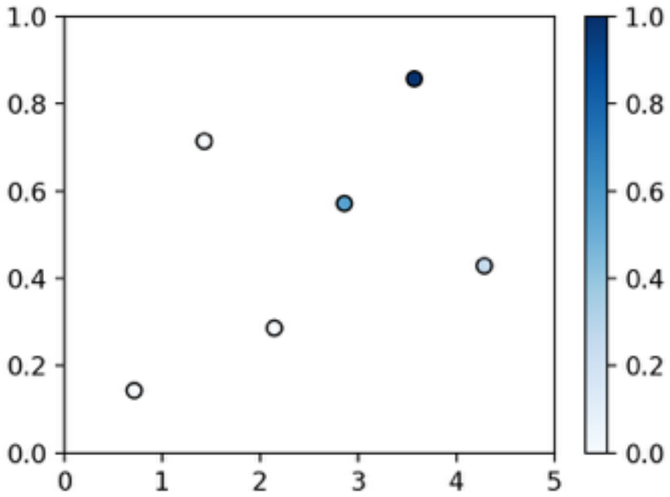
- Costly and inefficient (esp. in high dimension parameter space)
- No rationale on how to space points

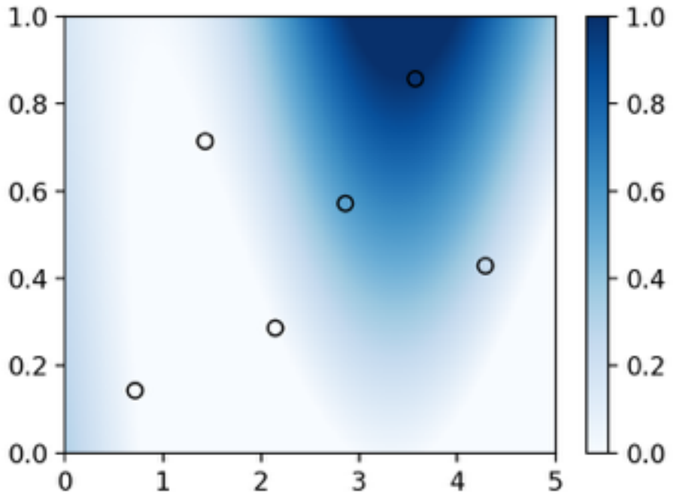
Alternative solution: surrogate model

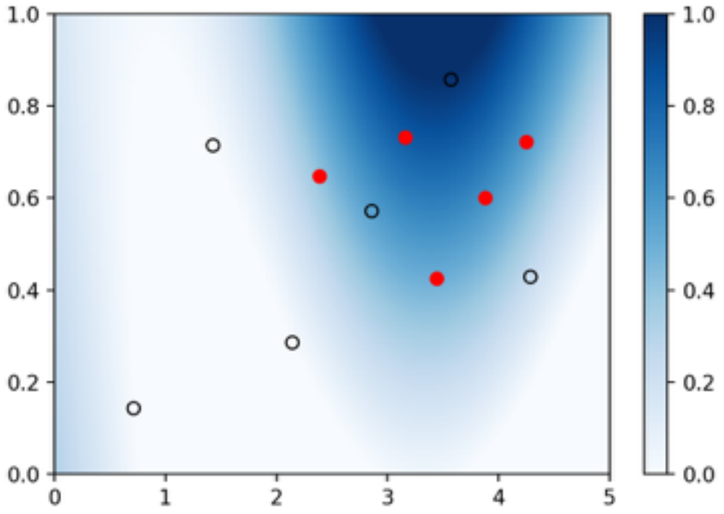
Perform a few simulations

Build **surrogate model** that **interpolates** the existing data but can be computed quickly

Choose **next simulation points** (by optimizing a cost function)







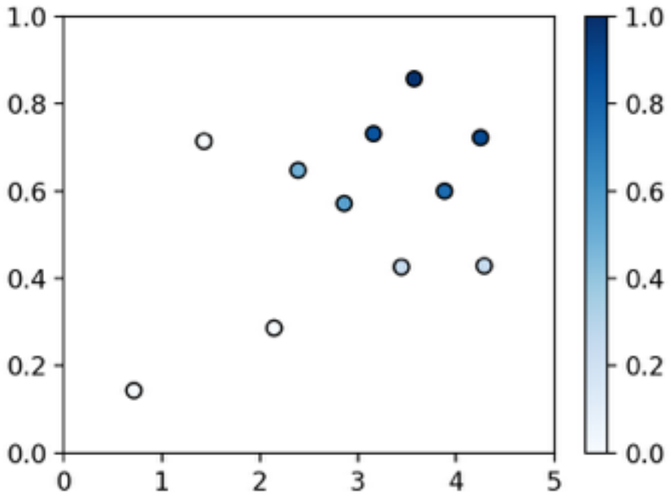
Iterate and update/refine model

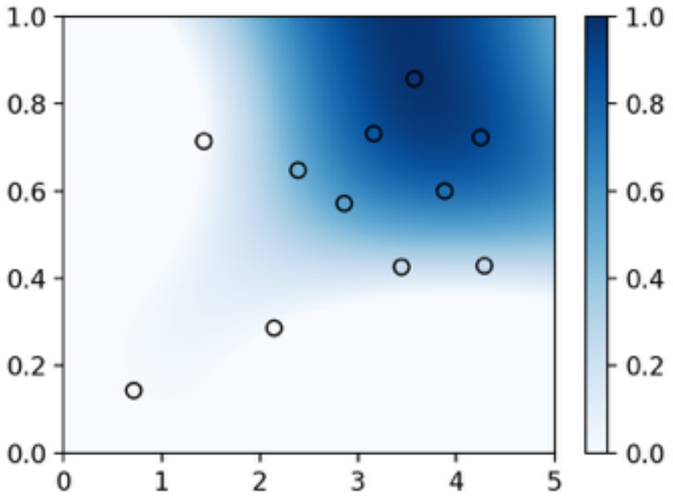
Alternative solution: surrogate model

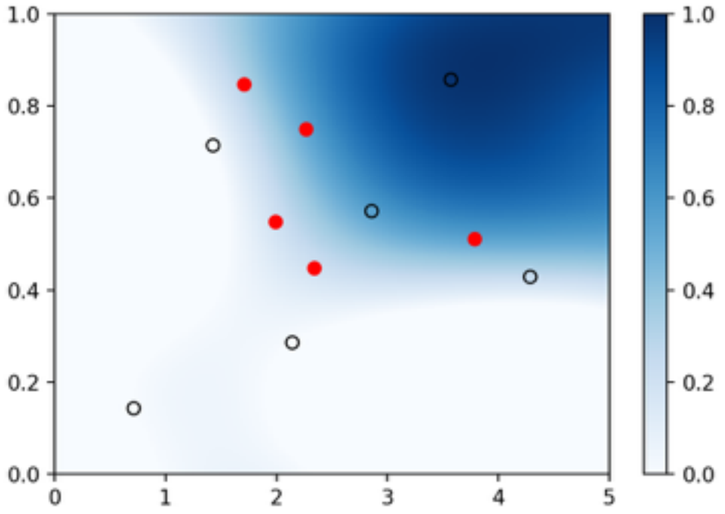
Perform a few simulations

Build **surrogate model** that **interpolates** the existing data but can be computed quickly

Choose **next simulation points** (by optimizing a cost function)





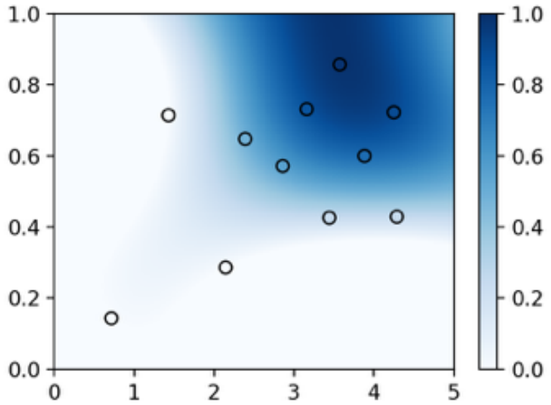


Iterate and update/refine model

Building the surrogate model: the Kriging method

Surrogate model

$$y = f(\vec{X})$$



Scalar:
in our case, amount of electrons exiting

Multi-dimensional vector:
here: density + laser intensity

f is obtained by **training** on past simulation results, i.e. on the set of $(\vec{X}^{(j)}, y^{(j)})$

One type of surrogate model: **Kriging method**

- $f(\vec{X})$ is a **weighted sum** of the values $y^{(j)}$ at nearby points $\vec{X}^{(j)}$
- The weights depend on an **anisotropic** distance between points

$$d(\vec{X}, \vec{X}^{(j)})^2 = \sum_{i=1}^{d} \frac{(X_i - X_i^{(j)})^2}{\ell_i^2}$$

- The ℓ_i are the **characteristic length scales** along each dimension, and are **estimated** from past simulation results.
- Bonus: the Kriging provides an estimate of the **uncertainty** of the model

Choosing the next point: the cost function

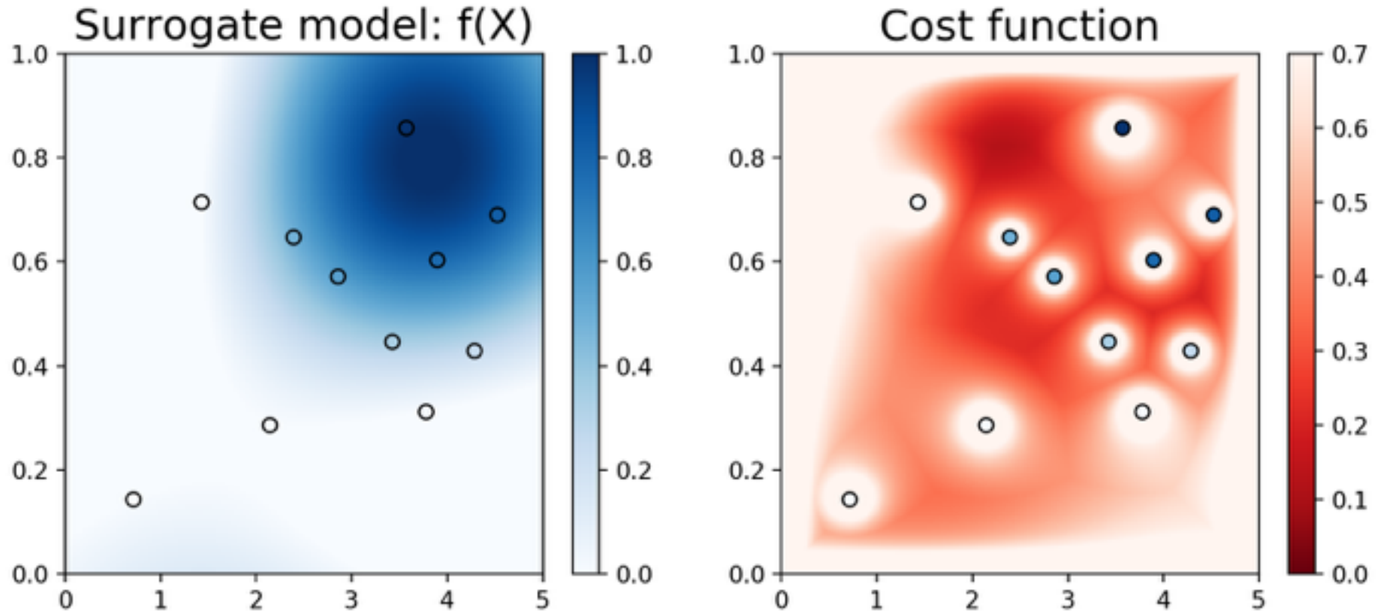
Chosen cost function

$$Cost(\vec{X}) = -Objective(f(\vec{X})) + \frac{1}{d(\vec{X} - \vec{X}^{(j_{nearest})})}$$

Chosen to favor points near the injection threshold

Surrogate model

Prevents being too close to past simulation points



Minimization with genetic algorithm to find the next simulation points

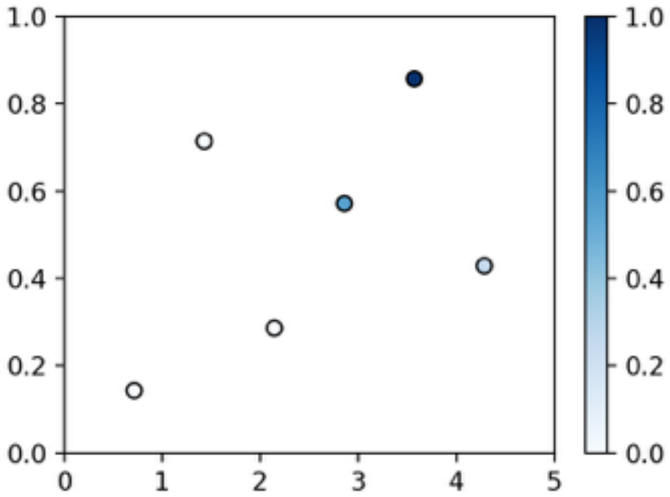
(require many evaluations but evaluation is fast with surrogate models)

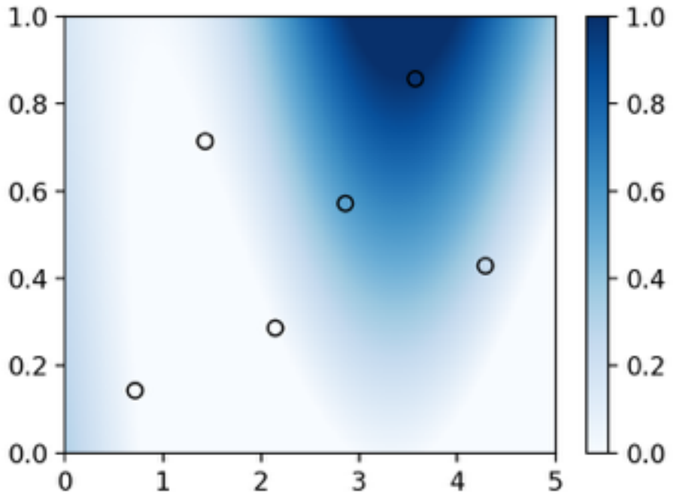
On a GPU cluster: series of batch jobs with dependencies

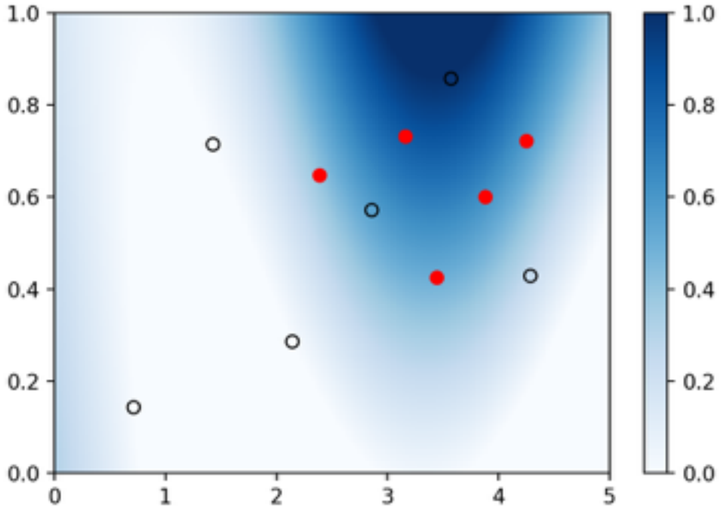
Perform a few simulations

Build **surrogate model** that **interpolates** the existing data but can be computed quickly

Choose **next simulation points** (by optimizing a cost function)







Iterate and update/refine model

On a GPU cluster: series of batch jobs with dependencies

Perform a few simulations

GPU
GPU
GPU
GPU
GPU

Multi-GPU batch job: one simulation per GPU

(a few hours)

Build **surrogate model** that **interpolates** the existing data but can be computed quickly

Choose **next simulation points** (by optimizing a cost function)

CPU **Single-CPU** batch job

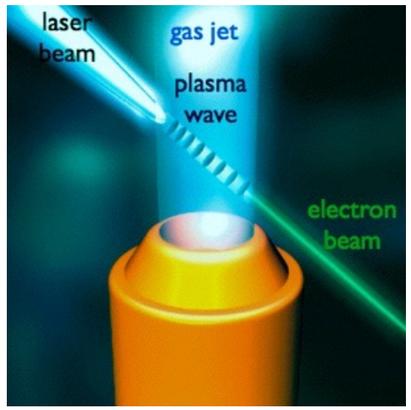
(a few minutes)

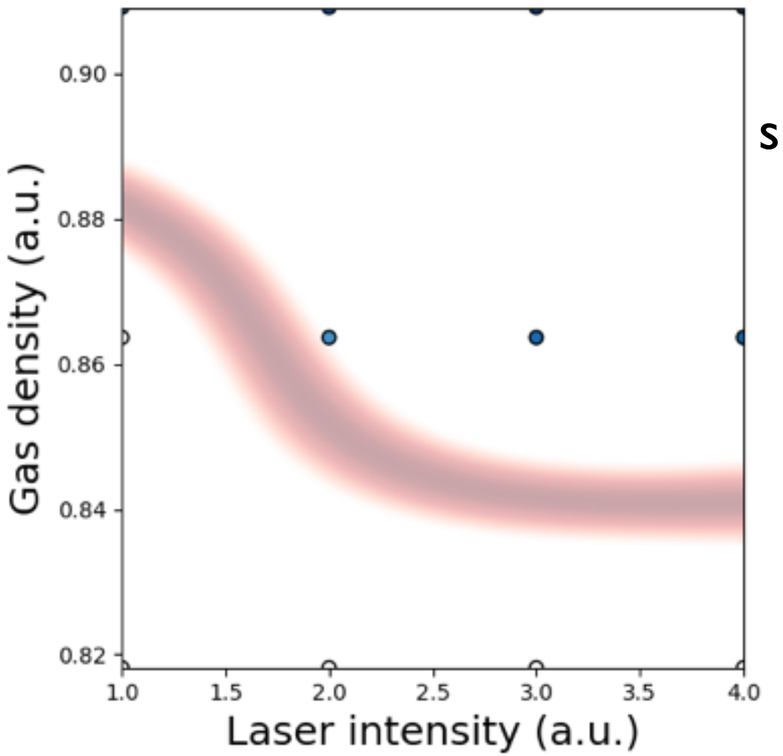
Job dependencies through SLURM / PBS

Iterate and update/refine model

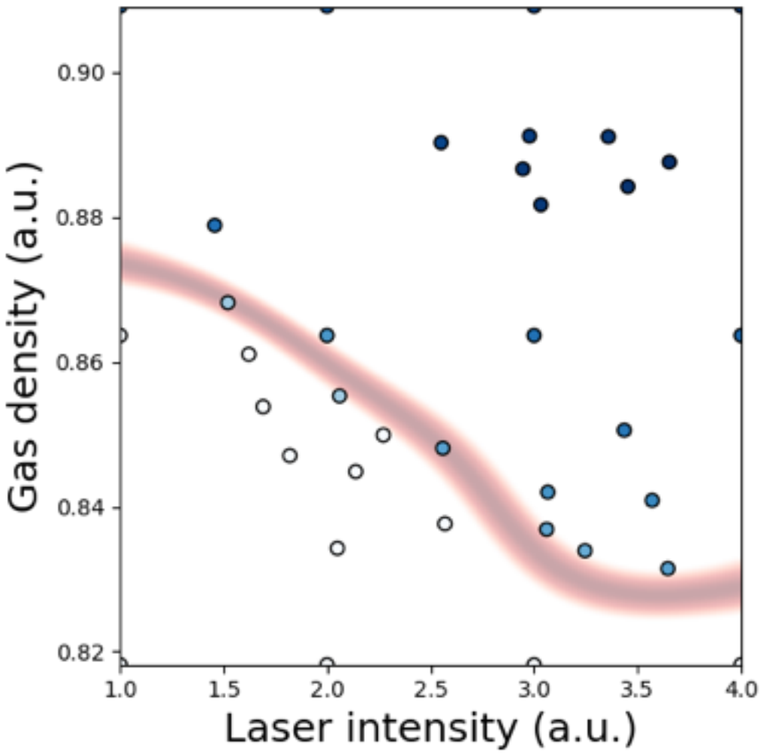
Results from actual simulations

$E_{laser} \sim 3\ J$





Iterating and refining surrogate model



Width of red line represents uncertainty on the position of the injection threshold.

Conclusion

- **Quasi-cylindrical PIC code on GPU** allows to run multiple simulations at **low computational cost**, for injection problems

- Algorithm based on **surrogate model** allows to rapidly **concentrate the simulation effort** on the region of interest, in parameter space

Acknowledgements

GPU clusters

- **Titan @ OLCF**
  This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

- **Lawrencium @ LBNL**
  This research used the Lawrencium computational cluster resource provided by the IT Division at the Lawrence Berkeley National Laboratory (Supported by the Director, Office of Science, Office of Basic Energy Sciences, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231)

Funding

- **DoE**
  This work was supported by the Director, Office of Science, Office of High Energy Physics, U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231

Open-source software

- **Numba**
- **Scikit-learn**
- **Inspyred**

Thank you for your attention.