



Prototyping a 2D Physics-Based Animation with Python

Vincent Bonnet

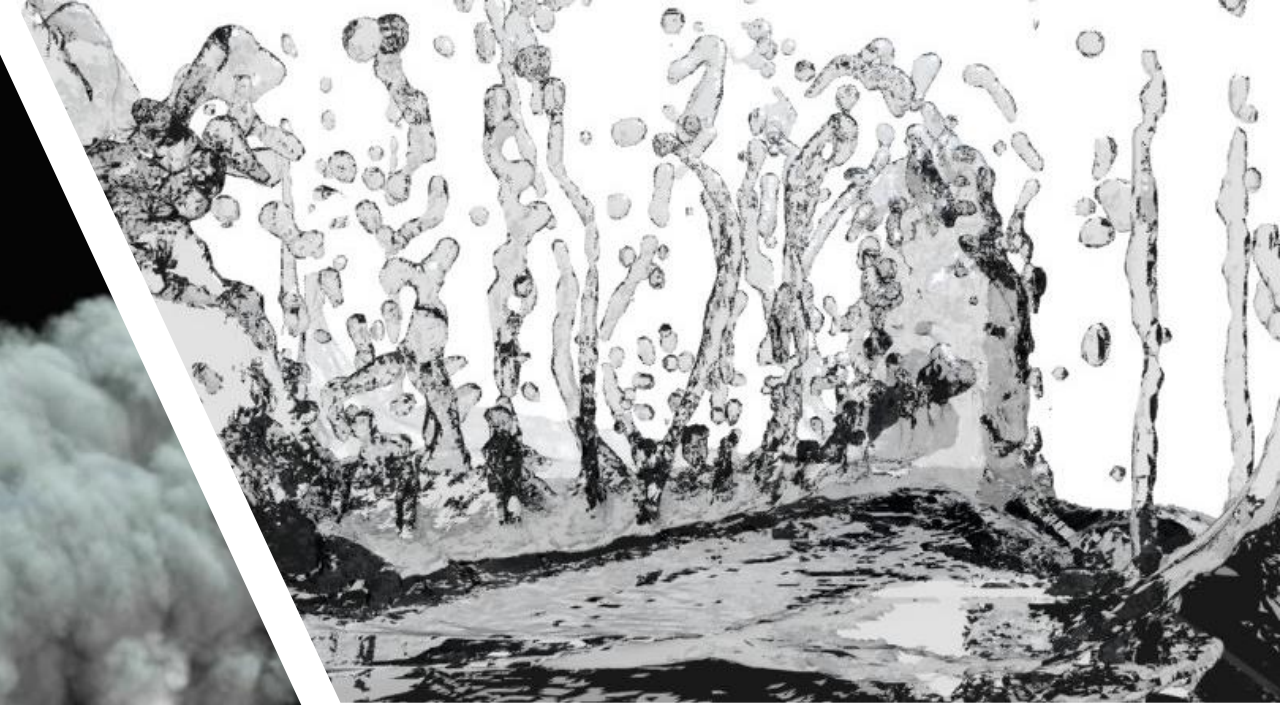
Python Meetup - March 2019



Content

- Integration
 - Standard Plugin
 - Inter-process communication
- Solver
 - Solver in a Nutshell
 - Python Ecosystem

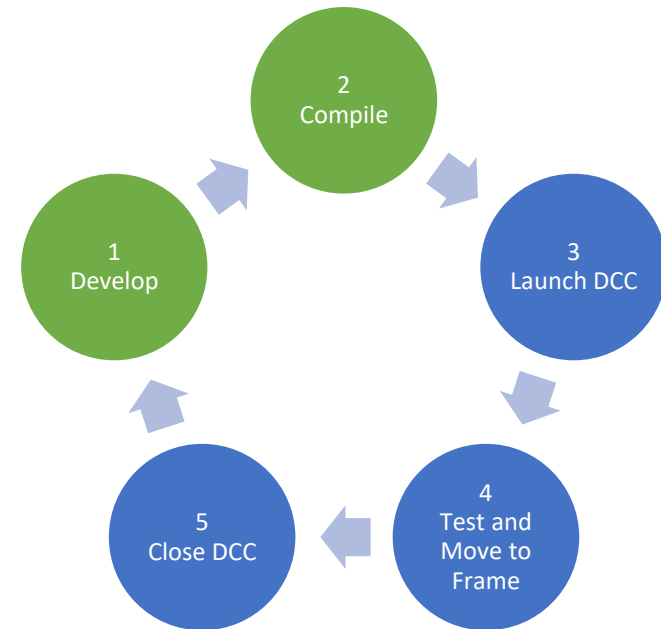
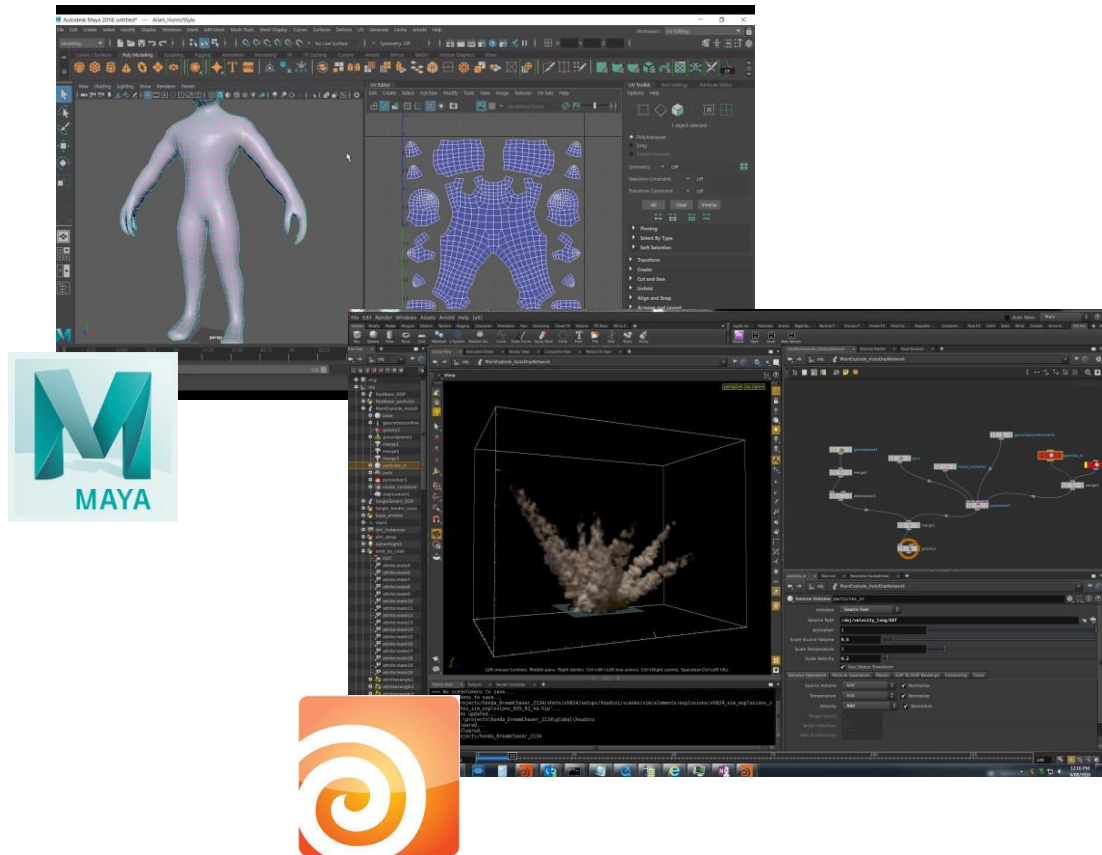
Physics-Based Animation



Part 1 - Integration



Traditional Integration Plugin System



Inter-process communication

Process A - Client



Inter-process communication

Process A - Client



Python API

Inter-process communication

Process A - Client



Python API



Process B - Client

Inter-process communication

Process A - Client

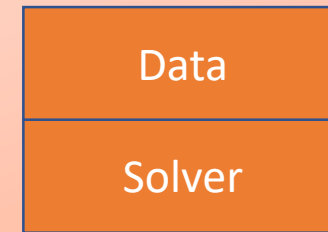


Python API



Process B - Client

Process C - Server



Inter-process communication

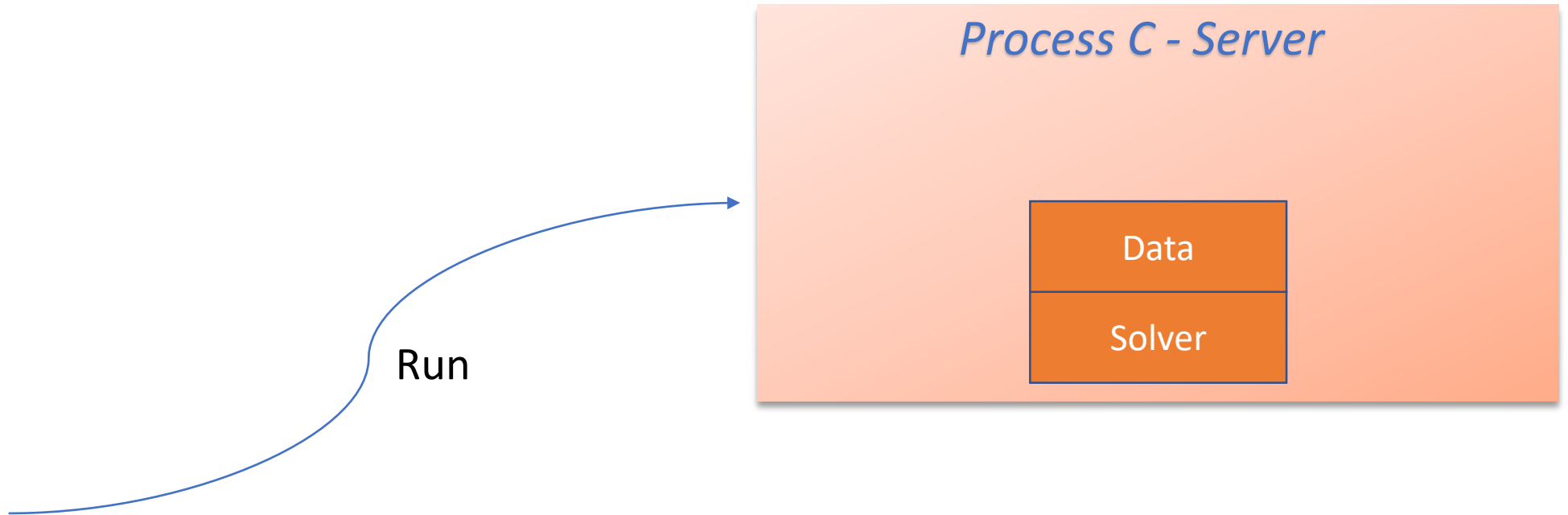
Process A - Client



Python API



Process B - Client



Inter-process communication

Process A - Client



Python API

Update Data



Process B - Client

Process C - Server

Data

Solver

Inter-process communication

Process A - Client



Python API



Process B - Client

Update Data

Update
Data/Program

Process C - Server

Data

Solver

A fluffy, light-brown and white kitten is perched on a windowsill, looking out a window. Its reflection is visible in the glass. The scene is dimly lit, with the light coming from the window. The text "Waiting for Python 3 ..." is overlaid on the image in a white, sans-serif font, preceded by a vertical line.

| Waiting for
Python 3 ...

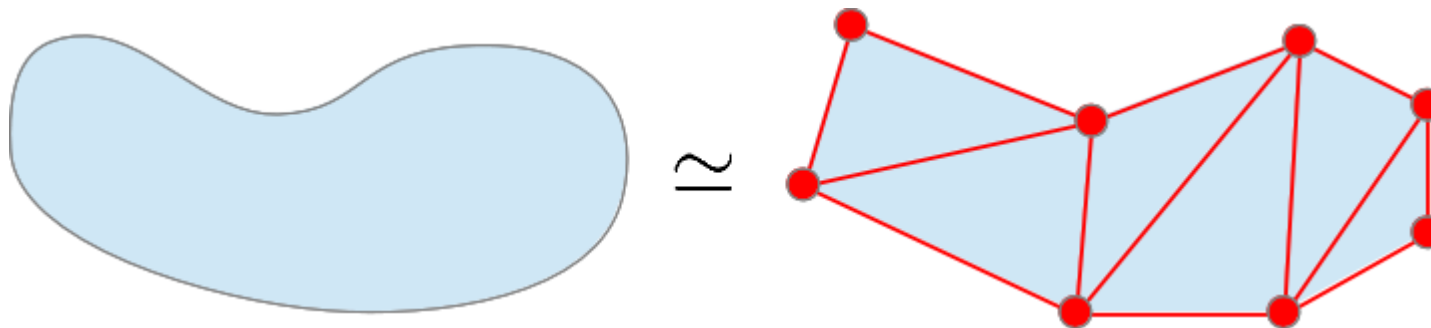
A close-up photograph of a person's hands holding a Rubik's cube. The hands are positioned to hold the cube from the sides, with fingers visible. The cube is partially solved, showing red, blue, and yellow faces. The background is a solid, dark blue color. The text "Part 2 - Solver" is overlaid in white, centered on the image.

Part 2 - Solver

Solver in a Nutshell

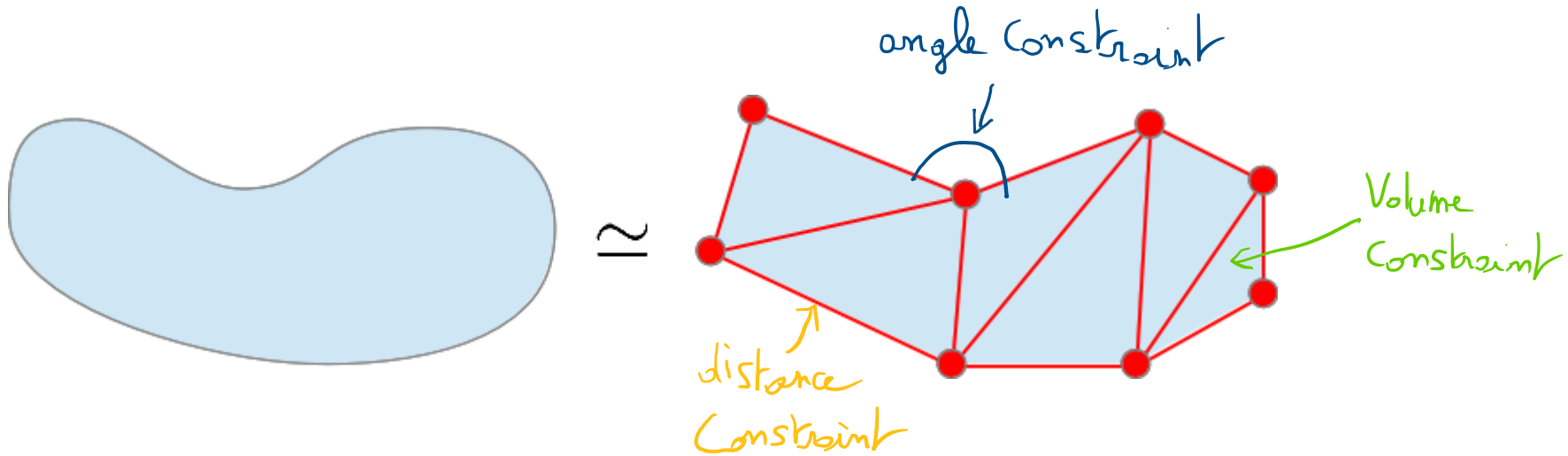
Solver Requirements

Discretization



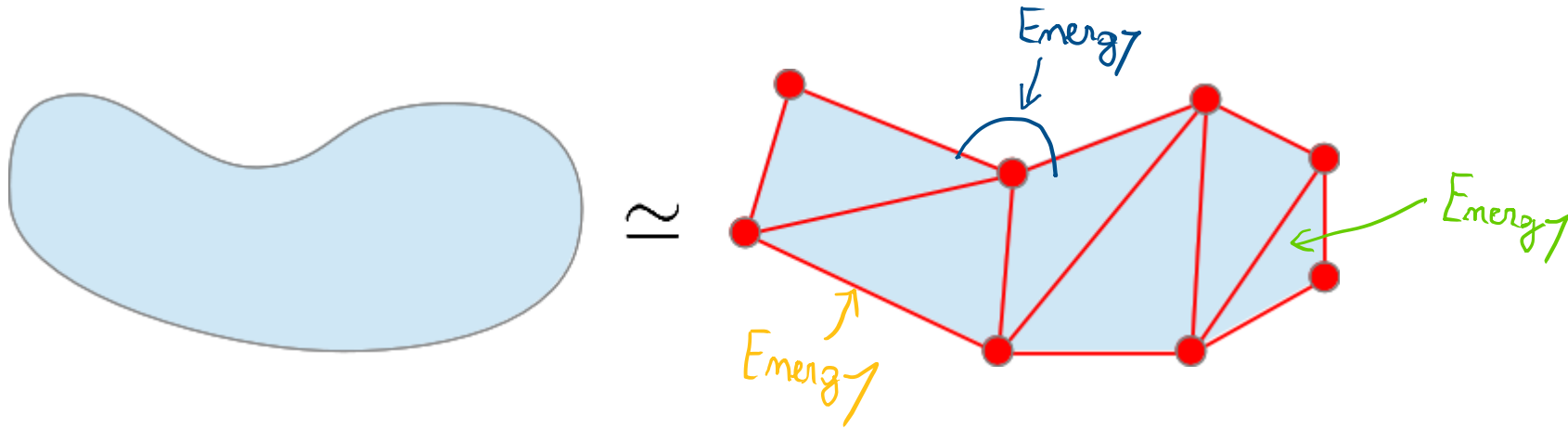
Solver Requirements

Force Computation



Solver Requirements

Force Computation

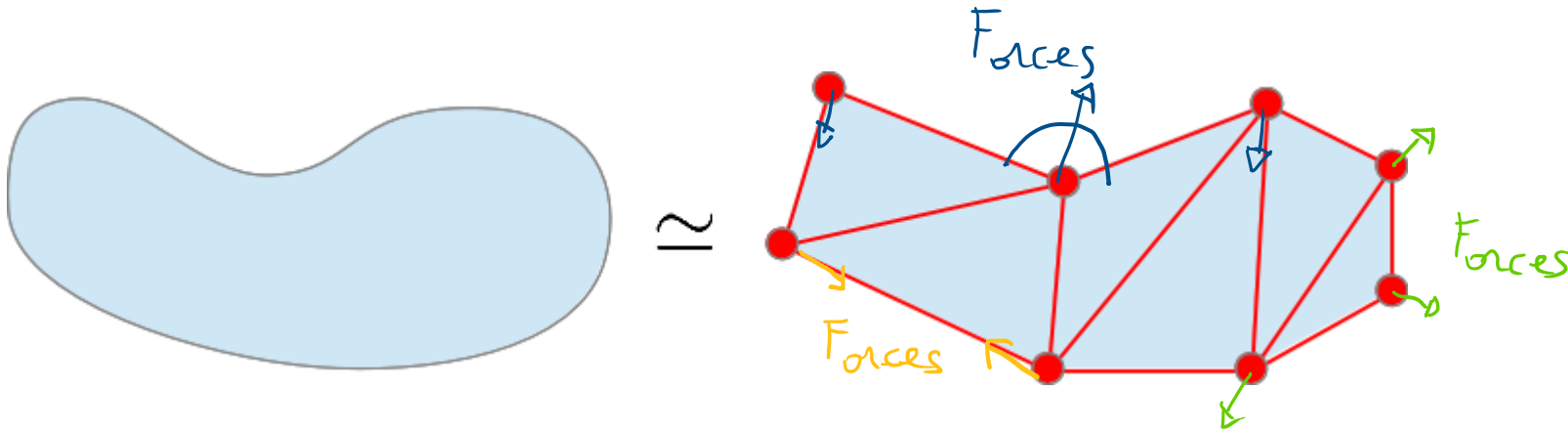


potential energy E based on constraint C

$$E(\mathbf{x}_1, \dots, \mathbf{x}_n) = \frac{1}{2}kC(\mathbf{x}_1, \dots, \mathbf{x}_n)^2$$

Solver Requirements

Force Computation



$$\begin{aligned}\mathbf{F}_j(\mathbf{x}_1, \dots, \mathbf{x}_n) &= -\frac{\partial}{\partial \mathbf{x}_j} E(\mathbf{x}_1, \dots, \mathbf{x}_n) \\ &= -kC(\mathbf{x}_1, \dots, \mathbf{x}_n) \frac{\partial C(\mathbf{x}_1, \dots, \mathbf{x}_n)}{\partial \mathbf{x}_j}\end{aligned}$$

Solver Requirements

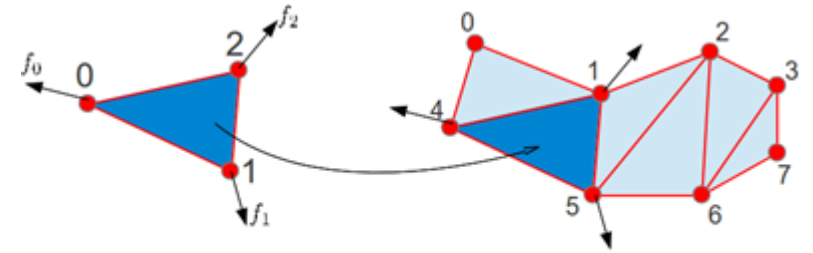
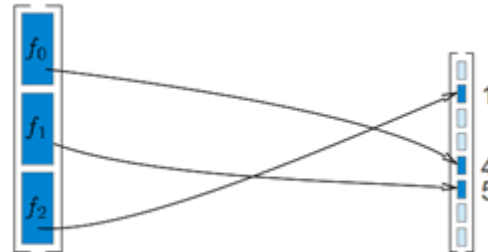
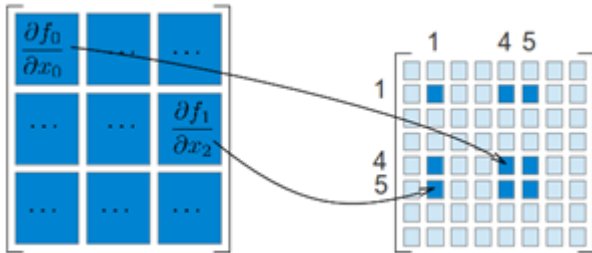
Forward in Time

$$\left(M - \frac{df}{dx} h^2\right) \Delta_v = h \left(f_0 + h \frac{df}{dx} v_0\right)$$

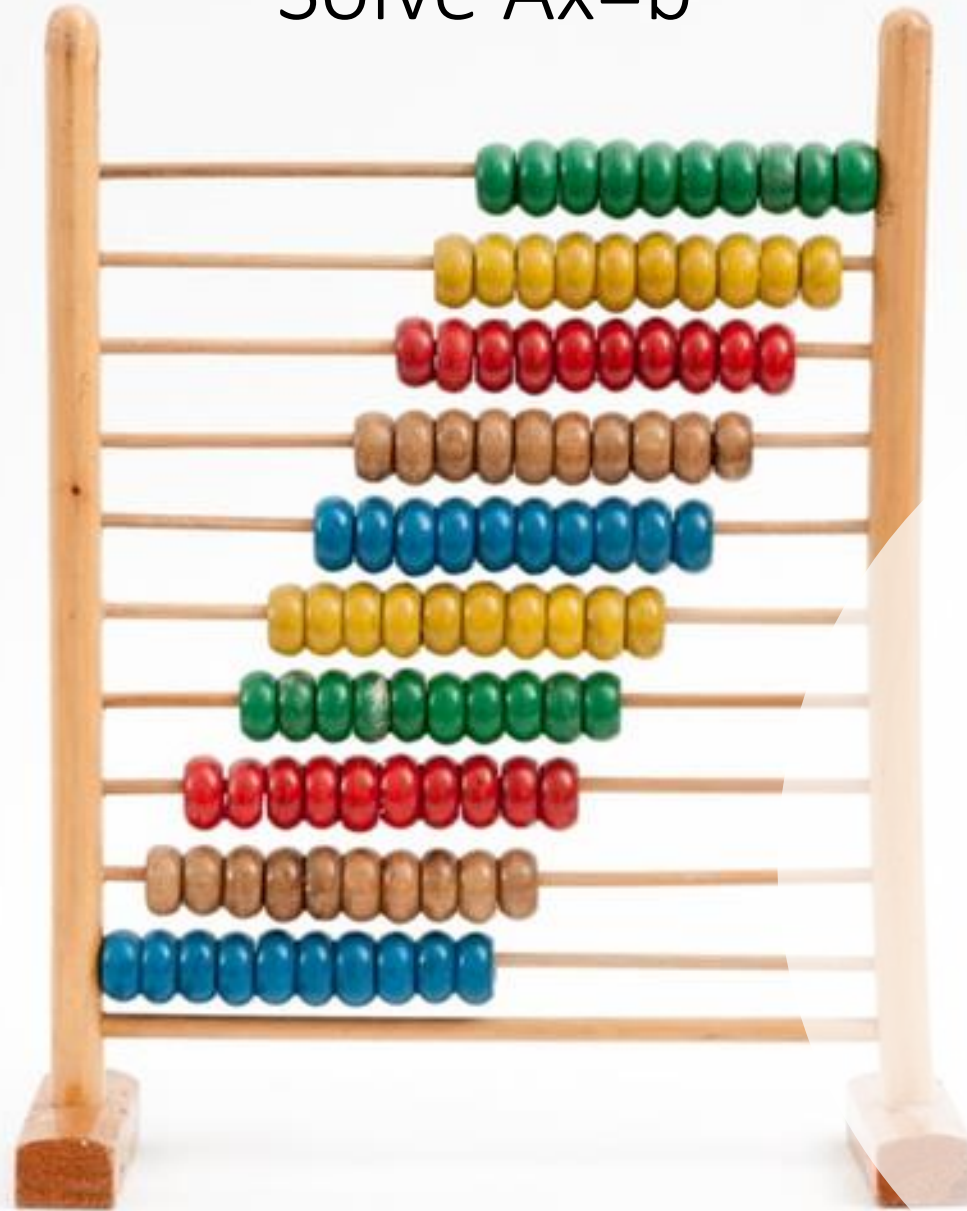
A
matrix

x
vector

b
vector



Solve $Ax=b$



Do It Again

Implementation

Python + Numba / Julia



- . N-dimensional array
- . Memory alignment

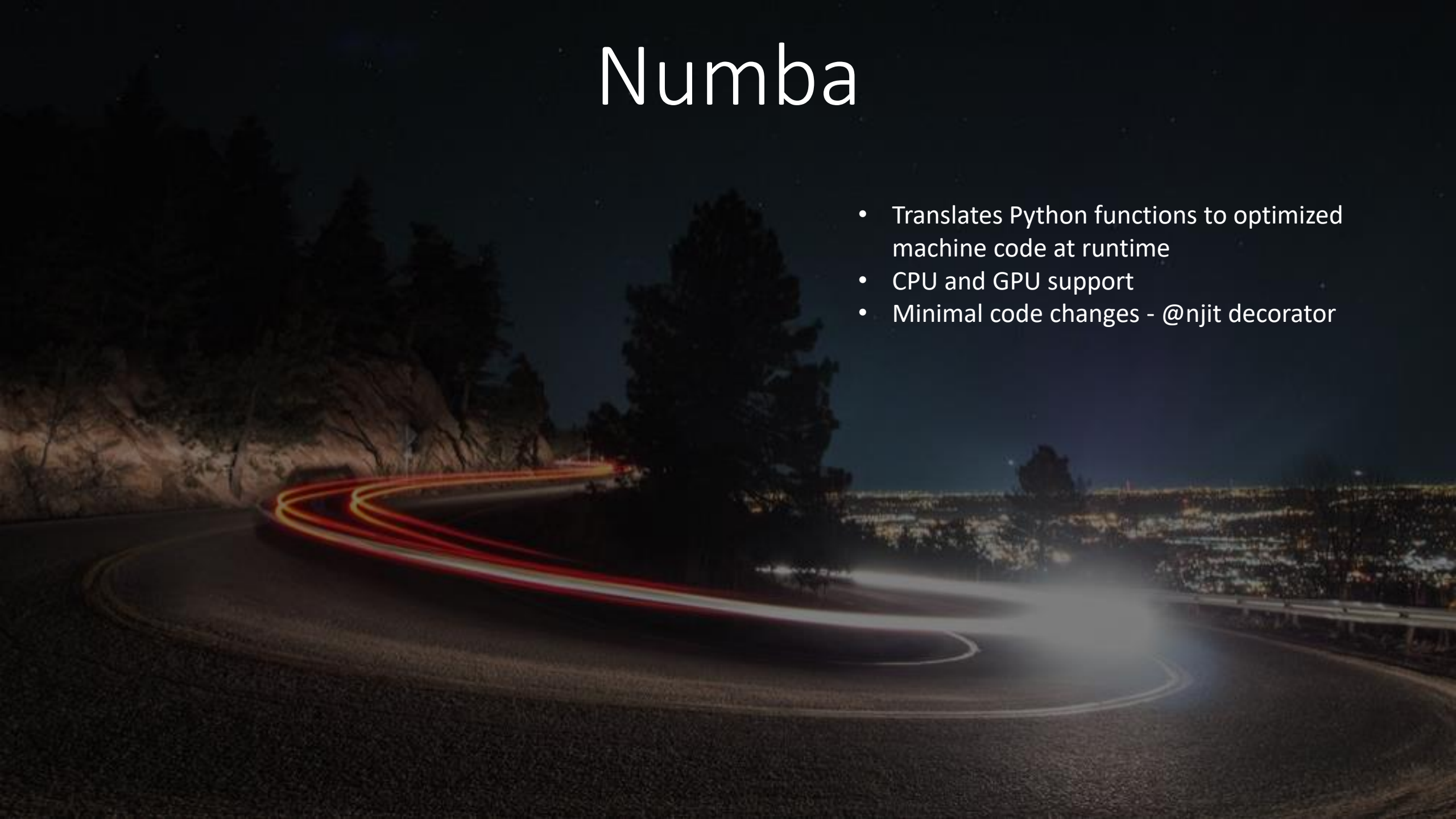


- . Optimization
- . Linear Algebra
- . Sparse Matrices
- ...

NumPy / SciPy

Numba

- Translates Python functions to optimized machine code at runtime
- CPU and GPU support
- Minimal code changes - @njit decorator



Numba

Benchmark

Apply $\sqrt{\tan(x) \cos(x)}$ on a 1-D array

	Python	Numba	Numba MT
1e6	0.295 sec	0.009 sec	0.002 sec
1e7	3.01 sec	0.069 sec	0.016 sec
1e8	29.50 sec	0.689 sec	0.165 sec
1e9	-	10.939 sec	1.681 sec

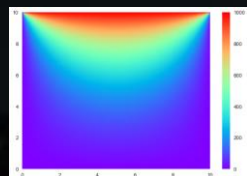
- Translates Python functions to optimized machine code at runtime
- CPU and GPU support
- Minimal code changes - @njit decorator

```
@njit(parallel=True)
def run(array):
    for i in prange(array.shape[0]):
        array[i] = function(array[i])
    return array
```


Numba

Benchmark

Solve $\nabla^2 f = 0$ on a 2-D array



	Python	Numba	Numba MT
64x64 2000 iter.	9.696 sec	0.007 sec	0.012 sec
128x128 10000 iter.	-	0.133 sec	0.131 sec
256x256 50000 iter.	-	2.459 sec	1.401 sec
512x512 250000 iter.	-	50.343 sec	19.872 sec

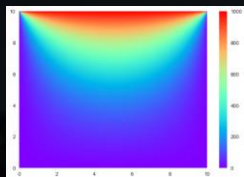
- Translates Python functions to optimized machine code at runtime
- CPU and GPU support
- Minimal code changes - @njit decorator

[illegible]

Numba

Benchmark

Solve $\nabla^2 f = 0$ on a 2-D array



Stencil Op.	Python	Numba	Numba MT
8.192 millions	9.696 sec	0.007 sec	0.012 sec
163.840 millions	-	0.133 sec	0.131 sec
3.2768 billions	-	2.459 sec	1.401 sec
65.536 billions	-	50.343 sec	19.872 sec

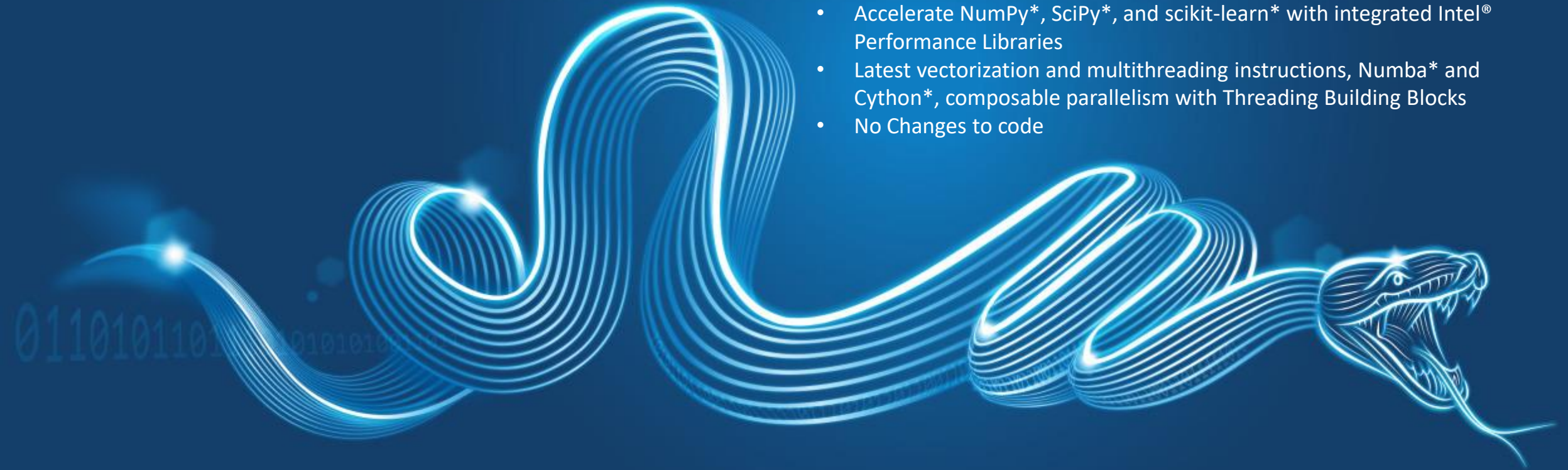
- Translates Python functions to optimized machine code at runtime
- CPU and GPU support
- Minimal code changes - @njit decorator

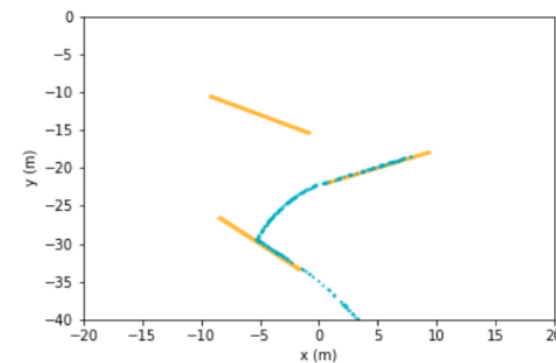
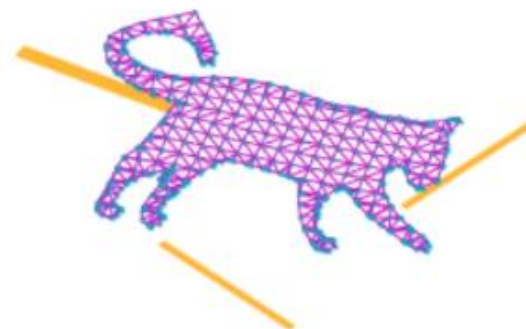
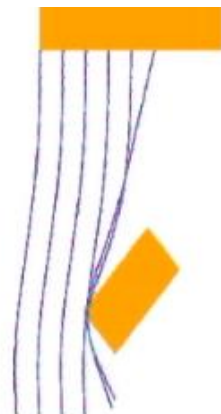
[illegible]

INTEL® DISTRIBUTION FOR PYTHON



- Accelerate NumPy*, SciPy*, and scikit-learn* with integrated Intel® Performance Libraries
- Latest vectorization and multithreading instructions, Numba* and Cython*, composable parallelism with Threading Building Blocks
- No Changes to code





Results