

# Diseño de Software para Cómputo Científico

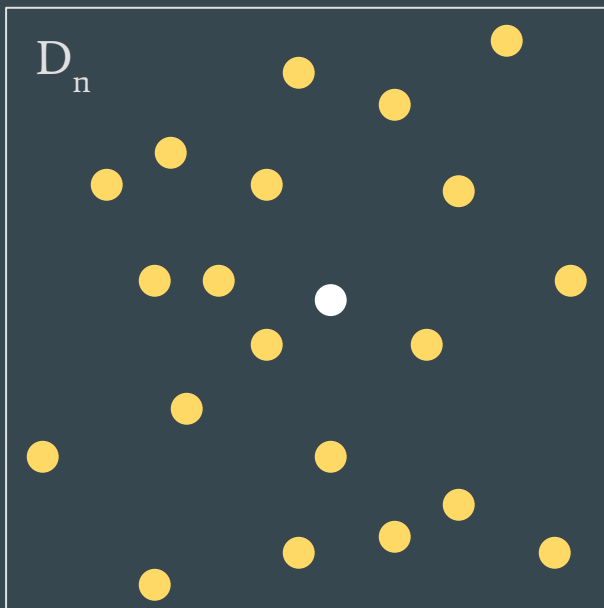
Proyecto Final: GriSPy (Grid Search in Python)



Chalela, M., Sillero, E., Pereyra, L., García, M. A., Cabral, J. B., Lares, M., & Merchán, M. (2019).

*GriSPy: A Python package for Fixed-Radius Nearest Neighbors Search.*  
*arXiv preprint arXiv:1912.09585.*

# Problema: Búsqueda de vecinos cercanos

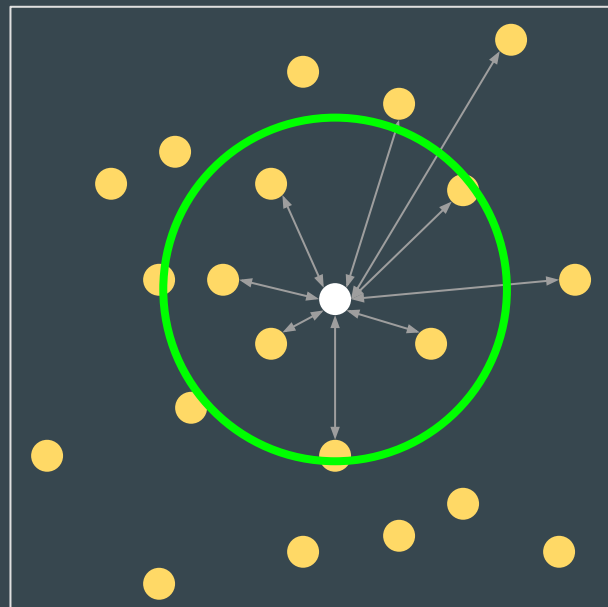


$$P = \{x \in \mathbb{R}^d \mid x \in D_n\}$$

k-esimo  
vecino de c



$$d(c, x_k) = d_k \in \{d_1, d_2, \dots, d_n\}$$



$$S = \{x_k \in D_n \mid d_k \leq r_s\}$$

# Búsqueda de vecinos más cercano (NNS)

Métodos (según su solución):

- Aproximados (Maneewong-vatana y Mount 1999):  
De interés para conjuntos de alta dimensión y recuperan puntos tal que:  
$$D(q, p) \leq (1+\epsilon) D(q, p)^*$$
donde  $D(q, p)^*$  es la distancia real y  $\epsilon$  el parámetro de incertidumbre.
- Exatos (Particionado - Indexado):
  - Binary tree (k-d tree - Friedman+ 1977 ; Bentley 1975): Dividen el espacio en dos nodos en cada iteración hasta que se alcanza un cierto número de hojas.
  - Cell techniques: Crean una red de hipercubos y cada punto se asigna a su celda. Luego usan el indexado (tabla hash) para consultas futuras.

**Objetivo del proyecto:**  
Desarrollar una paquete  
(Python) para la búsqueda de  
vecinos en  $k$ -dimensiones

$$N \times N = N^d$$

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |



$$\Delta x_d = [\max(x_d) - \min(x_d)] / N$$

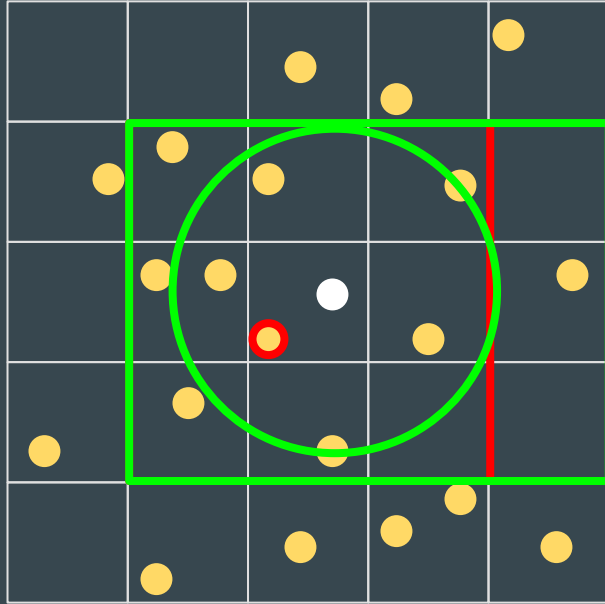
## CONSTRUCCIÓN DE LA GRILLA

- Elegir N
- Calcular  $\Delta x_d$
- Recorrer S indexando  $x_i$   

$$g_{i,d} = \text{int}(N [(x_{i,d} - \min(x_d)) / \Delta x_d])$$

- Aplanar indices  $g_{i,d}$
- Agrupar puntos por indice
- Eliminar celdas vacías
- Crear tabla hash

$$\text{Grid} = \{ (g_c) : [i, \dots, n_c] \}$$



## BÚSQUEDA DE VECINOS

- Esfera

$$S = \{x_k \in D_n \mid d_k \leq r_S\}$$

- Cascara

$$S = \{x_k \in D_n \mid r_{\min} \leq d_k \leq r_{\max}\}$$

Radios de búsqueda individuales

- k-esimos vecinos

$$d(c, x_k) = d_k \in \{d_1, d_2, \dots, d_n\}$$

# Calculo de distancias - Metricas

## Euclidean

$$D(x_0, x) = \sqrt{\sum_{i=1}^k (x_{0,i} - x_i)^2}$$

## Haversine

$$D(x_0, x) = 2 \arcsin \sqrt{A + B}$$

$$A = \sin^2\left(\frac{\Delta\phi}{2}\right)$$

$$B = \cos^2\phi_0 \cos^2\phi \sin^2\left(\frac{\Delta\lambda}{2}\right)$$

## Vincenty

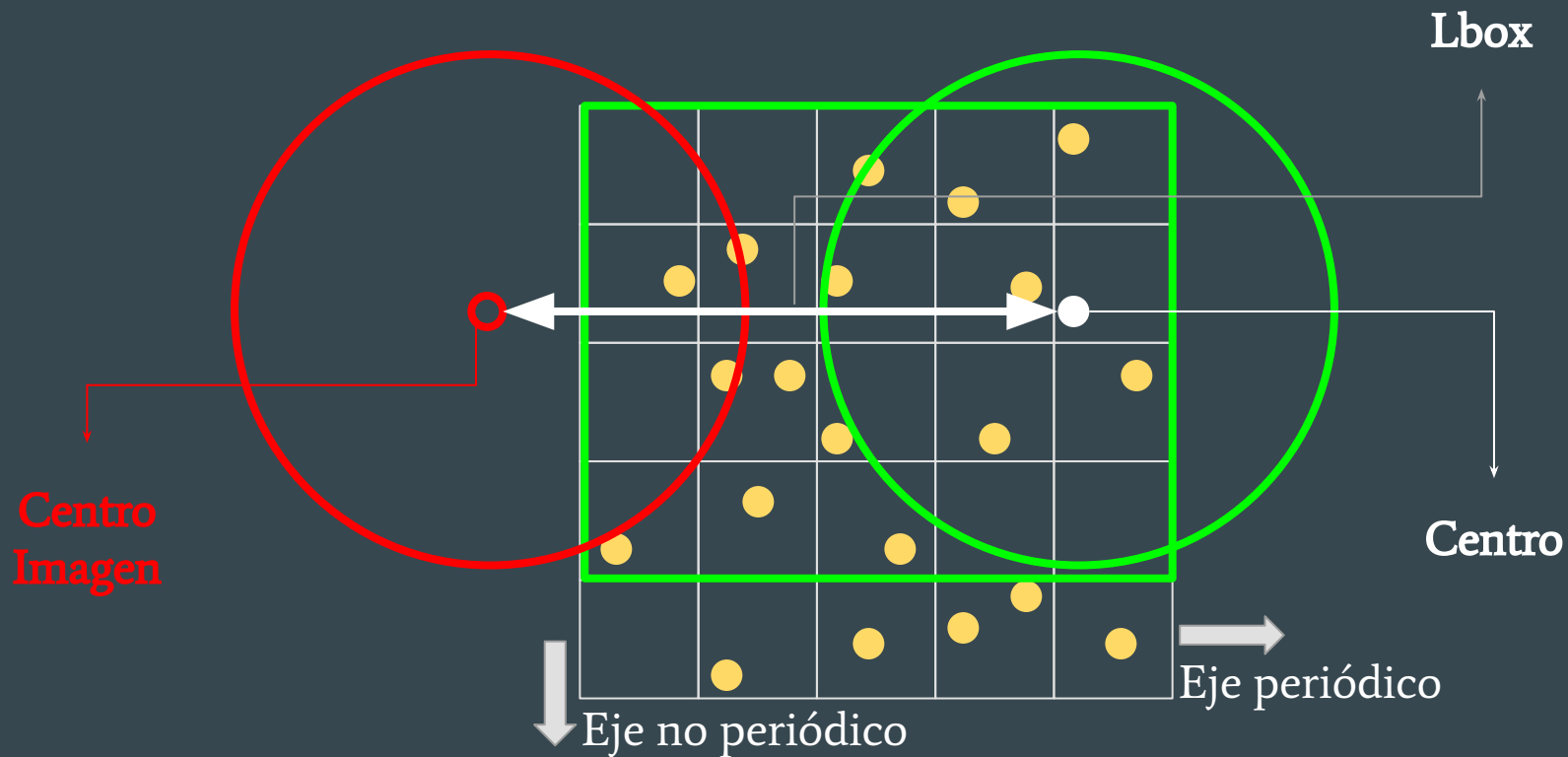
$$D(x_0, x) = \arctan \frac{\sqrt{E^2 + F^2}}{G}$$

$$E = \cos(\phi) \sin(\Delta\phi)$$

$$F = \cos(\phi_0) \sin(\phi) - \sin(\phi_0) \cos(\phi) (\Delta\lambda)$$

$$G = \sin(\phi_0) \sin(\phi) - \cos(\phi_0) \cos(\phi) (\Delta\lambda)$$

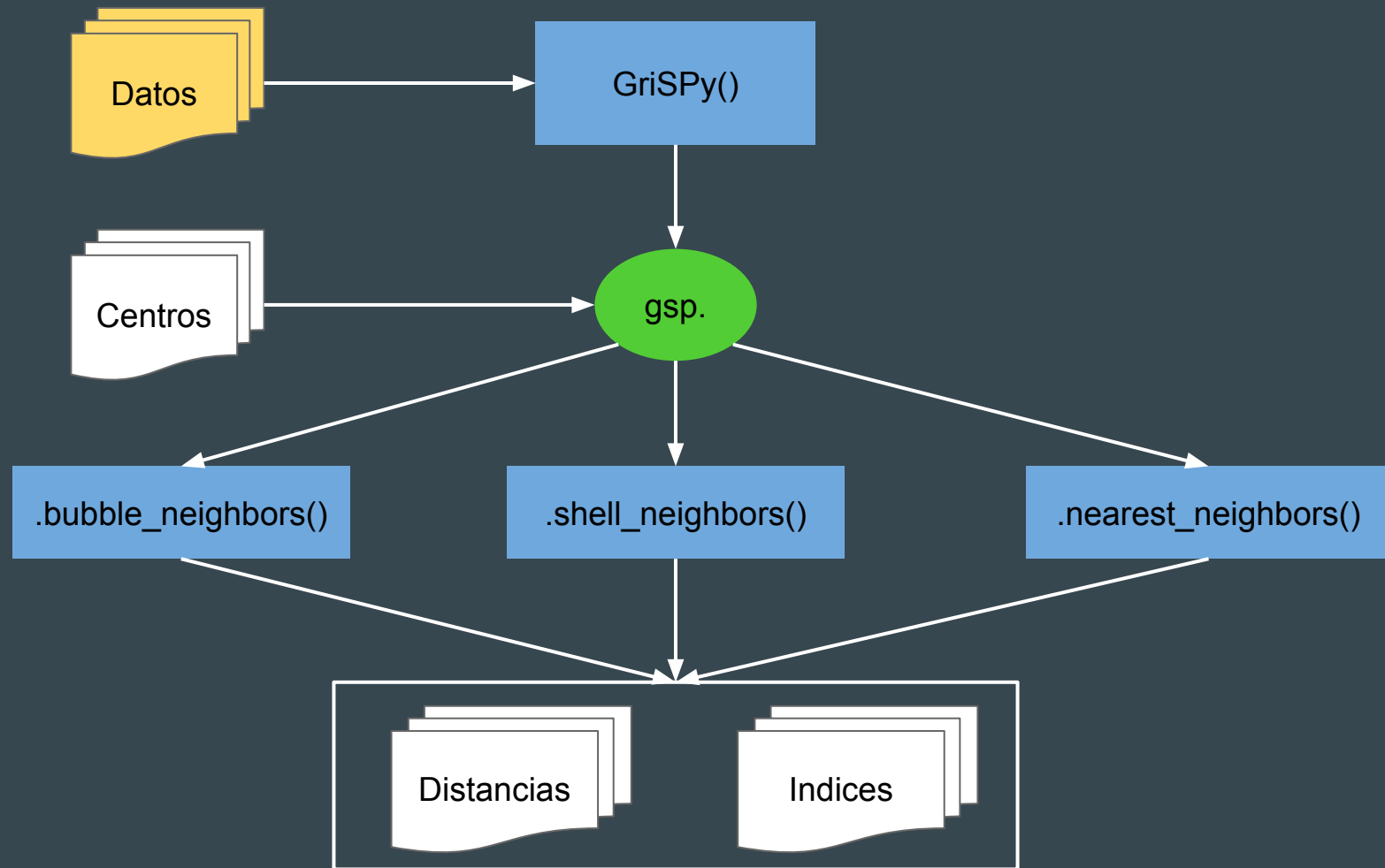
# Periodicidad





# **Desarrollo de GriSPy:**

## **Implementación interna y API**



# Dependencias:

```
13 from scipy.spatial.distance import cdist
14 import numpy as np
15 import time
16 import datetime
17 import attr
18 from . import utils
19
```

# Construcción de la Grilla:

```
21 @attr.s
22 class GriSPy(object):
23     """Grid Search in Python.
24
25     GriSPy is a regular grid search algorithm for quick nearest-neighbor
26     lookup.
27     """
28
29     # User input params
30     data = attr.ib(
31         default=None, kw_only=False, repr=False, validator=utils.validate_data,
32     )
33     N_cells = attr.ib(default=20, validator=utils.validate_N_cells)
34     periodic = attr.ib(default={}) # The validator runs in set_periodicity()
35     metric = attr.ib(default="euclid", validator=utils.validate_metric)
36     copy_data = attr.ib(
37         default=False, validator=attr.validators.instance_of(bool),
38     )
39
40     def __attrs_post_init__(self):
41         """Init more params and build the grid."""
42         if self.copy_data:
43             self.data = self.data.copy()
44             self.dim = self.data.shape[1]
45             self.set_periodicity(self.periodic)
46             self._build_grid()
47             self._empty = np.array([], dtype=int) # Useful for empty arrays
```

```

56
57 def _build_grid(self, epsilon=1.0e-6):
58     """Build the grid."""
59     t0 = time.time()
60     data_ind = np.arange(len(self.data))
61     self.k_bins = np.zeros((self.N_cells + 1, self.dim))
62     k_digit = np.zeros(self.data.shape, dtype=int)
63     for k in range(self.dim):
64         k_data = self.data[:, k]
65         self.k_bins[:, k] = np.linspace(
66             k_data.min() - epsilon,
67             k_data.max() + epsilon,
68             self.N_cells + 1,
69         )
70         k_digit[:, k] = self._digitize(k_data, bins=self.k_bins[:, k])
71
72     compact_ind = np.ravel_multi_index(
73         k_digit.T, (self.N_cells,) * self.dim, order="F",
74     )
75
76     compact_ind_sort = np.argsort(compact_ind)
77     compact_ind = compact_ind[compact_ind_sort]
78     k_digit = k_digit[compact_ind_sort]
79
80     split_ind = np.searchsorted(
81         compact_ind, np.arange(self.N_cells ** self.dim)
82     )
83     deleted_cells = np.diff(np.append(-1, split_ind)).astype(bool)
84     split_ind = split_ind[deleted_cells]
85     if split_ind[-1] > data_ind[-1]:
86         split_ind = split_ind[:-1]
87
88     list_ind = np.split(data_ind[compact_ind_sort], split_ind[1:])
89     k_digit = k_digit[split_ind]
90
91     self.grid = {}
92     for i, j in enumerate(k_digit):
93         self.grid[tuple(j)] = list(list_ind[i])
94
95     # Record date and build time
96     self.time = {"builddtime": time.time() - t0}
97     currentDT = datetime.datetime.now()
98     self.time["datetime"] = currentDT.strftime("%Y-%b-%d %H:%M:%S")

```

# Búsqueda de vecinos: bubble\_neighbors()

```
377 # User methods
378 def bubble_neighbors(self, centres, distance_upper_bound=-1.0,
379 sorted=False, kind="quicksort"):
380     """Find all points within given distances of each centre."""
381
382     # Validate inputs
383     utils.validate_centres(centres, self.data)
384     utils.validate_distance_bound(distance_upper_bound, self.periodic)
385     utils.validate_bool(sorted)
386     utils.validate_sortkind(kind)
387     # Match distance upper bound shape with centres shape
388     if np.isscalar(distance_upper_bound):
389         distance_upper_bound *= np.ones(len(centres))
390     else:
391         utils.validate_equalsize(centres, distance_upper_bound)
392
393     # Get neighbors
394     neighbor_cells = self.get_neighbor_cells(
395         centres, distance_upper_bound
396     )
397     neighbors_distances, neighbors_indices = self.get_neighbor_distance(
398         centres, neighbor_cells
399     )
400
```

```
401 # We need to generate mirror centres for periodic boundaries...
402 if self.periodic_flag:
403     terran_centres, terran_indices = self.mirror_universe(
404         centres, distance_upper_bound
405     )
406     # terran_centres are the centres in the mirror universe for those
407     # near the boundary.
408     terran_neighbor_cells = self.get_neighbor_cells(
409         terran_centres, distance_upper_bound[terran_indices]
410     )
411     terran_neighbors_distances, \
412         terran_neighbors_indices = self.get_neighbor_distance(
413         terran_centres, terran_neighbor_cells
414     )
415
416     for i, t in zip(terran_indices, np.arange(len(terran_centres))):
417         # i runs over normal indices that have a terran counterpart
418         # t runs over terran indices, 0 to len(terran_centres)
419         neighbors_distances[i] = np.concatenate(
420             (neighbors_distances[i], terran_neighbors_distances[t])
421         )
422         neighbors_indices[i] = np.concatenate(
423             (neighbors_indices[i], terran_neighbors_indices[t])
424         )
425
426 for i in range(len(centres)):
427     mask_distances = neighbors_distances[i] <= distance_upper_bound[i]
428     neighbors_distances[i] = neighbors_distances[i][mask_distances]
429     neighbors_indices[i] = neighbors_indices[i][mask_distances]
430     if sorted:
431         sorted_ind = np.argsort(neighbors_distances[i], kind=kind)
432         neighbors_distances[i] = neighbors_distances[i][sorted_ind]
433         neighbors_indices[i] = neighbors_indices[i][sorted_ind]
434
435 return neighbors_distances, neighbors_indices
```



# 1. Celdas vecinas

```
250 # Neighbor-cells methods
251 def _get_neighbor_cells(self, centres, distance_upper_bound,
252     distance_lower_bound=0, shell_flag=False):
253
254     """Retrieve cells touched by the search radius."""
255     cell_point = np.zeros((len(centres), self.dim), dtype=int)
256     out_of_field = np.zeros(len(cell_point), dtype=bool)
257     for k in range(self.dim):
258         cell_point[:, k] = (
259             self._digitize(centres[:, k], bins=self.k_bins[:, k])
260         )
261         out_of_field[
262             (centres[:, k] - distance_upper_bound > self.k_bins[-1, k])
263         ] = True
264         out_of_field[
265             (centres[:, k] + distance_upper_bound < self.k_bins[0, k])
266         ] = True
267
268     if np.all(out_of_field):
269         return [self._empty] * len(centres) # no neighbor cells
270
271     # Armo la caja con celdas a explorar
272     k_cell_min = np.zeros((len(centres), self.dim), dtype=int)
273     k_cell_max = np.zeros((len(centres), self.dim), dtype=int)
274     for k in range(self.dim):
275         k_cell_min[:, k] = (
276             self._digitize(
277                 centres[:, k] - distance_upper_bound,
278                 bins=self.k_bins[:, k],
279             )
280         )
281         k_cell_max[:, k] = (
282             self._digitize(
283                 centres[:, k] + distance_upper_bound,
284                 bins=self.k_bins[:, k],
285             )
286         )
287
288         k_cell_min[k_cell_min[:, k] < 0, k] = 0
289         k_cell_max[k_cell_max[:, k] < 0, k] = 0
290         k_cell_min[k_cell_min[:, k] >= self.N_cells, k] = self.N_cells - 1
291         k_cell_max[k_cell_max[:, k] >= self.N_cells, k] = self.N_cells - 1
292
293     cell_size = self.k_bins[1, :] - self.k_bins[0, :]
294     cell_radii = 0.5 * np.sum(cell_size ** 2) ** 0.5
295
296     neighbor_cells = []
297     for i in range(len(centres)):
298         # Para cada centro i, agrego un arreglo con shape (:,k)
299         k_grids = [
300             np.arange(k_cell_min[i, k], k_cell_max[i, k] + 1)
301             for k in range(self.dim)
302         ]
303         k_grids = np.meshgrid(*k_grids)
304         neighbor_cells += [
305             np.array(list(map(np.ndarray.flatten, k_grids))).T
306         ]
307
308         # Calculo la distancia de cada centro i a sus celdas vecinas,
309         # luego descarto las celdas que no toca el circulo definido por
310         # la distancia
311         cells_physical = [
312             self.k_bins[neighbor_cells[i][:, k], k] + 0.5 * cell_size[k]
313             for k in range(self.dim)
314         ]
315         cells_physical = np.array(cells_physical).T
316         mask_cells = (
317             self._distance(
318                 centres[i], cells_physical
319             ) < distance_upper_bound[i] + cell_radii
320         )
321
322         if shell_flag:
323             mask_cells *= (
324                 self._distance(
325                     centres[i], cells_physical
326                 ) > distance_lower_bound[i] - cell_radii
327             )
328
329         if np.any(mask_cells):
330             neighbor_cells[i] = neighbor_cells[i][mask_cells]
331         else:
332             neighbor_cells[i] = self._empty
333     return neighbor_cells
334
```

## 2. Distancias a los vecinos

```
227
228 def _get_neighbor_distance(self, centres, neighbor_cells):
229     """Retrieve neighbor distances within the given cells."""
230     neighbors_indices = []
231     neighbors_distances = []
232     for i in range(len(centres)):
233         if len(neighbor_cells[i]) == 0: # no hay celdas vecinas
234             neighbors_indices += [self._empty]
235             neighbors_distances += [self._empty]
236             continue
237         # Genera una lista con los vecinos de cada celda
238         # print neighbor_cells[i]
239         ind_tmp = []
240         self.grid.get(tuple(neighbor_cells[i][j]), [])
241         for j in range(len(neighbor_cells[i])):
242             # Une en una sola lista todos sus vecinos
243             neighbors_indices += [np.concatenate(ind_tmp).astype(int)]
244             neighbors_distances += [
245                 self._distance(centres[i], self.data[neighbors_indices[i], :])
246             ]
247     return neighbors_distances, neighbors_indices
248
249
```

```
184 def _distance(self, centre_0, centres):
185     """Compute distance between points."""
186
187     if len(centres) == 0:
188         return self._empty
189
190     if self.metric == "euclid":
191         c0 = centre_0.reshape((-1, self.dim))
192         d = cdist(c0, centres).reshape((-1,))
193         return d
194
195     elif self.metric == "haversine":
196         lon1 = np.deg2rad(centre_0[0])
197         lat1 = np.deg2rad(centre_0[1])
198         lon2 = np.deg2rad(centres[:, 0])
199         lat2 = np.deg2rad(centres[:, 1])
200
201         sdlon = np.sin((lon2 - lon1) / 2.)
202         sdlat = np.sin((lat2 - lat1) / 2.)
203         clat1 = np.cos(lat1)
204         clat2 = np.cos(lat2)
205         num1 = sdlat ** 2
206         num2 = clat1 * clat2 * sdlon ** 2
207         sep = 2 * np.arcsin(np.sqrt(num1 + num2))
208         return np.rad2deg(sep)
209
210     elif self.metric == "vincenty":
211         lon1 = np.deg2rad(centre_0[0])
212         lat1 = np.deg2rad(centre_0[1])
213         lon2 = np.deg2rad(centres[:, 0])
214         lat2 = np.deg2rad(centres[:, 1])
215
216         sdlon = np.sin(lon2 - lon1)
217         cdlon = np.cos(lon2 - lon1)
218         slat1 = np.sin(lat1)
219         slat2 = np.sin(lat2)
220         clat1 = np.cos(lat1)
221         clat2 = np.cos(lat2)
222         num1 = clat2 * sdlon
223         num2 = clat1 * slat2 - slat1 * clat2 * cdlon
224         denominator = slat1 * slat2 + clat1 * clat2 * cdlon
225         sep = np.arctan2(np.sqrt(num1 ** 2 + num2 ** 2), denominator)
226         return np.rad2deg(sep)
227
```



# 3. Periodicidad

```
335 def _near_boundary(self, centres, distance_upper_bound):
336     mask = np.zeros((len(centres), self.dim), dtype=bool)
337     for k in range(self.dim):
338         if self.periodic[k] is None:
339             continue
340         mask[:, k] = abs(
341             centres[:, k] - self.periodic[k][0]
342         ) < distance_upper_bound
343         mask[:, k] += abs(
344             centres[:, k] - self.periodic[k][1]
345         ) < distance_upper_bound
346     return mask.sum(axis=1, dtype=bool)
347
348 def _mirror(self, centre, distance_upper_bound):
349     mirror_centre = centre - self._periodic_edges
350     mask = self._periodic_dirs * distance_upper_bound
351     mask += mirror_centre
352     mask = (mask >= self._pd_low) * (mask <= self._pd_hi)
353     mask = np.prod(mask, 1, dtype=bool)
354     return mirror_centre[mask]
355
356 def _mirror_universe(self, centres, distance_upper_bound):
357     """Generate Terran centres in the Mirror Universe."""
358     terran_centres = np.array([[]] * self.dim).T
359     terran_indices = np.array([], dtype=int)
360     near_boundary = self._near_boundary(centres, distance_upper_bound)
361     if not np.any(near_boundary):
362         return terran_centres, terran_indices
363
364     for i, centre in enumerate(centres):
365         if not near_boundary[i]:
366             continue
367         mirror_centre = self._mirror(centre, distance_upper_bound[i])
368         if len(mirror_centre) > 0:
369             terran_centres = np.concatenate(
370                 (terran_centres, mirror_centre), axis=0
371             )
372             terran_indices = np.concatenate(
373                 (terran_indices, np.repeat(i, len(mirror_centre)))
374             )
375     return terran_centres, terran_indices
---
```

# shell\_neighbors()

```
436 def shell_neighbors(self, centres, distance_lower_bound=-1.0,
437 distance_upper_bound=-1.0, sorted=False, kind="quicksort"):
438     """Find all points within given lower and upper distances of each centre."""
439
440     # Validate inputs
441     utils.validate_centres(centres, self.data)
442     utils.validate_bool(sorted)
443     utils.validate_sortkind(kind)
444     utils.validate_shell_distances(
445         distance_lower_bound, distance_upper_bound, self.periodic,
446     )
447
448     # Match distance bounds shapes with centres shape
449     if np.isscalar(distance_lower_bound):
450         distance_lower_bound *= np.ones(len(centres))
451     else:
452         utils.validate_equalsize(centres, distance_lower_bound)
453     if np.isscalar(distance_upper_bound):
454         distance_upper_bound *= np.ones(len(centres))
455     else:
456         utils.validate_equalsize(centres, distance_upper_bound)
457
458     # Get neighbors
459     neighbor_cells = self._get_neighbor_cells(
460         centres,
461         distance_upper_bound=distance_upper_bound,
462         distance_lower_bound=distance_lower_bound,
463         shell_flag=True,
464     )
465     neighbors_distances, neighbors_indices = self._get_neighbor_distance(
466         centres, neighbor_cells)
467
468     # We need to generate mirror centres for periodic boundaries...
469     if self.periodic_flag:
470         terran_centres, terran_indices = self._mirror_universe(
471             centres, distance_upper_bound)
472         # terran centres are the centres in the mirror universe for those
473         # near the boundary.
474         terran_neighbor_cells = self._get_neighbor_cells(
475             terran_centres, distance_upper_bound[terran_indices])
476
477         terran_neighbors_distances, \
478             terran_neighbors_indices = self._get_neighbor_distance(
479             terran_centres, terran_neighbor_cells)
480
481         for i, t in zip(terran_indices, np.arange(len(terran_centres))):
482             # i runs over normal indices that have a terran counterpart
483             # t runs over terran indices, 0 to len(terran_centres)
484             neighbors_distances[i] = np.concatenate(
485                 (neighbors_distances[i], terran_neighbors_distances[t]) )
486             neighbors_indices[i] = np.concatenate(
487                 (neighbors_indices[i], terran_neighbors_indices[t]) )
488
489     for i in range(len(centres)):
490         mask_distances_upper = neighbors_distances[i] <= distance_upper_bound[i]
491         mask_distances_lower = neighbors_distances[i][mask_distances_upper]
492         mask_distances_lower = mask_distances_lower > distance_lower_bound[i]
493
494         aux = neighbors_distances[i]
495         aux = aux[mask_distances_upper]
496         aux = aux[mask_distances_lower]
497         neighbors_distances[i] = aux
498
499         aux = neighbors_indices[i]
500         aux = aux[mask_distances_upper]
501         aux = aux[mask_distances_lower]
502         neighbors_indices[i] = aux
503
504     if sorted:
505         sorted_ind = np.argsort(neighbors_distances[i], kind=kind)
506         neighbors_distances[i] = neighbors_distances[i][sorted_ind]
507         neighbors_indices[i] = neighbors_indices[i][sorted_ind]
508
509     return neighbors_distances, neighbors_indices
510
```

# nearest\_neighbors( )

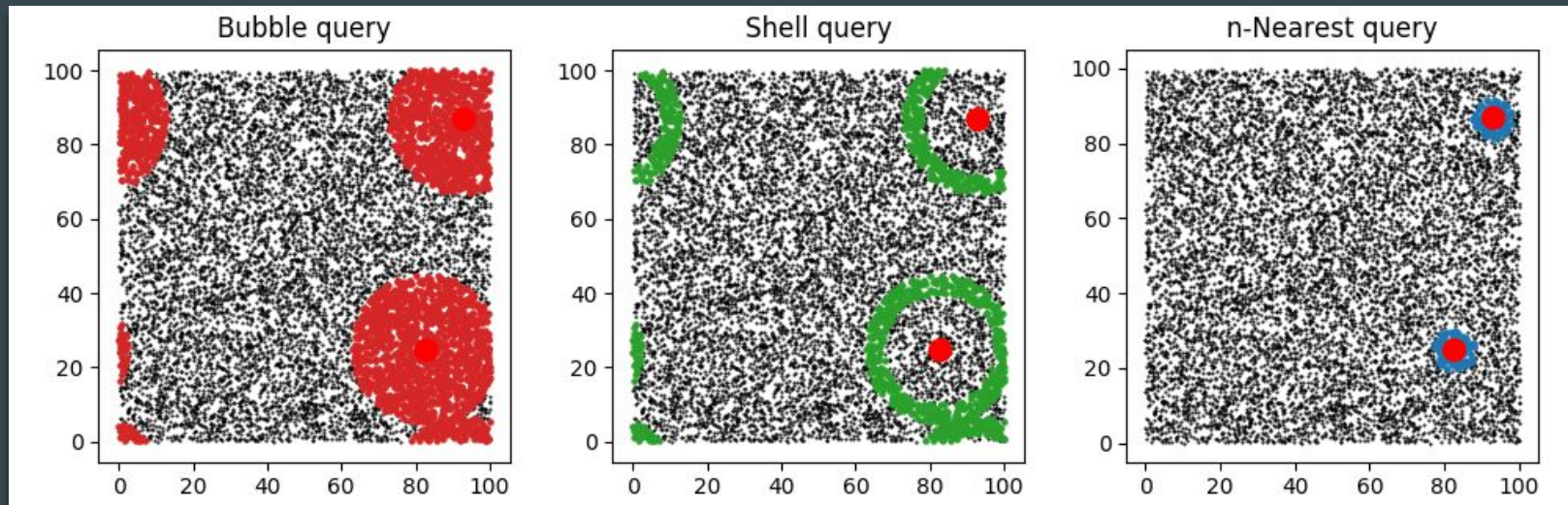
```
511 def nearest_neighbors(self, centres, n=1, kind="quicksort"):
512     """Find the n nearest-neighbors for each centre."""
513
514     # Validate input
515     utils.validate_centres(centres, self.data)
516     utils.validate_n_nearest(n, self.data, self.periodic)
517     utils.validate_sortkind(kind)
518
519     # Initial definitions
520     N_centres = len(centres)
521     centres_lookup_ind = np.arange(0, N_centres)
522     n_found = np.zeros(N_centres, dtype=bool)
523     lower_distance_tmp = np.zeros(N_centres)
524     upper_distance_tmp = np.zeros(N_centres)
525
526     # Abro la celda del centro como primer paso
527     centre_cell = self.get_neighbor_cells(
528         centres, distance_upper_bound=upper_distance_tmp
529     )
530     # crear funcion que regrese vecinos sin calcular distancias
531     neighbors_distances, neighbors_indices = self.get_neighbor_distance(
532         centres, centre_cell
533     )
534
535     # Calculo una primera aproximacion con la
536     # 'distancia media' = 0.5 * (n/density)**(1/dim)
537     # Factor de escala para la distancia inicial
538     mean_distance_factor = 1.0
539     cell_size = self.k_bins[1, :] - self.k_bins[0, :]
540     cell_volume = np.prod(cell_size.astype(float))
541     neighbors_number = np.array(list(map(len, neighbors_indices)))
542     mask_zero_neighbors = neighbors_number == 0
543     neighbors_number[mask_zero_neighbors] = 1
544     mean_distance = 0.5 * (n / (neighbors_number / cell_volume)) ** (
545         1.0 / self.dim)
546
547     upper_distance_tmp = mean_distance_factor * mean_distance
548
549     neighbors_indices = [self.empty] * N_centres
550     neighbors_distances = [self.empty] * N_centres
551     while not np.all(n_found):
552         neighbors_distances_tmp, \
553             neighbors_indices_tmp = self.shell_neighbors(
554                 centres[-n_found],
555                 distance_lower_bound=lower_distance_tmp[-n_found],
556                 distance_upper_bound=upper_distance_tmp[-n_found],
557             )
558
559     for i_tmp, i in enumerate(centres_lookup_ind[-n_found]):
560         if n_found[i]:
561             continue
562         if n <= len(neighbors_indices_tmp[i_tmp]) + len(
563             neighbors_indices[i]
564         ):
565             n_more = n - len(neighbors_indices[i])
566             n_found[i] = True
567         else:
568             n_more = len(neighbors_indices_tmp[i_tmp])
569             lower_distance_tmp[i_tmp] = upper_distance_tmp[
570                 i_tmp
571             ].copy()
572             upper_distance_tmp[i_tmp] += cell_size.min()
573
574     sorted_ind = np.argsort(
575         neighbors_distances_tmp[i_tmp], kind=kind
576     )[:n_more]
577     neighbors_distances[i] = np.hstack(
578         (
579             neighbors_distances[i],
580             neighbors_distances_tmp[i_tmp][sorted_ind],
581         )
582     )
583     neighbors_indices[i] = np.hstack(
584         (
585             neighbors_indices[i],
586             neighbors_indices_tmp[i_tmp][sorted_ind],
587         )
588     )
589
590     return neighbors_distances, neighbors_indices
```

# Save & Load

```
652 def save_grid(self, file="grispy.gsp", overwrite=False):
653     """Save all grid attributes in a binary file for future use."""
654
655     # Validate input
656     utils.validate_filename(file)
657     utils.validate_bool(overwrite)
658     utils.validate_canwrite(file, overwrite)
659
660     import pickle
661     with open(file, "wb") as fp:
662         pickle.dump(self, fp, protocol=pickle.HIGHEST_PROTOCOL)
663
664     print("GriSPy grid attributes saved to: {}".format(file))
665     return None
666
667 @classmethod
668 def load_grid(cls, file):
669     """Load a GriSPy instance previously saved with the save_grid() method."""
670
671     # Validate input
672     utils.validate_filename(file)
673
674     import os.path
675     if not os.path.isfile(file):
676         raise FileNotFoundError("There is no file named {}".format(file))
677
678     import pickle
679     with open(file, "rb") as fp:
680         gsp = pickle.load(fp)
681         if not isinstance(gsp, cls):
682             raise TypeError("Unpickled object is not a GriSPy instance.")
683
684     print(
685         "Successfully loaded GriSPy grid created on {}".format(
686             gsp.time["datetime"]
687         )
688     )
689     return gsp
690
```



# Ejemplo



**Calidad:**

**Testing e integración continua**

# Testing

Se divide en cuatro áreas:

- Core (13)
- Distancias (12)
- Consistencia de datos (25)
- I/O (7)

# Testing

Core:

- GriSPy (8)
  - Orden correcto del vector de distancias en `nearest_neighbors`.
  - Puntos dentro de la burbuja en `bubble_neighbors`.
  - Puntos dentro del rango en `shell_neighbors`.
  - Precisión para todos los métodos de búsqueda de vecinos.
- Periodicidad (2)
  - Control para `bubble_neighbors` y `shell_neighbors`.
- Hiperesfera (3)
  - Pruebas de cálculo para todos los métodos de búsqueda de vecinos.



# Testing

Distancias:

Se ejecuta un test de cada condición para cada una de las métricas (`euclid`, `vincenty` y `haversine`).

- Condición I (3)
  - $\text{dist}(a, b) \neq \text{NaN}$
- Condición II (3)
  - $0 \leq \text{dist}(a, b)$
- Condición III (3)
  - $\text{dist}(a, b) = \text{dist}(b, a)$
- Condición IV (3)
  - $\text{dist}(a, c) \leq \text{dist}(a, b) + \text{dist}(b, c)$

# Testing

## Consistencia de datos:

- Datos retornados (9)
  - Tipo y tamaño de los datos retornados para todos los métodos de búsqueda de vecinos (un centro y múltiples centros).
  - Periodicidad. Tipo y tamaño de datos retornados por `mirror_universe`, `_mirror` y `_near_boundary`.
- Inicialización (10)
  - Tipo, valor y estructura de data.
  - Tipo y valor de `periodic`, `metric`, `copy_data`, y `N_cells`.
- Inputs de queries (6)
  - En `bubble_neighbors`, tipo y forma forma de centros y radios, valor y tipo de `kind` y `sorted`.
  - En `shell_neighbors`, tipo y forma forma de centros y radios.
  - En `nearest_neighbors`, tipo y valor de `n`.

# Coverage: 99%

```
attrs==19.1.0, backcall==0.1.0, coverage==4.5.4, cyclo==0.10.0, decorator==4.4.1, filelock==3.0.12, grispy==2019.12, importlib-
metadata==0.23, ipdb==0.12.2, ipython==7.10.1, ipython-genutils==0.2.0, jedi==0.15.1, kiwisolver==1.1.0, matplotlib==3.1.1, more-
itertools==8.0.0, numpy==1.17.2, packaging==19.2, parso==0.5.1, pexpect==4.7.0, pickleshare==0.7.5, pluggy==0.13.1, prompt-
toolkit==3.0.2, pyprocess==0.6.0, py==1.8.0, Pygments==2.5.2, pyparsing==2.4.5, pytest==5.3.1, pytest-cov==2.8.1, python-
dateutil==2.8.1, scipy==1.3.1, six==1.13.0, toml==0.10.0, tox==3.14.0, traitlets==4.3.3, virtualenv==16.7.8, wcwidth==0.1.7, zipp==0.6.0
215 coverage run-test-pre: PYTHONHASHSEED='146659337'
216 coverage run-test: commands[0] | - coverage erase
217 coverage run-test: commands[1] | pytest -q tests/ --cov=grispy/ --cov-append --cov-report=
218 ..... [100%]
219
220
221 57 passed in 2.19s
222 coverage run-test: commands[2] | coverage report --fail-under=80 -m
223 Name Stmts Miss Cover Missing
224 -----
225 grispy/_init_.py 3 0 100%
226 grispy/core.py 316 1 99% 675
227 grispy/utis.py 107 1 99% 118
228 -----
229 TOTAL 426 2 99%
230 _____ summary _____
231 coverage: commands succeeded
232 congratulations :)
233 The command "tox -r" exited with 0.
234
235 $ bash <(curl -s https://codecov.io/bash)
236 Done. Your build exited with 0.
```



Travis CI

©TRAVIS CI, GMBH

Rigaer Straße 8  
10247 Berlin, Germany  
Work with Travis CI

HELP

Documentation  
Community  
Changelog

LEGAL

Imprint  
Terms of Service  
Privacy Policy

TRAVIS CI STATUS

Travis CI Status

# Integración continua con Travis CI

Travis CI

Dashboard Changelog Documentation Help

Search all repositories

My Repositories +

✓ mchalela/GriSPy # 37

Duration: 3 min 57 sec

Finished: a day ago

mchalela / GriSPy

build: passing

Current Branches Build History Pull Requests

More options

✓ master fixed testenv:docs in tox

Commit 7864321

Compare 792af88...7864321

Branch master

Emanuel Sillero

→ #37 passed

Ran for 1 min 11 sec

Total time 3 min 57 sec

a day ago

Restart build

Build jobs View config

|          |       |                 |                 |        |  |
|----------|-------|-----------------|-----------------|--------|--|
| ✓ # 37.1 | AMD64 | </> Python: 3.7 | TOXENV=style    | 22 sec |  |
| ✓ # 37.2 | AMD64 | </> Python: 3.7 | TOXENV=coverage | 42 sec |  |
| ✓ # 37.3 | AMD64 | </> Python: 3.6 | TOXENV=py36     | 51 sec |  |
| ✓ # 37.4 | AMD64 | </> Python: 3.7 | TOXENV=py37     | 54 sec |  |
| ✓ # 37.5 | AMD64 | </> Python: 3.7 | TOXENV=docstyle | 40 sec |  |
| ✓ # 37.6 | AMD64 | </> Python: 3.7 | TOXENV=docs     | 28 sec |  |

**Optimización:**  
Profiling, paralelización...

# Benchmarking (Medición de tiempos)

## Métodos

El código puede separarse en dos partes

- Build
- Query

## Objetivo

Encontrar los mejores parámetros que minimicen los tiempos Build y Query.

Además entender las limitaciones de nuestro código (Peores Casos)

# Parámetros libres del problema

- N - Celdas
- N - Puntos
- N - Centros
- K - dimensiones
- Radio de búsqueda
- Métrica
- Condiciones Periódicas
- El grado de clustering de los datos

# Parámetros libres del problema

- N - Celdas
- N - Puntos
- N - Centros
- K - dimensiones
- Radio de búsqueda
- Métrica
- Condiciones Periódicas
- El grado de clustering de los datos

**fracción del Box**

**Euclidiana**

**No periódicas**

**Uniforme**



# Variando K-dim

## [2 - 3 - 4]

$$L_{\text{box}} = [0.0, 1.0]$$

$$\text{Núm. de centros} - N_{\text{cent}} = 10^3$$

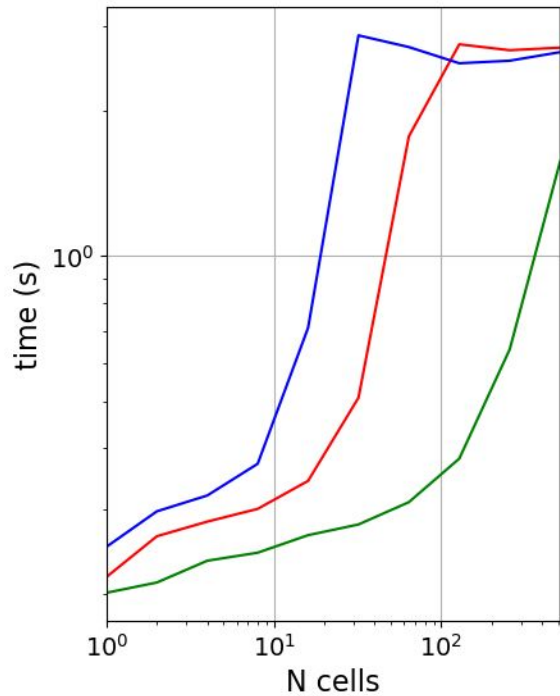
$$\text{Núm. de puntos} - N = 10^6$$

Núm. de celdas -

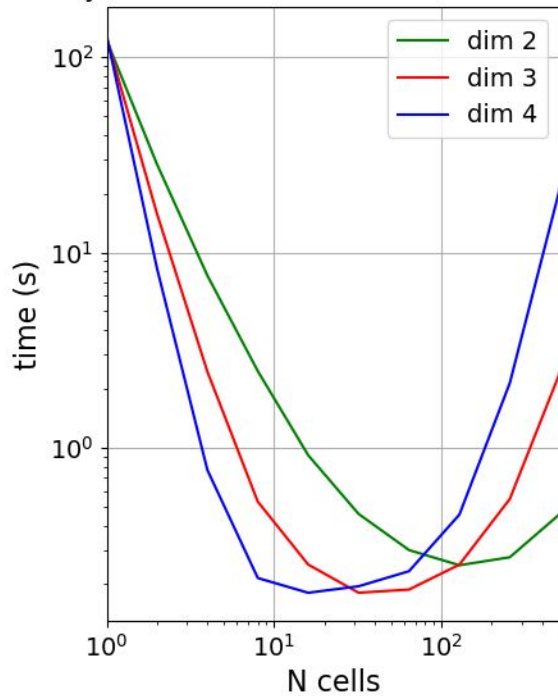
$$\underline{\quad [1, 2, 4, 8, 16, 32, 64, 128, 264, 512]}$$

# UNIFORME

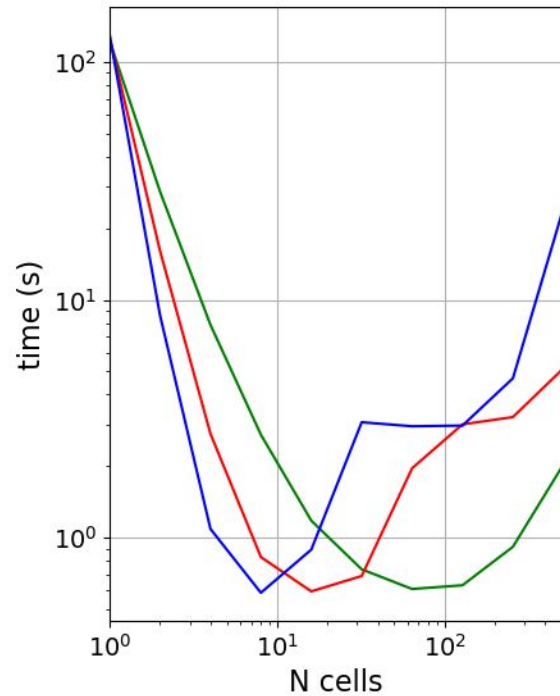
Build Time



Query Time ( $N_{\text{cent}} = 10^3$ ,  $r = 0.01$ ,  $N = 10^6$ )



Total Time



Variando N  
[ $10^3, 10^4, 10^5, 10^6$ ]

$$L_{\text{box}} = [0.0, 1.0]$$

$$\text{Núm. de centros} - N_{\text{cent}} = 10^3$$

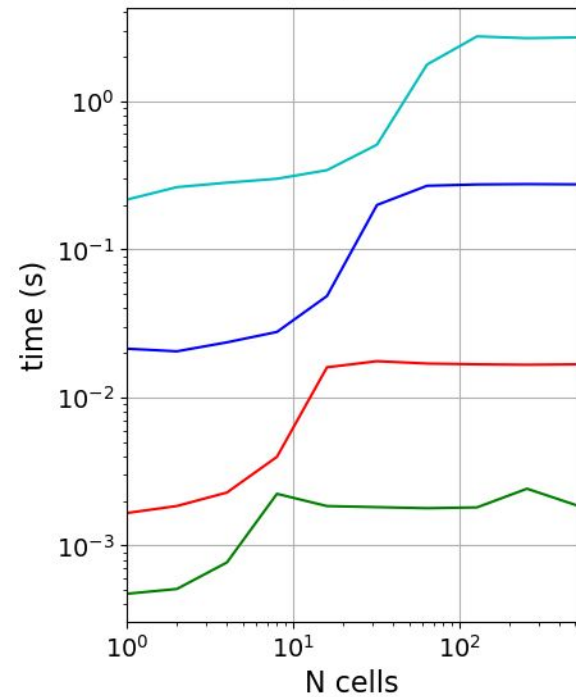
$$\text{Núm. de dimensiones} - K = 3$$

$$\text{Núm. de celdas} -$$

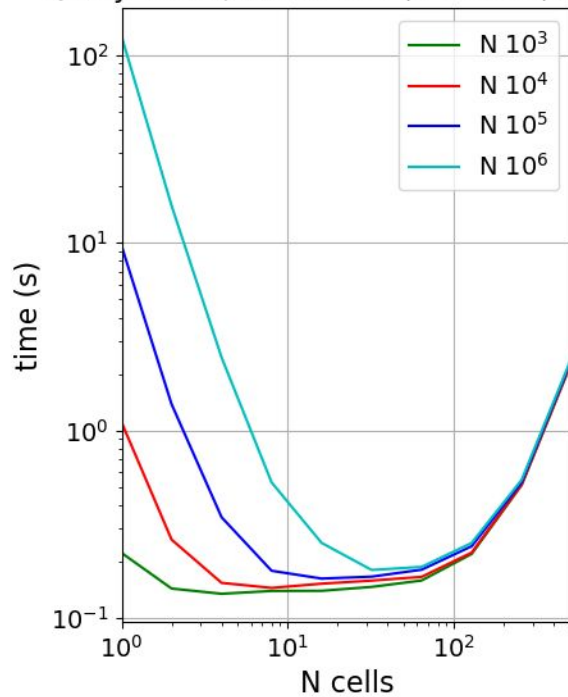
$$\underline{\hspace{1cm}} [1, 2, 4, 8, 16, 32, 64, 128, 264, 512]$$

# UNIFORME

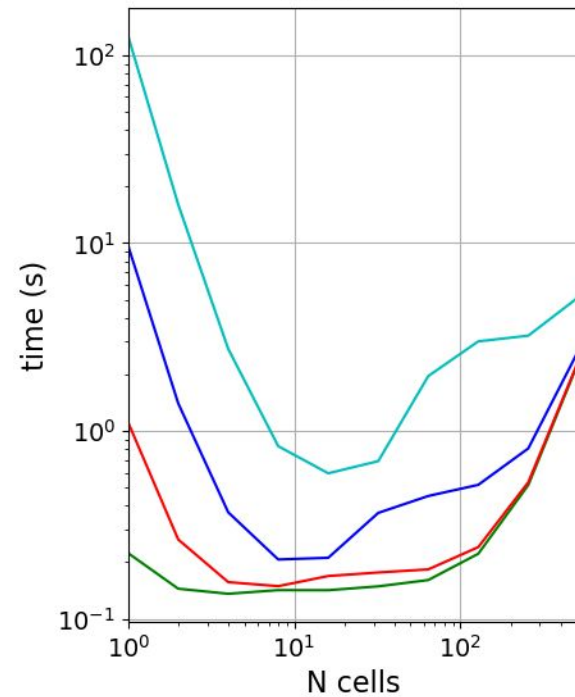
Build Time



Query Time ( $N_{\text{cent}} = 10^3$ ,  $r = 0.01$ , dim 3)



Total Time



# Variando Ncentros [ $10^3, 10^4, 10^5, 10^6$ ]

$$L_{\text{box}} = [0.0, 1.0]$$

$$\text{Núm. de centros} - N = 10^6$$

$$\text{Núm. de dimensiones} - K = 3$$

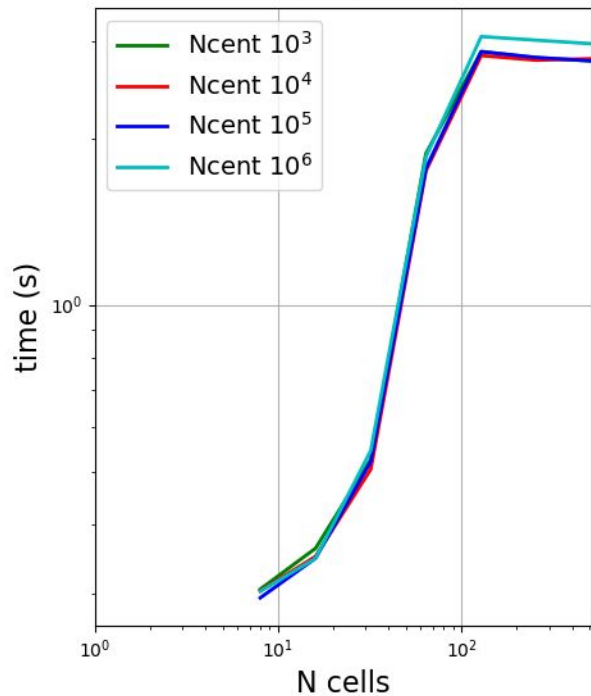
$$\text{Núm. de celdas} -$$

---

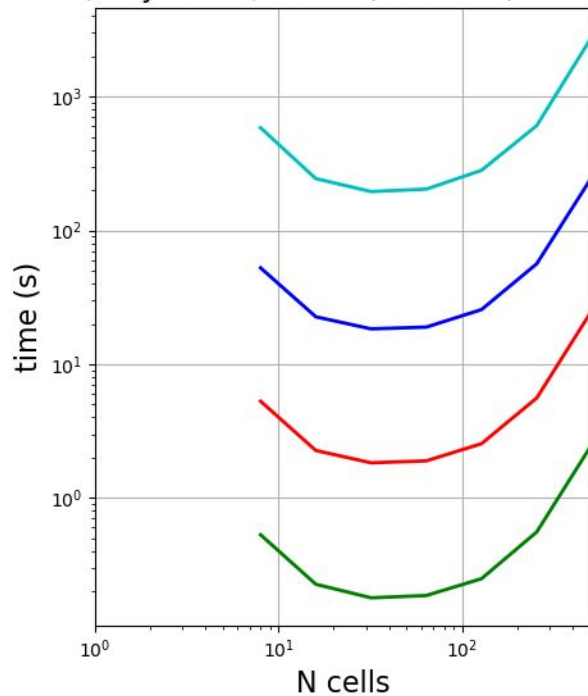
$$[8, 16, 32, 64, 128, 264, 512]$$

# UNIFORME

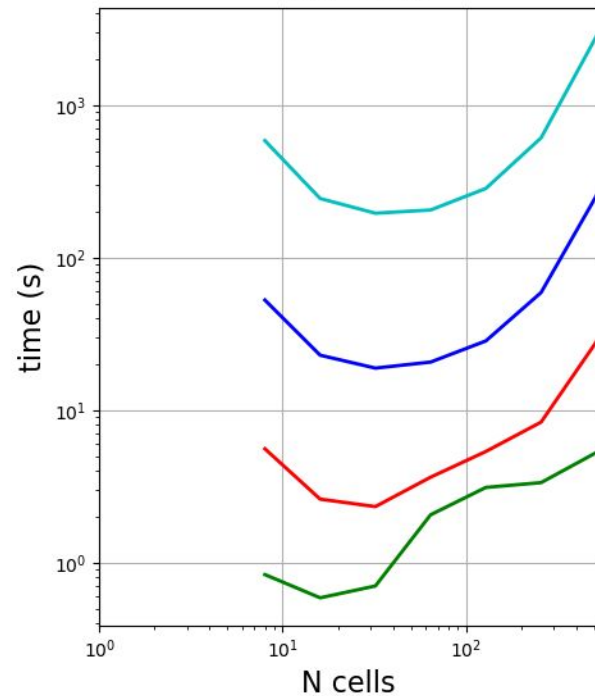
Build Time



Query Time (N =  $10^6$ , r = 0.01, dim 3)



Total Time



# Pruebas con Clustering (datos reales)

Variando N  
[ $10^3$ ,  $10^4$ ,  $10^5$ ,  $10^6$ ]

$$L_{\text{box}} = [0.0, 1.0]$$

$$\text{Núm. de centros} - N_{\text{cent}} = 10^3$$

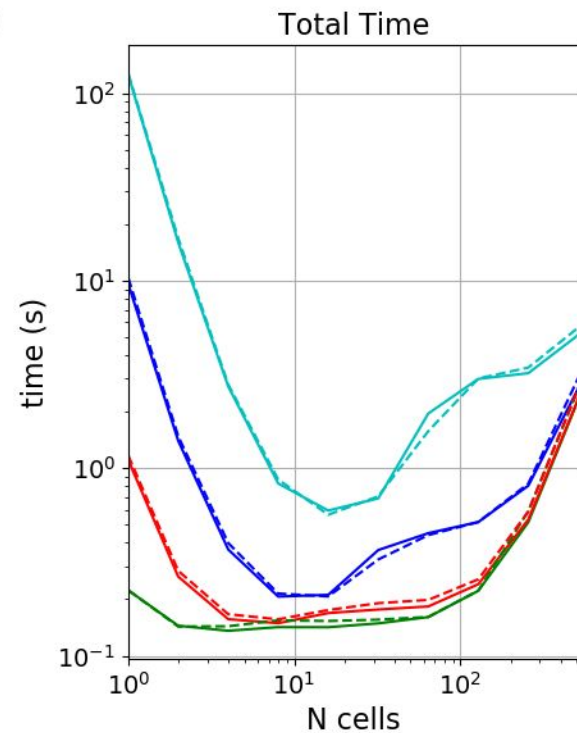
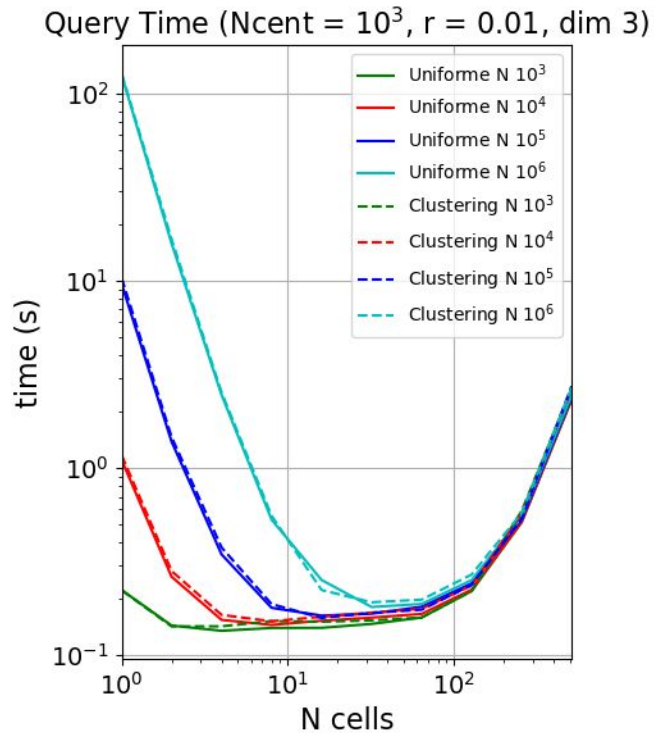
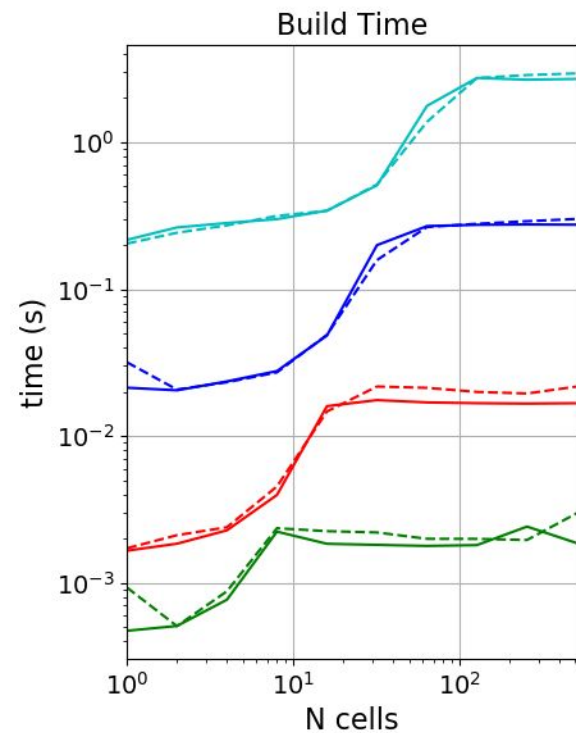
$$\text{Núm. de dimensiones} - K = 3$$

Núm. de celdas -

[1, 2, 4, 8, 16, 32, 64, 128, 264, 512]

# CLUSTERING

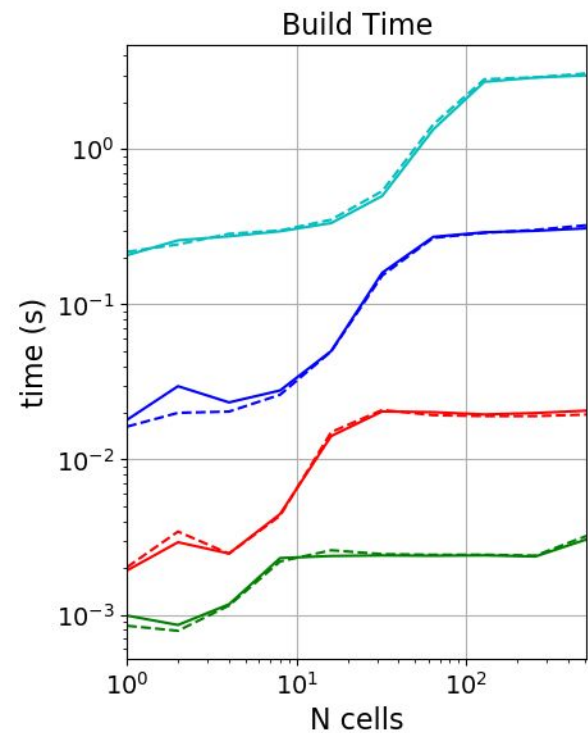
## Centros Random simulación vs Centros Random uniforme



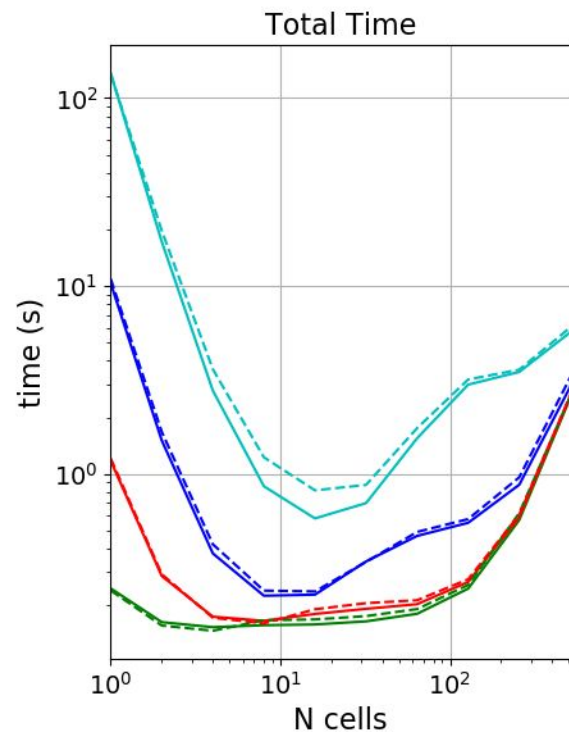
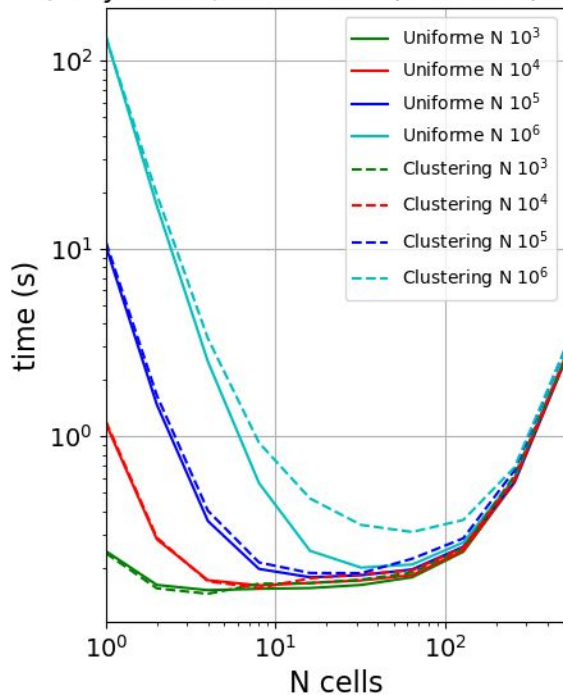


# CLUSTERING

## Centros FoF vs Centros Random en la simulación



Query Time ( $N_{\text{cent}} = 10^3$ ,  $r = 0.01$ , dim 3)



# Resumen

## Dimensiones

- Se comporta de manera adecuada con el número de dimensiones

## Número de centros

- En el rango de prueba el tiempo de Query escala bien con el número de centros

## Número de celdas

- Número óptimo depende de la dimensión
- Aumentar Ncells no mejora la performance
- Depende de la dimensión y el grado de Clustering de los datos

**Publicar!**

# Publicar!

## GriSPy: A Python package for Fixed-Radius Nearest Neighbors Search

Martin Chalela<sup>a,b,c</sup>, Emanuel Sillero<sup>a,b,c</sup>, Luis Pereyra<sup>a,b,c</sup>, Mario Alejandro Garcia<sup>d</sup>, Juan B. Cabral<sup>e,a</sup>, Marcelo Lares<sup>a</sup>, Manuel Merchn<sup>a,b</sup>


<sup>a</sup> Instituto De Astronomía Teórica Y Experimental - Observatorio Astronómico Córdoba (IATE, UNC-CONICET), Córdoba, Argentina.

<sup>b</sup> Observatorio Astronómico de Córdoba, Universidad Nacional de Córdoba, Laprida 854, X5000BGR, Córdoba, Argentina

<sup>c</sup> Facultad de Matemática, Astronomía y Física Universidad Nacional de Córdoba (FaMAF-UNC) Bvd. Medina Allende s/n, Ciudad Universitaria, X5000BGR, Córdoba, Argentina

<sup>d</sup> Universidad Tecnológica Nacional, Facultad Regional Córdoba (UTN-FRC), Maestro M. Lopez esq. Cruz Roja Argentina, Ciudad Universitaria - Córdoba Capital

<sup>e</sup> Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas (CIFASIS, CONICET-UNR), Ocampo y Esmeralda, S2000EZZP, Rosario, Argentina.

 Search projects

Help Sponsor Log in Register

**grispy 0.0.2** Latest version

`pip install grispy` Released: Dec 16, 2019


Grid Search in Python

Navigation

- Project description
- Release history
- Download files

### Project description

#### GriSPy (Grid Search in Python)



`pypi package` `0.0.2` `build` `passing` `docs` `passing` `License` `MIT` `python` `3.6+`


GriSPy is a regular grid search algorithm for quick nearest-neighbor lookup.

This class indexes a set of k-dimensional points in a regular grid providing a fast approach for nearest neighbors queries. Optional periodic boundary conditions can be provided for each axis individually.

GriSPy has the following queries implemented:

Statistics

- GitHub statistics:
  - Stars: 12
  - Forks: 0
  - Open issues/PRs: 1

 **GriSPy**  
latest

Search docs

**CONTENTS:**

- Requirements
- Installation
- Licence
- Tutorial
- API Module

Reach over 7 million devs each month when you advertise with Read the Docs.

Sponsored - Ads served ethically

Docs » GriSPy documentation

[Edit on GitHub](#)

## GriSPy documentation



`pypi package` `0.0.2` `build` `passing` `docs` `passing` `License` `MIT` `Python` `3.6+`

GriSPy (Grid Search in Python) is a regular grid search algorithm for quick nearest-neighbor lookup.

This class indexes a set of k-dimensional points in a regular grid providing a fast approach for nearest neighbors queries. Optional periodic boundary conditions can be provided for each axis individually. Additionally GriSPy provides the possibility of working with individual search radius for each query point in fixed-radius searches and minimum and maximum search radius for shell queries.

### Authors

Martin Chalela (E-mail: [tinchochalela@gmail.com](mailto:tinchochalela@gmail.com)),  
Emanuel Sillero, Luis Pereyra and Alejandro Garcia