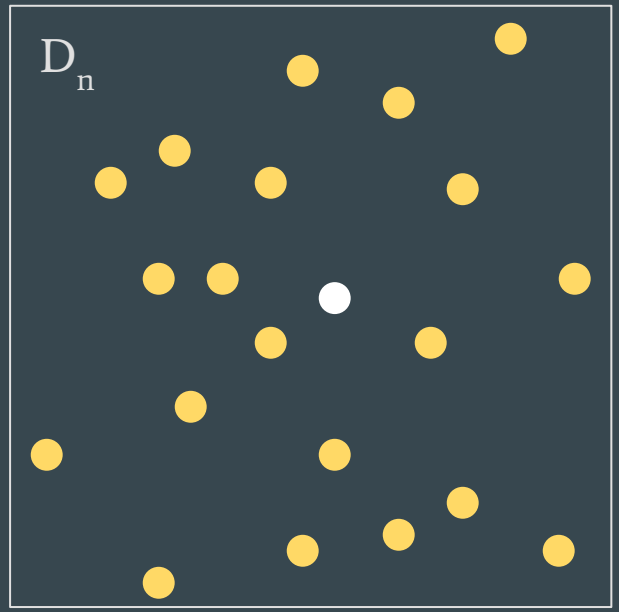
Proyecto: GriSPy

Grid Search in Python

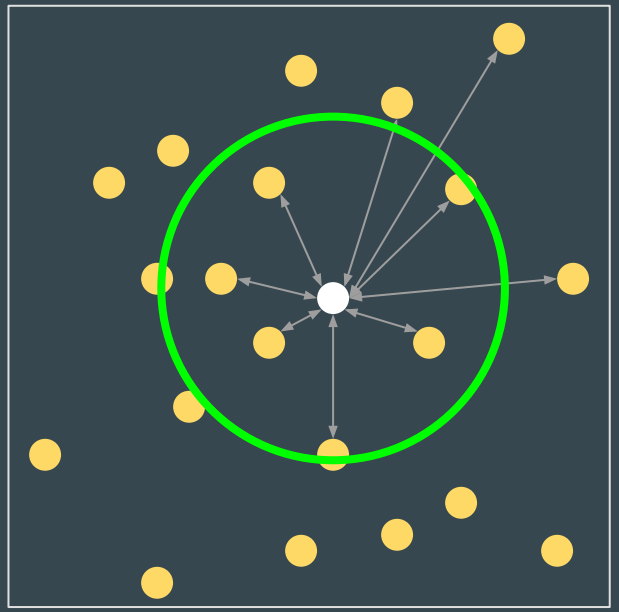
Martin Chalela - Emanuel Sillero - Luis Pereyra - Alejandro Garcia

Problema: Búsqueda de vecinos cercanos

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



k-esimo vecino de c



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

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

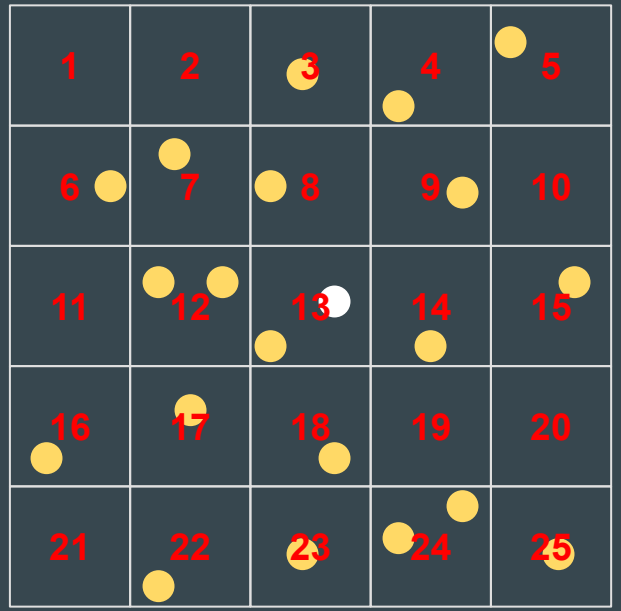
Búsqueda de vecinos más cercano (NNS)

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 ϵ 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

$N \times N = N^d$



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

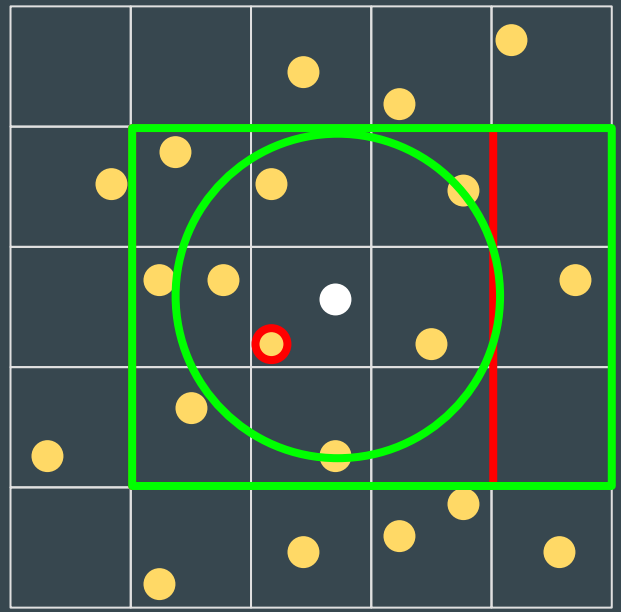
CONSTRUCCIÓN DE LA GRILLA

- Elegir N
- Calcular Δx_d
- Recorrer S indexando x_i

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

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

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



BÚSQUEDA DE VECINOS

- Esfera

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

- Cascara

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

Radios de búsqueda individuales

- k-esimos vecinos

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

Calculo de distancias - Metricas

Euclidea

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

Haversine

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

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

$$B = cos^2\varphi_0\ cos^2\varphi\ sin^2\left(\frac{\Delta\lambda}{2}\right)$$

Vincenty

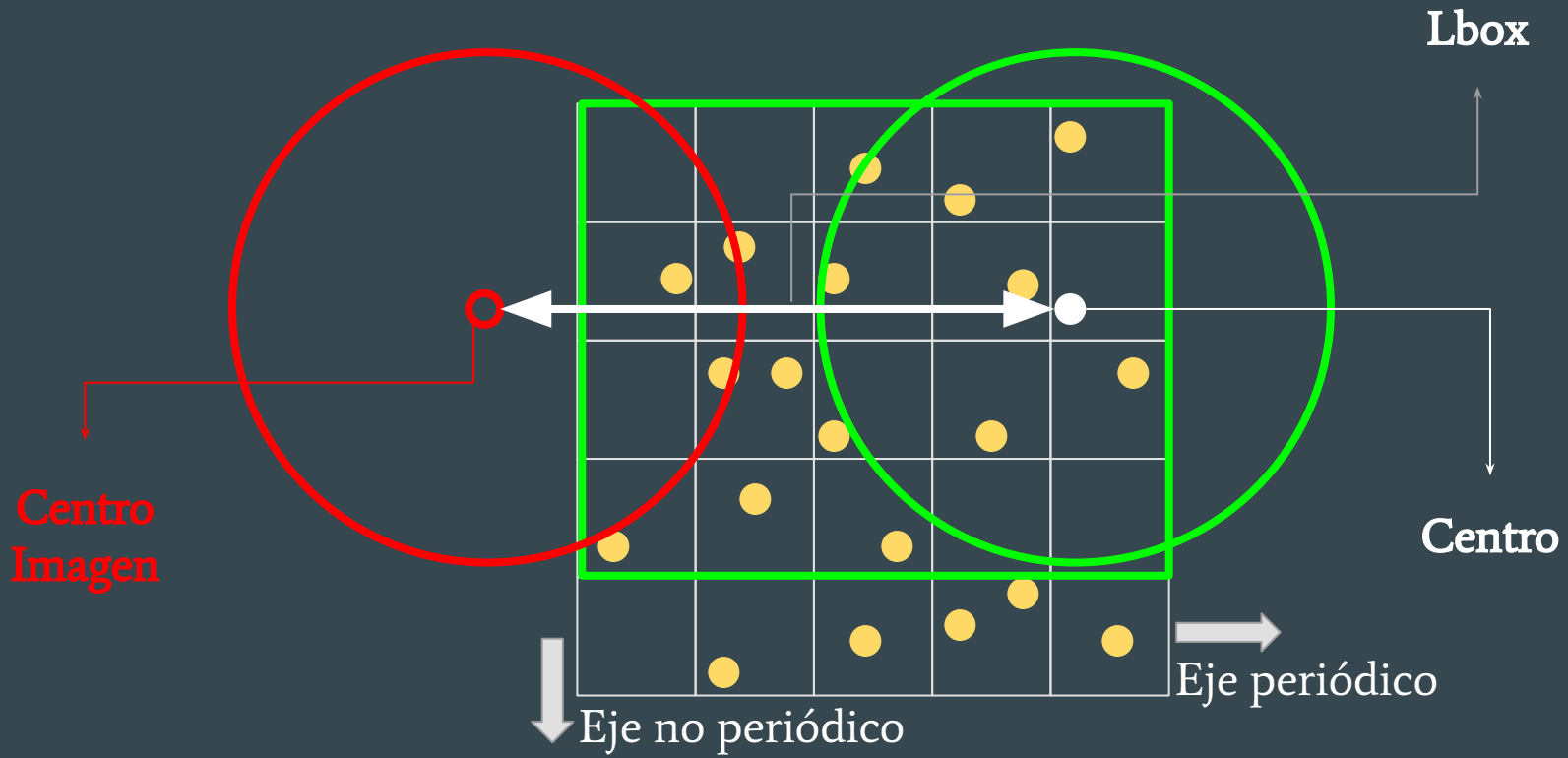
$$D(x_0, x) = arctan\ \frac{\sqrt{E^2 + F^2}}{G}$$

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

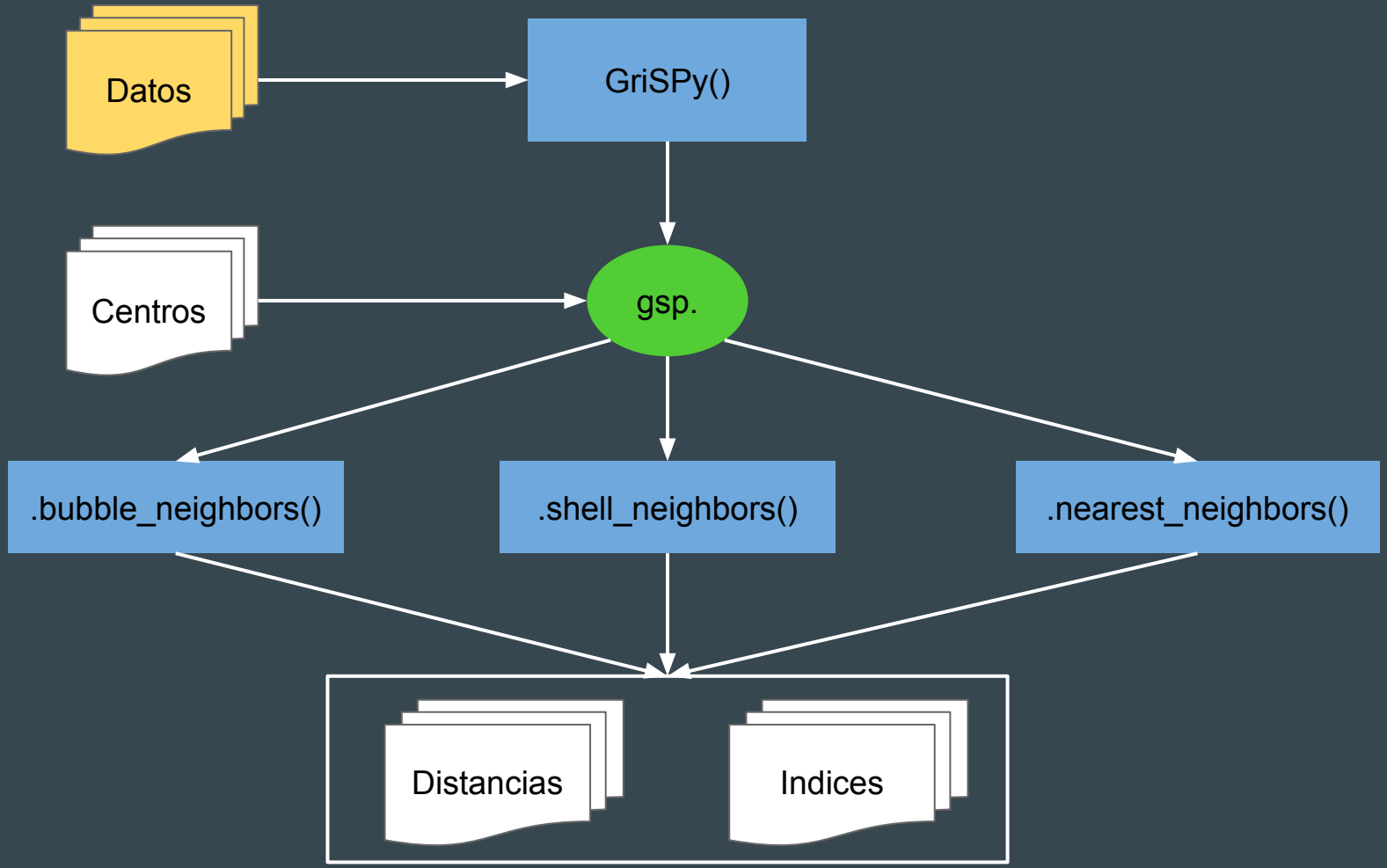
$$F = cos(\varphi_0)\ sin(\varphi) - sin(\varphi_0)\ cos(\varphi)\ (\Delta\lambda)$$

$$G = sin(\varphi_0)\ sin(\varphi) - cos(\varphi_0)\ cos(\varphi)\ (\Delta\lambda)$$

Periodicidad



Desarrollo de GriSPy:
Implementación interna y API


Datos
GriSPy()
Centros
gsp.
.bubble_neighbors()
.shell_neighbors()
.nearest_neighbors()
Distancias
Indices

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
```

```python

    def _build_grid(self, epsilon=1.0e-6):
        """Build the grid."""
        t0 = time.time()
        data_ind = np.arange(len(self.data))
        self.k_bins = np.zeros((self.N_cells + 1, self.dim))
        k_digit = np.zeros(self.data.shape, dtype=int)
        for k in range(self.dim):
            k_data = self.data[:, k]
            self.k_bins[:, k] = np.linspace(
                k_data.min() - epsilon,
                k_data.max() + epsilon,
                self.N_cells + 1,
            )
            k_digit[:, k] = self._digitize(k_data, bins=self.k_bins[:, k])

        compact_ind = np.ravel_multi_index(
            k_digit.T, (self.N_cells,) * self.dim, order="F",
        )

        compact_ind_sort = np.argsort(compact_ind)
        compact_ind = compact_ind[compact_ind_sort]
        k_digit = k_digit[compact_ind_sort]

        split_ind = np.searchsorted(
            compact_ind, np.arange(self.N_cells ** self.dim)
        )
        deleted_cells = np.diff(np.append(-1, split_ind)).astype(bool)
        split_ind = split_ind[deleted_cells]
        if split_ind[-1] > data_ind[-1]:
            split_ind = split_ind[:-1]

        list_ind = np.split(data_ind[compact_ind_sort], split_ind[1:])
        k_digit = k_digit[split_ind]

        self.grid = {}
        for i, j in enumerate(k_digit):
            self.grid[tuple(j)] = list(list_ind[i])

        # Record date and build time
        self.time = {"buildtime": time.time() - t0}
        currentDT = datetime.datetime.now()
        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 iputs
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         )
```

```
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         return neighbors_distances, neighbors_indices
```

1. Celdas vecinas

```python
    # Neighbor-cells methods
    def _get_neighbor_cells(self, centres, distance_upper_bound,
        distance_lower_bound=0, shell_flag=False):

        """Retrieve cells touched by the search radius."""
        cell_point = np.zeros((len(centres), self.dim), dtype=int)
        out_of_field = np.zeros(len(cell_point), dtype=bool)
        for k in range(self.dim):
            cell_point[:, k] = (
                self._digitize(centres[:, k], bins=self.k_bins[:, k])
            )
            out_of_field[
                (centres[:, k] - distance_upper_bound > self.k_bins[-1, k])
            ] = True
            out_of_field[
                (centres[:, k] + distance_upper_bound < self.k_bins[0, k])
            ] = True

        if np.all(out_of_field):
            return [self._empty] * len(centres)  # no neighbor cells

        # Armo la caja con celdas a explorar
        k_cell_min = np.zeros((len(centres), self.dim), dtype=int)
        k_cell_max = np.zeros((len(centres), self.dim), dtype=int)
        for k in range(self.dim):
            k_cell_min[:, k] = (
                self._digitize(
                    centres[:, k] - distance_upper_bound,
                    bins=self.k_bins[:, k],
                )
            )
            k_cell_max[:, k] = (
                self._digitize(
                    centres[:, k] + distance_upper_bound,
                    bins=self.k_bins[:, k],
                )
            )

            k_cell_min[k_cell_min[:, k] < 0, k] = 0
            k_cell_max[k_cell_max[:, k] < 0, k] = 0
            k_cell_min[k_cell_min[:, k] >= self.N_cells, k] = self.N_cells - 1
            k_cell_max[k_cell_max[:, k] >= self.N_cells, k] = self.N_cells - 1
```

```python
        cell_size = self.k_bins[1, :] - self.k_bins[0, :]
        cell_radii = 0.5 * np.sum(cell_size ** 2) ** 0.5

        neighbor_cells = []
        for i in range(len(centres)):
            # Para cada centro i, agrego un arreglo con shape (:,k)
            k_grids = [
                np.arange(k_cell_min[i, k], k_cell_max[i, k] + 1)
                for k in range(self.dim)
            ]
            k_grids = np.meshgrid(*k_grids)
            neighbor_cells += [
                np.array(list(map(np.ndarray.flatten, k_grids))).T
            ]

            # Calculo la distancia de cada centro i a sus celdas vecinas,
            # luego descarto las celdas que no toca el circulo definido por
            # la distancia
            cells_physical = [
                self.k_bins[neighbor_cells[i][:, k], k] + 0.5 * cell_size[k]
                for k in range(self.dim)
            ]
            cells_physical = np.array(cells_physical).T
            mask_cells = (
                self._distance(
                    centres[i], cells_physical
                ) < distance_upper_bound[i] + cell_radii
            )

            if shell_flag:
                mask_cells *= (
                    self._distance(
                        centres[i], cells_physical
                    ) > distance_lower_bound[i] - cell_radii
                )

            if np.any(mask_cells):
                neighbor_cells[i] = neighbor_cells[i][mask_cells]
            else:
                neighbor_cells[i] = self._empty
        return neighbor_cells
```

2. Distancias a los vecinos

```
227
228     def _get_neighbor_distance(self, centres, neighbor_cells):
229         """Retrieve neighbor distances whithin 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             ]
243             # Une en una sola lista todos sus vecinos
244             neighbors_indices += [np.concatenate(ind_tmp).astype(int)]
245             neighbors_distances += [
246                 self._distance(centres[i], self.data[neighbors_indices[i], :])
247             ]
248         return neighbors_distances, neighbors_indices
```

```
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)
```

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_direc * 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( )

```python
    def shell_neighbors(self, centres, distance_lower_bound=-1.0,
        distance_upper_bound=-1.0, sorted=False, kind="quicksort"):
        """Find all points within given lower and upper distances of each centre."""

        # Validate inputs
        utils.validate_centres(centres, self.data)
        utils.validate_bool(sorted)
        utils.validate_sortkind(kind)
        utils.validate_shell_distances(
            distance_lower_bound, distance_upper_bound, self.periodic,
        )

        # Match distance bounds shapes with centres shape
        if np.isscalar(distance_lower_bound):
            distance_lower_bound *= np.ones(len(centres))
        else:
            utils.validate_equalsize(centres, distance_lower_bound)
        if np.isscalar(distance_upper_bound):
            distance_upper_bound *= np.ones(len(centres))
        else:
            utils.validate_equalsize(centres, distance_upper_bound)

        # Get neighbors
        neighbor_cells = self._get_neighbor_cells(
            centres,
            distance_upper_bound=distance_upper_bound,
            distance_lower_bound=distance_lower_bound,
            shell_flag=True,
        )
        neighbors_distances, neighbors_indices = self._get_neighbor_distance(
            centres, neighbor_cells)
```

```python
        # We need to generate mirror centres for periodic boundaries...
        if self.periodic_flag:
            terran_centres, terran_indices = self._mirror_universe(
                centres, distance_upper_bound)
            # terran_centres are the centres in the mirror universe for those
            # near the boundary.
            terran_neighbor_cells = self._get_neighbor_cells(
                terran_centres, distance_upper_bound[terran_indices])

            terran_neighbors_distances,\
                terran_neighbors_indices = self._get_neighbor_distance(
                    terran_centres, terran_neighbor_cells)

            for i, t in zip(terran_indices, np.arange(len(terran_centres))):
                # i runs over normal indices that have a terran counterpart
                # t runs over terran indices, 0 to len(terran_centres)
                neighbors_distances[i] = np.concatenate(
                    (neighbors_distances[i], terran_neighbors_distances[t]) )
                neighbors_indices[i] = np.concatenate(
                    (neighbors_indices[i], terran_neighbors_indices[t]) )

        for i in range(len(centres)):
            mask_distances_upper = neighbors_distances[i] <= distance_upper_bound[i]
            mask_distances_lower = neighbors_distances[i][mask_distances_upper]
            mask_distances_lower = mask_distances_lower > distance_lower_bound[i]

            aux = neighbors_distances[i]
            aux = aux[mask_distances_upper]
            aux = aux[mask_distances_lower]
            neighbors_distances[i] = aux

            aux = neighbors_indices[i]
            aux = aux[mask_distances_upper]
            aux = aux[mask_distances_lower]
            neighbors_indices[i] = aux

            if sorted:
                sorted_ind = np.argsort(neighbors_distances[i], kind=kind)
                neighbors_distances[i] = neighbors_distances[i][sorted_ind]
                neighbors_indices[i] = neighbors_indices[i][sorted_ind]

        return neighbors_distances, neighbors_indices
```

nearest_neighbors( )

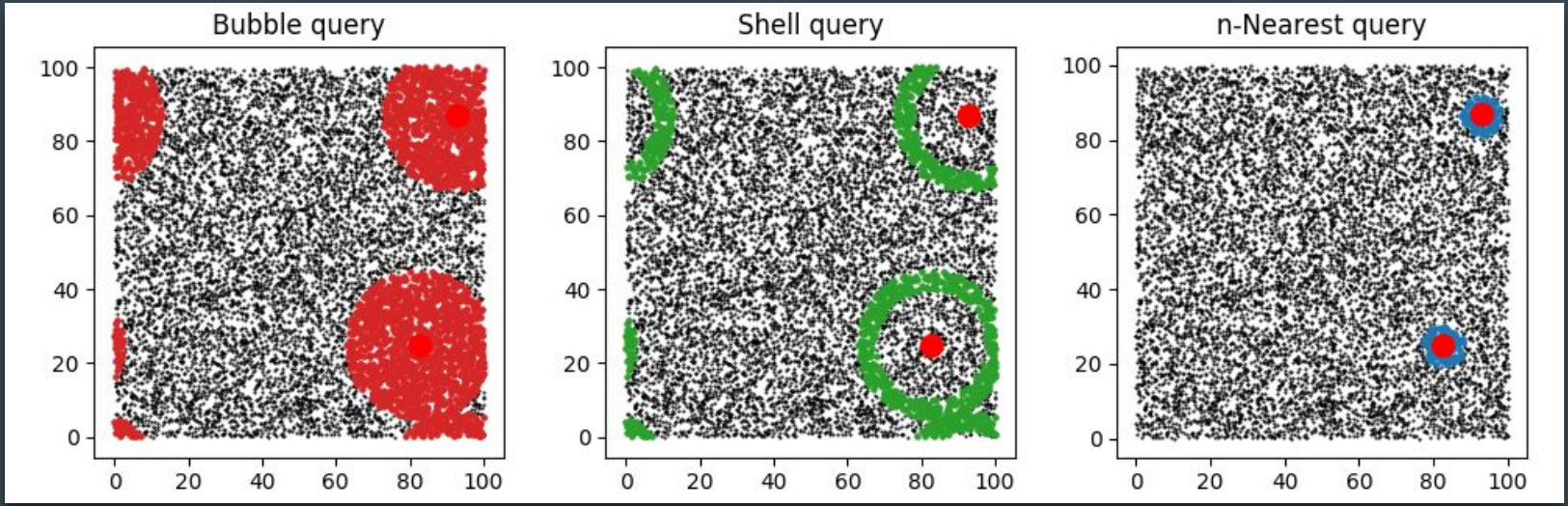
```python
    def nearest_neighbors(self, centres, n=1, kind="quicksort"):
        """Find the n nearest-neighbors for each centre."""

        # Validate input
        utils.validate_centres(centres, self.data)
        utils.validate_n_nearest(n, self.data, self.periodic)
        utils.validate_sortkind(kind)

        # Initial definitions
        N_centres = len(centres)
        centres_lookup_ind = np.arange(0, N_centres)
        n_found = np.zeros(N_centres, dtype=bool)
        lower_distance_tmp = np.zeros(N_centres)
        upper_distance_tmp = np.zeros(N_centres)

        # Abro la celda del centro como primer paso
        centre_cell = self._get_neighbor_cells(
            centres, distance_upper_bound=upper_distance_tmp
        )
        # crear funcion que regrese vecinos sin calcular distancias
        neighbors_distances, neighbors_indices = self._get_neighbor_distance(
            centres, centre_cell
        )

        # Calculo una primera aproximacion con la
        # 'distancia media' = 0.5 * (n/denstiy)**(1/dim)
        # Factor de escala para la distancia inicial
        mean_distance_factor = 1.0
        cell_size = self.k_bins[1, :] - self.k_bins[0, :]
        cell_volume = np.prod(cell_size.astype(float))
        neighbors_number = np.array(list(map(len, neighbors_indices)))
        mask_zero_neighbors = neighbors_number == 0
        neighbors_number[mask_zero_neighbors] = 1
        mean_distance = 0.5 * (n / (neighbors_number / cell_volume)) ** (
            1.0 / self.dim)

        upper_distance_tmp = mean_distance_factor * mean_distance
```

```python
        neighbors_indices = [self._empty] * N_centres
        neighbors_distances = [self._empty] * N_centres
        while not np.all(n_found):
            neighbors_distances_tmp,\
                neighbors_indices_tmp = self.shell_neighbors(
                    centres[~n_found],
                    distance_lower_bound=lower_distance_tmp[~n_found],
                    distance_upper_bound=upper_distance_tmp[~n_found],
                )

            for i_tmp, i in enumerate(centres_lookup_ind[~n_found]):
                if n_found[i]:
                    continue
                if n <= len(neighbors_indices_tmp[i_tmp]) + len(
                    neighbors_indices[i]
                ):
                    n_more = n - len(neighbors_indices[i])
                    n_found[i] = True
                else:
                    n_more = len(neighbors_indices_tmp[i_tmp])
                    lower_distance_tmp[i_tmp] = upper_distance_tmp[
                        i_tmp
                    ].copy()
                    upper_distance_tmp[i_tmp] += cell_size.min()

                sorted_ind = np.argsort(
                    neighbors_distances_tmp[i_tmp], kind=kind
                )[:n_more]
                neighbors_distances[i] = np.hstack(
                    (
                        neighbors_distances[i],
                        neighbors_distances_tmp[i_tmp][sorted_ind],
                    )
                )
                neighbors_indices[i] = np.hstack(
                    (
                        neighbors_indices[i],
                        neighbors_indices_tmp[i_tmp][sorted_ind],
                    )
                )

        return neighbors_distances, neighbors_indices
```

Save & Load

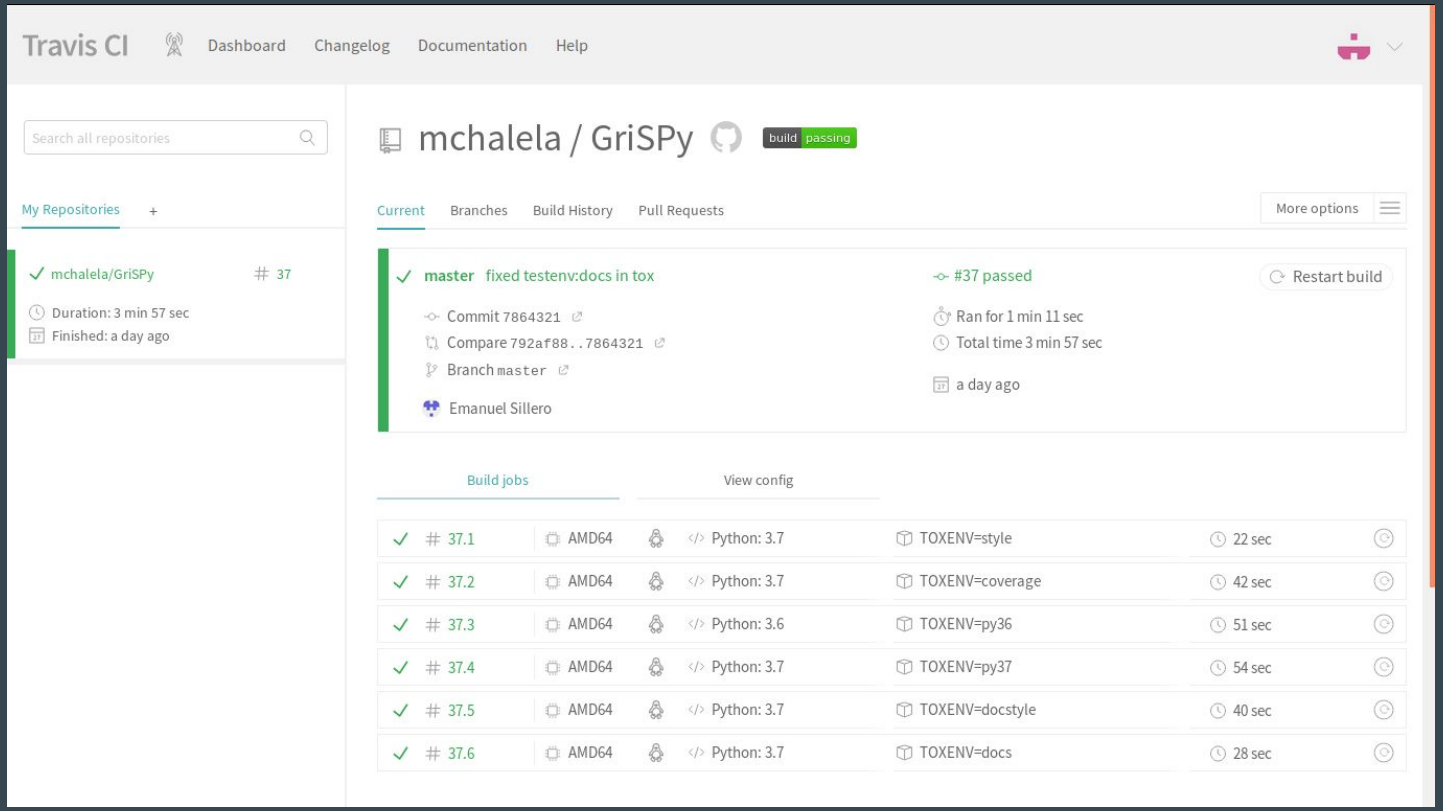
```
652     def save_grid(self, file="grispy.gsp", overwrite=False):
653         """Save all grid attributes in a binary file for future use."""
654 
655         # Validate input
656         utils.validate_filename(file)
657         utils.validate_bool(overwrite)
658         utils.validate_canwrite(file, 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             "Succsefully loaded GriSPy grid created on {}".format(
686                 gsp.time["datetime"])
687         )
688         return gsp
```

Ejemplo



Calidad:
Testing e integración continua

Integración continua con Travis CI



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`).

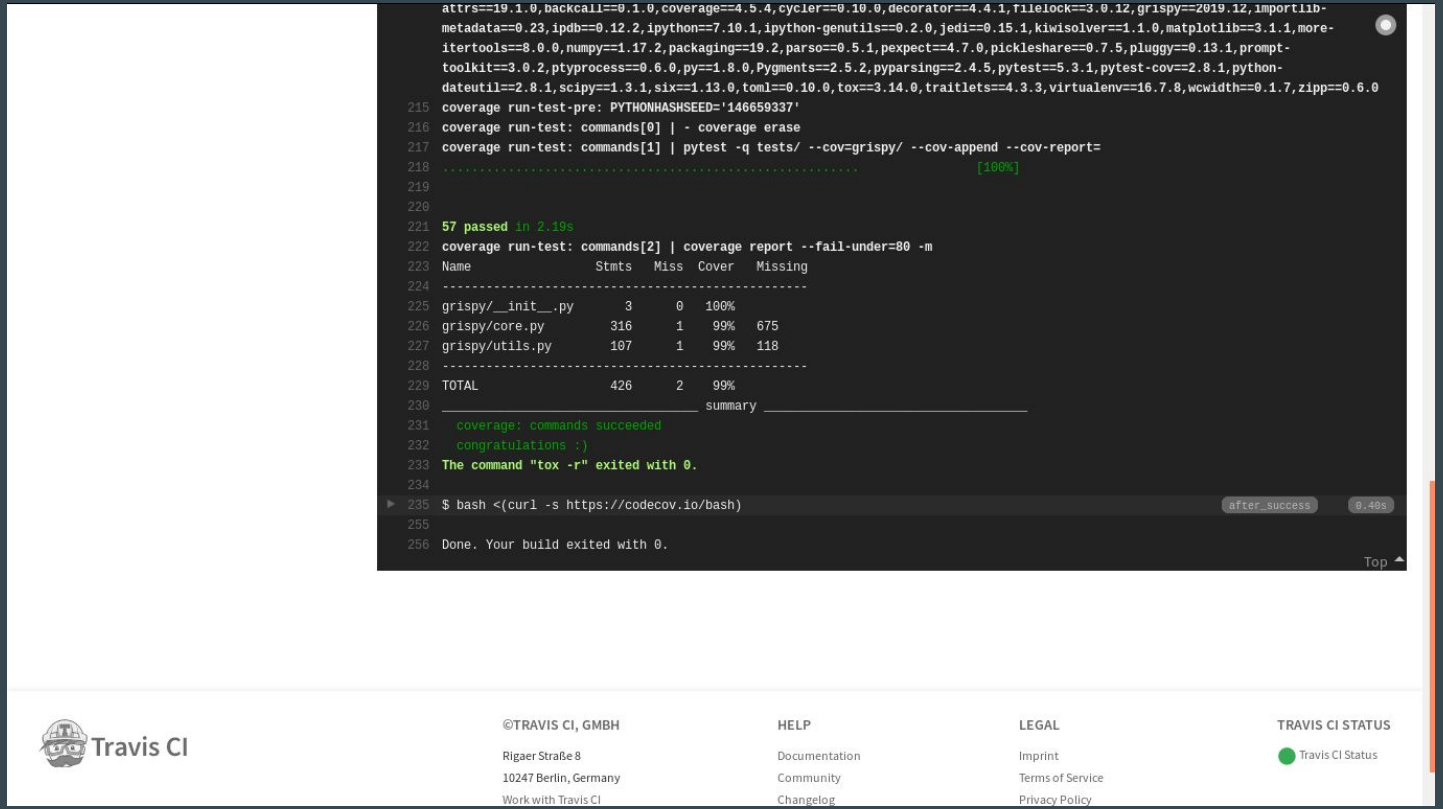
- Condición I (3)
  - dist(a, b) != NaN
- Condición II (3)
  - $0 \leq$ dist(a, b)
- Condición III (3)
  - dist(a, b) = dist(b, a)
- Condición IV (3)
  - dist(a, c) $\leq$ dist(a, b) + dist(b, c)

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,cycler==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,ptyprocess==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/utils.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) after_success 0.40s
255
256 Done. Your build exited with 0.
Top
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

Benchmarking

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 — **fracción del Box**
- Métrica — **Euclidiana**
- Condiciones Periódicas — **No periódicas**
- El grado de clustering de los datos — **Uniforme**

Variando K-dim [2 - 3 - 4]

Lbox = [0.0, 1.0]

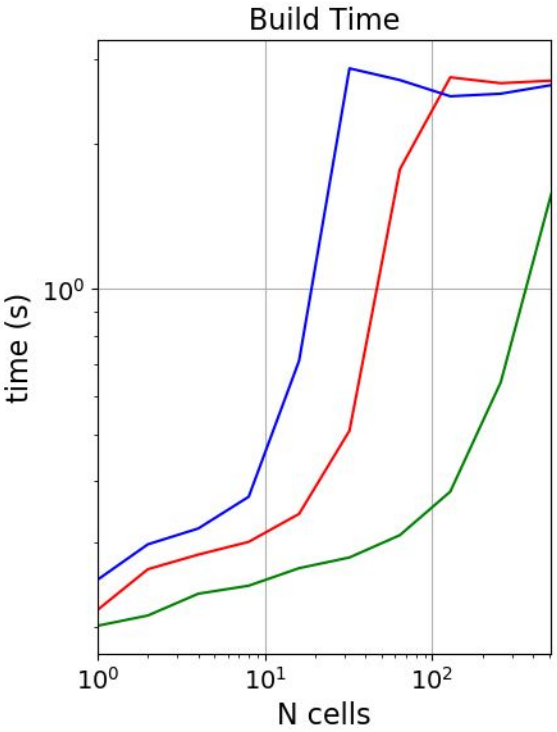
Núm. de centros - Ncent = 10^3

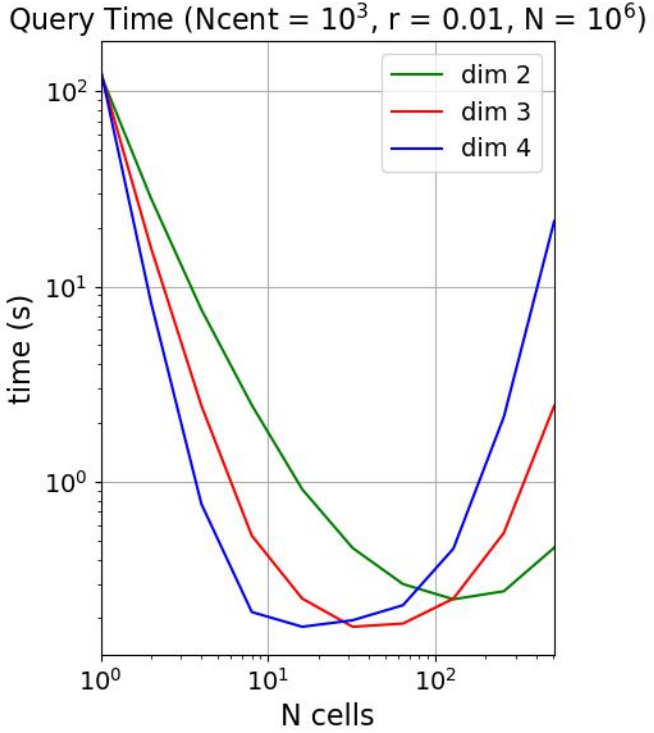
Núm. de puntos - N = 10^6

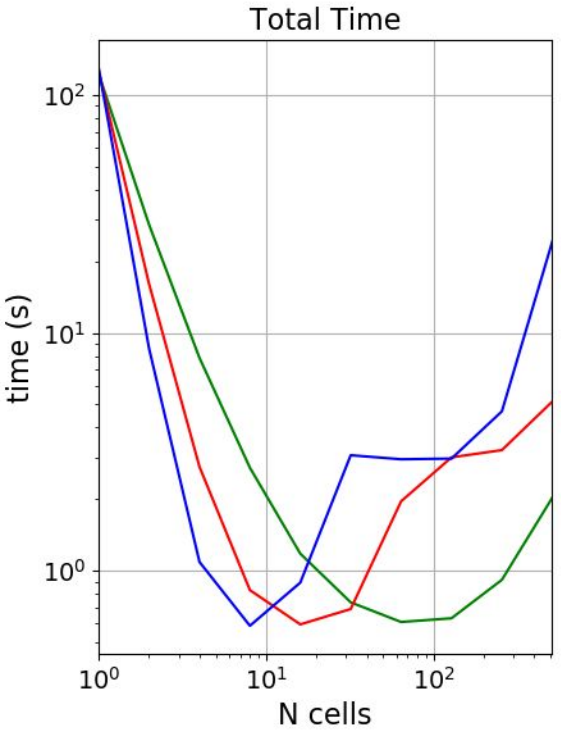
Núm. de celdas -

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

UNIFORME


Build Time
time (s)
10^0
10^0
10^1
10^2
N cells


Query Time (Ncent = 10^3, r = 0.01, N = 10^6)
dim 2
dim 3
dim 4
time (s)
10^2
10^1
10^0
10^0
10^1
10^2
N cells


Total Time
time (s)
10^2
10^1
10^0
10^0
10^1
10^2
N cells

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

Lbox = [0.0, 1.0]

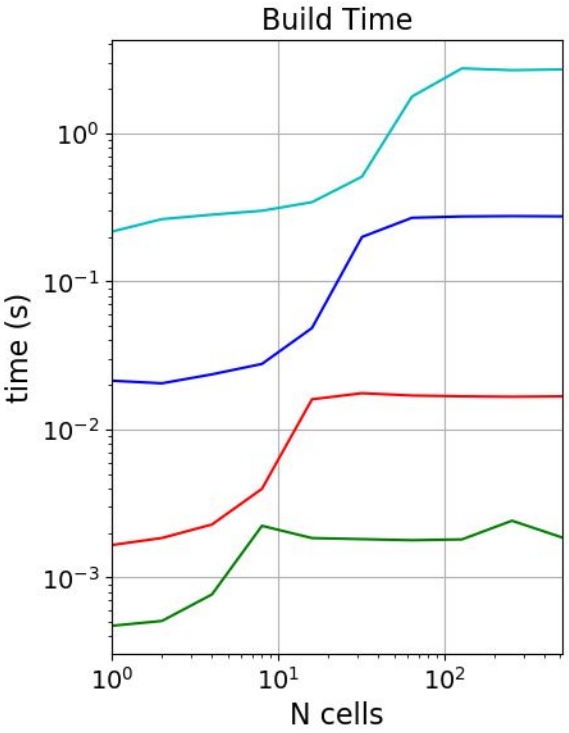
Núm. de centros - Ncent = 10^3

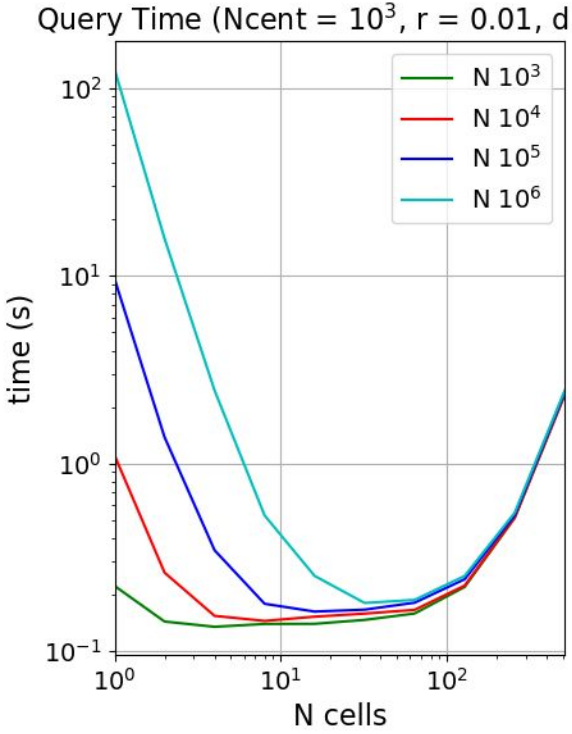
Núm. de dimensiones - K = 3

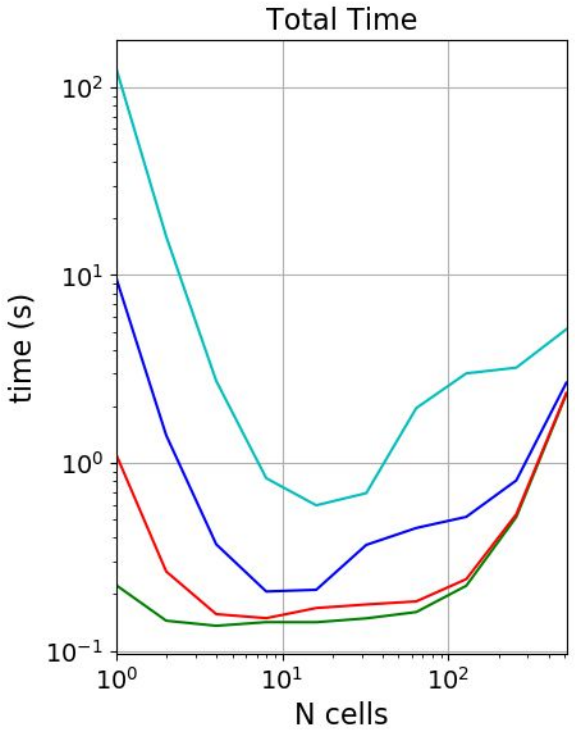
Núm. de celdas -

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

UNIFORME


Build Time
time (s)
N cells


Query Time (Ncent = 10^3, r = 0.01, dim 3)
N 10^3
N 10^4
N 10^5
N 10^6
time (s)
N cells


Total Time
time (s)
N cells

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

Lbox = [0.0, 1.0]

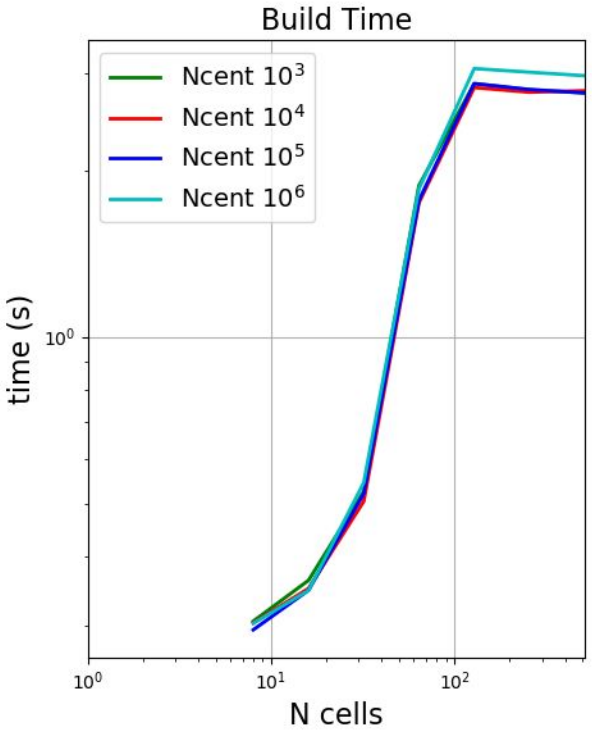
Núm. de centros - $N = 10^6$

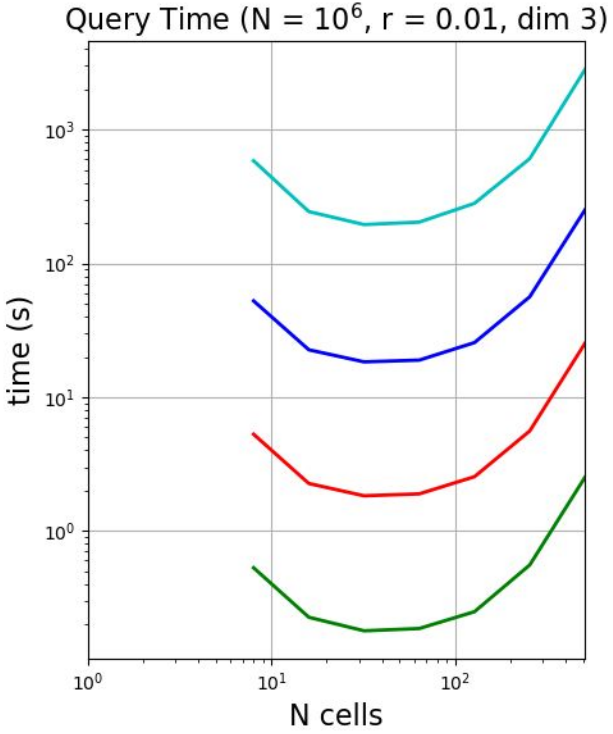
Núm. de dimensiones - K = 3

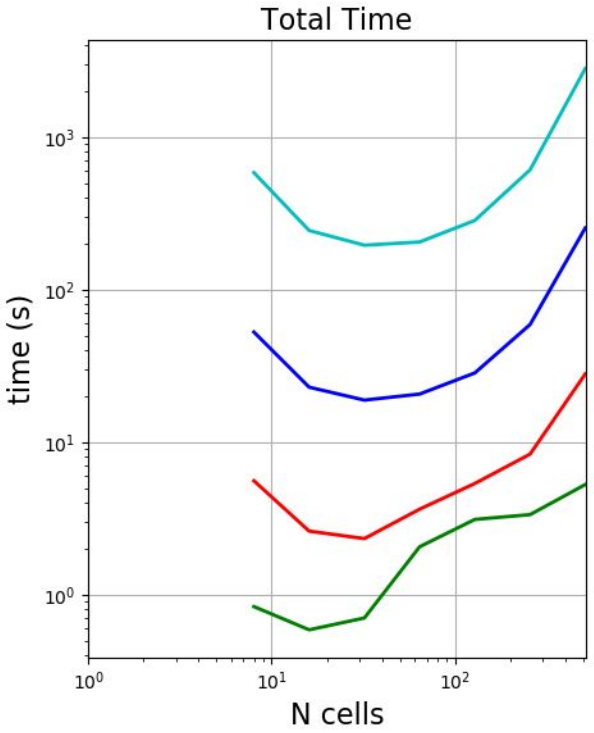
Núm. de celdas -

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

UNIFORME


Build Time
Ncent 10^3
Ncent 10^4
Ncent 10^5
Ncent 10^6
time (s)
10^0
10^0
10^1
10^2
N cells


Query Time (N = 10^6, r = 0.01, dim 3)
time (s)
10^3
10^2
10^1
10^0
10^0
10^1
10^2
N cells


Total Time
time (s)
10^3
10^2
10^1
10^0
10^0
10^1
10^2
N cells

Pruebas con Clustering

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

Lbox = [0.0, 1.0]

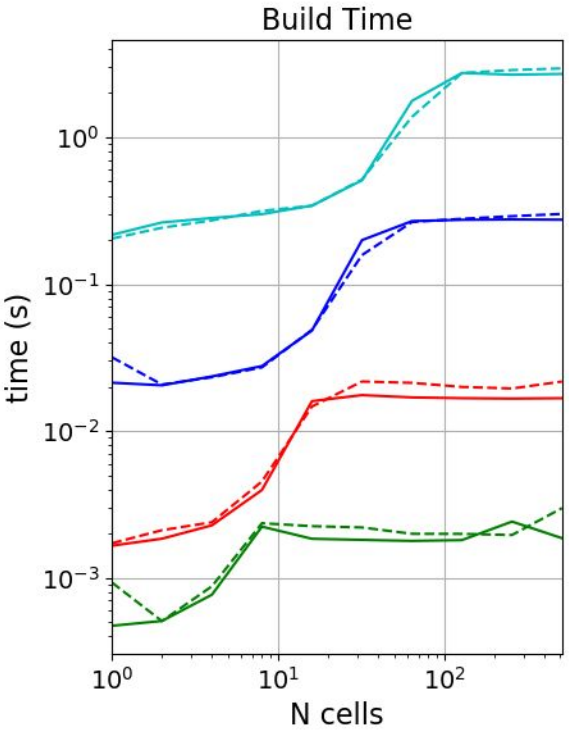
Núm. de centros - Ncent = 10^3

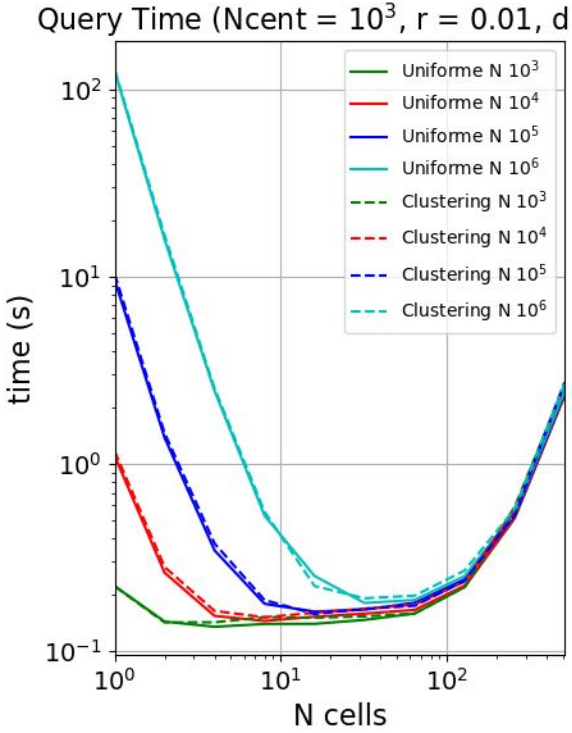
Núm. de dimensiones - K = 3

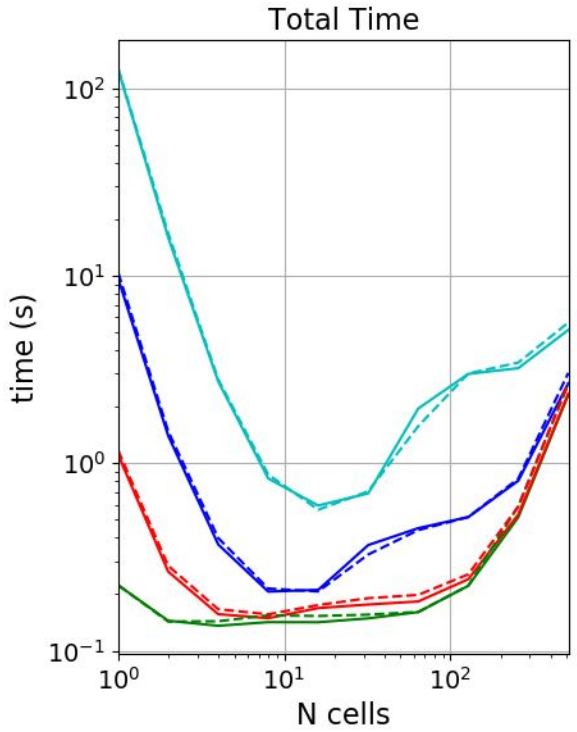
Núm. de celdas -

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

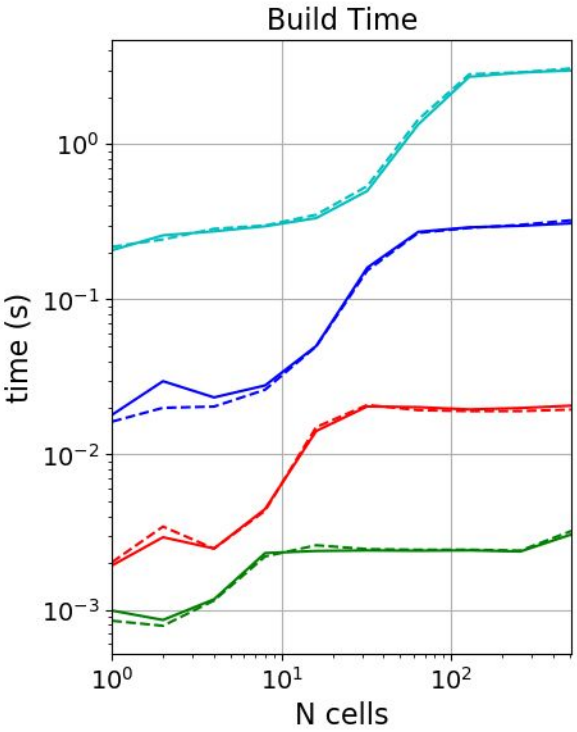
CLUSTERING
Centros Random simulación vs Centros Random uniforme

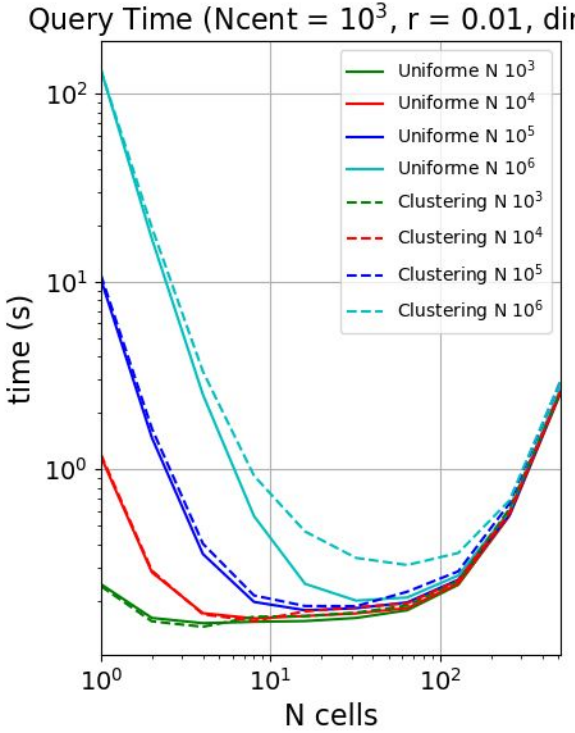


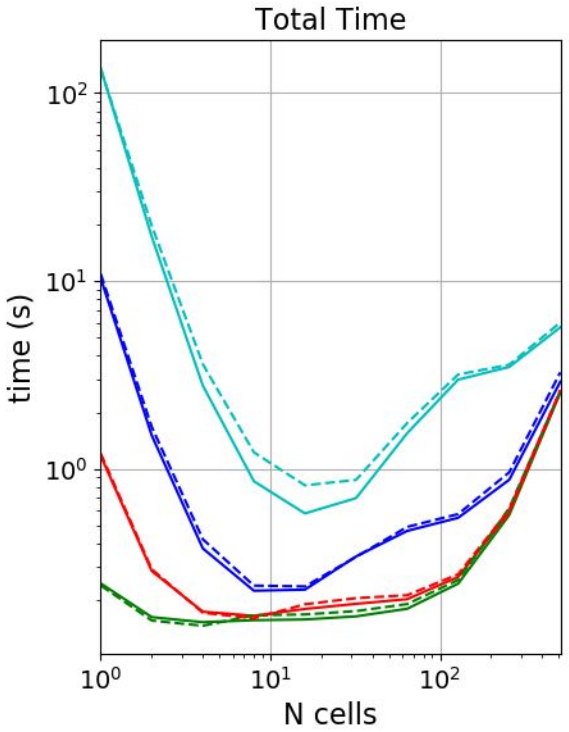




CLUSTERING
Centros FoF vs Centros Random en la simulación







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