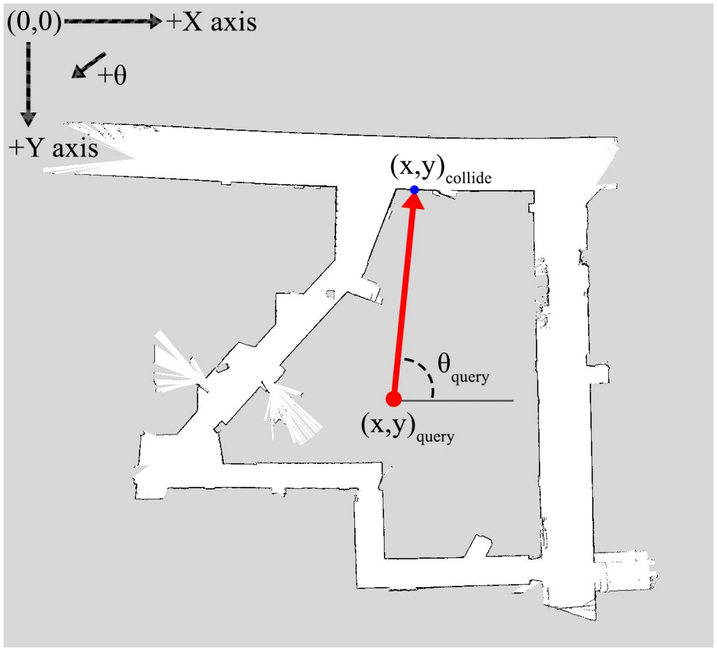
RangeLibc Usage and Information

RangeLibc is a C++/CUDA/Python library containing optimized implementations of several 2D ray casting methods, written by Corey Walsh, and inspired by Spring 2016 6.141. It is designed to accelerate the computation of particle filter sensor models in two dimensional occupancy grid maps, such as the below Stata basement map.



Problem Definition

We define the problem of ray casting in occupancy grids as follows. We assume a known occupancy grid map in which occupied cells have value 1, and unoccupied cells have value 0. Given a query pose $(x, y, \theta)_{query}$ in map space, the raycast operation finds the nearest occupied pixel $(x, y)_{collide}$ which lies on the ray starting at position $(x, y)_{query}$ and pointing in the θ_{query} direction, and returns the euclidean distance between the query and collision point.

Coordinate Space

For efficiency and simplicity, RangeLibc operates on the coordinate space of the occupancy grid map, where one pixel is one unit. Additionally, RangeLibc uses a left handed coordinate space. Since the conversion from occupancy grid coordinates to logical map coordinates can be confusing, we perform the necessary conversion for you behind the scenes inside RangeLib. To take advantage of this, you need to initialize the PyOmap class by directly passing an OccupancyGrid message from a running map_server node. If you use an alternate set of constructor arguments, then RangeLibc will not have the necessary information to do the conversion for you.

Algorithm Overview

Understanding the precise definition of the RangeLibc algorithms is not critical to usage. This section is included simply for reference or interest.

Bresenham's Line (BL)

One of the simplest and most widely used algorithms for 2D ray casting. Also the slowest option. It steps along the pixels approximating a query ray until an obstacle is encountered. More info.

Ray Marching (RM)

Similar to BL in the sense that it steps along the query ray until an obstacle is encountered. The key difference is that it makes larger steps along the ray by consulting a precomputed euclidean distance transform. Well known in computer graphics, less frequently used in robotics. In the worst case it is approximately equivalent to BL and thus has a long tail performance distribution.

Implemented for both the CPU and GPU. GPU version (RMGPU) is insanely fast, but works best with very large query batch sizes.

Compressed Directional Distance Transform (CDDT)

CDDT is a novel algorithm recently designed and implemented by Corey Walsh. It exploits the natural compressibility of the 3D lookup table approach (see below) while still allowing for 0(1) query time in a fixed size map. Provides good query performance and initialization time. Can be slightly accelerated with a “Pruning” operation which discards unnecessary data from the acceleration data structure - pruned version called PCDDT.

This **pre-publication** paper describes CDDT if you are interested, please **do not share it.**

3D Lookup Table (LUT)

Rather than performing ray casting at run time, one can precompute the ray cast solution over a discretized state space. At runtime, simply round the query to the nearest discrete state and read from the table. Very fast (constant time) queries. Very slow construction and high memory usage. CDDT is analogous, but uses ~2 orders of magnitude less memory and precomputation, with similar query times, which might be preferable for development purposes.

See section 3.4.1 of this paper by Fox et al. for more information.

Performance Comparison

The below speed comparisons are for queries on the basement map, running onboard the Jetson TX1. Benchmarks performed in C++ (without coordinate space conversion) so the Python wrapped version will be slightly slower; using batched queries is critical for good performance in Python.

Method	**Init time (sec)**	**Random Queries/Sec**	**Grid Queries/Sec**	**Memory (MB)**
BL	**0.02**	297,000	360,000	**1.37**
RM	0.59	1,170,000	1,600,000	5.49
RMGPU	0.68	**18,000,000**	**28,000,000**	5.49
CDDT	*0.25*	2,090,000	4,250,000	6.34
PCDDT	7.96	*2,260,000*	4,470,000	*4.07*
LUT	64.5	2,160,000	*4,850,000*	296.6

Library Overview

The core range methods are written in efficient C++. `RangeLib.h` contains the majority of the core code. All methods inherit from a base class called `RangeMethod` which defines the common interface and numpy wrapper interface. CUDA code is in `kernels.cu`. Python wrappers reside in the `pywrapper` directory. Every range method implements a `calc_range` function which given a query pose returns distance in the pixel coordinate space described above. Feel free to modify the library as you see fit to accelerate your particle filter - offloading computation to C++ will certainly help.

The Python wrappers expose `calc_range` directly, though we do not recommend that you use those functions as the function call overhead is large compared to computation time. Instead, you should use the `calc_range_many` or `calc_range_repeat_angles` function with large batch sizes (thousands of ray casts) per call. See below for more information.

Installation and Compilation

```
git clone https://github.com/kctess5/range_libc
cd range_libc
# to compile C++ code only with or without CUDA methods
# not necessary if you only want the Python wrappers
mkdir build && cd build
cmake .. -DWITH_CUDA=ON
cmake .. -DWITH_CUDA=OFF
make
# to build the Python wrapped version of the library
cd ../pywrapper
pip install Cython
sudo python setup.py install
# OR, to compile Python wrappers with CUDA methods
sudo WITH_CUDA=ON python setup.py install
```

Python Usage

See `pywrapper/RangeLIbc.pyx` for the definition of these wrapper functions.

```
class PyOmap(OccupancyGrid map_msg):
    bool save(string file_path)
    bool error()
```

Note: there are other `PyOmap` initialization argument options, which can be given raw boolean numpy array occupancy grids. However, we recommend you don't try to use them since that would require you to figure out some rather difficult coordinate space conversions.

Shared RangeMethod interface:

A core enabling principle behind the batched query implementation of RangeLibc is the fact that Numpy arrays are effectively pointer container types. Therefore, passing numpy arrays of arbitrary size requires only small constant time cost since no copy operation is done.

`nd[float32,Nx3]` implies a 2D size (N,3) c-contiguous `numpy.ndarray` of type `np.Float32`

`nd[float32,N*M]` implies a 1D size (N*M) c-contiguous numpy.ndarray of type `np.Float32`

```
class Py[all_methods](PyOMap map, float max_range_px, [int theta_discretization]):
    # calc_range is implemented but you should not use it, it is slow and
    # necessary coordinate space conversion from ROS to RangeLibc isn't implemented
    float calc_range(float x,y,theta)
    # use these batched calc_range calls for better efficiency
    void  calc_range_many(nd[float32,Nx3] ins, nd[float32,N] outs)
    void  calc_range_repeat_angles(nd[float32,Nx3] queries,nd[float32,M] angles
                                   nd[float32, N*M] outs)
    void  set_sensor_model(nd[float64,KxK] sensor_table)
    void  eval_sensor_model(nd[float32,M] obs, nd[float32,N*M] ranges,
                            nd[float64,N] weights,
                            int num_rays[=M], int num_particles[=N])
```

Other functions:

```
# if you compile with TRACE=ON then bl and rm methods will keep track of all the places
where the map is accessed while running calc_range functions. This can be useful for
debugging but is a bit slow, so it's currently disabled.
class PyBresenhamsLine(PyOMap map, float max_range):
    float saveTrace(string filename)

class PyRayMarching(PyOMap map, float max_range):
    float saveTrace(string filename)
```

```
class PyRayMarchingGPU(PyOMap map, float max_range):
    # DOES NOT PROVIDE calc_range function - batched queries required
    # can use any batched (void) function as expected
    # for high performance, use large batch sizes (i.e. one call per MCL update)
```

`float calc_range(float x,y,theta)`

Ray cast from position (x,y) in the theta direction. This function is provided for convenience, but for optimal performance you should avoid it since the function call overhead is large compared to the range query time. If you use this you will have a bad time since the coordinate space conversions from ROS to RangeLibc coordinates are only implemented for batched queries.

`void calc_range_many(nd[float32,N] queries, nd[float32,N] outs)`

Takes an Nx3 numpy array of (x,y,theta) queries and populates an output array of size N with ranges. Equivalent to the following, with significantly lower function call overhead. Handles coordinate space conversion from logical ROS coordinates to RangeLibc.

```
for i in xrange(queries.shape[0]):
    outs[i] = calc_range(queries[i,0],queries[i,1],queries[i,2])
```

`void calc_range_repeat_angles(nd[float32,Nx3] queries,`
`                               nd[float32,M] angles,`
`                               nd[float32, N*M] outs)`

Since it is likely that you would like to evaluate the calc_range function with many different angles for each given query position (x,y) while evaluating the sensor model, we also provide a `calc_range_repeat_angles` which takes query positions of size (N,3) and angles of size (M) and populates the ranges array of size (NM) by adding each angle to each query position. Handles coordinate space conversion from logical ROS coordinates to RangeLibc. Equivalently:

```
for i in xrange(queries.shape[0]):
    rays_per_particle = angles.shape[0]
    for a in xrange(rays_per_particle):
        outs[i*rays_per_particle+a] = \
           calc_range(queries[i,0],queries[i,1],queries[i,2]+angles[a])
```

`void set_sensor_model(nd[float64,KxK] sensor_table)`

If you are precomputing your sensor model into a table (which you should), then you can upload it to the range methods with the `set_sensor_model` function. It is important that your sensor model is the correct shape: it should be size `K=int(max_range_px)+1` in two dimensions. We set this up for you in the skeleton code.

```
void eval_sensor_model(nd[float32,M] obs, nd[float32,N*M] ranges,
                       nd[float64,N] weights,
                       int num_rays[=M], int num_particles[=N])
```

Once you have called `calc_range_repeat_angles`, you can look up partial weights from your precomputed sensor table and multiplicatively accumulate them in one function call with `eval_sensor_model`. It is equivalent to the following:

```
# num_rays=M is the number of ray casts performed per particle
# num_rays is in the range [1:1081] depending on your downsampling factor
# num_particles=N
for i in xrange(num_particles):
    weight = 1.0
    for a in xrange(num_rays):
        weight *= sensor_table[obs[a],ranges[i*num_rays+a]]
    weights[i] = weight
```

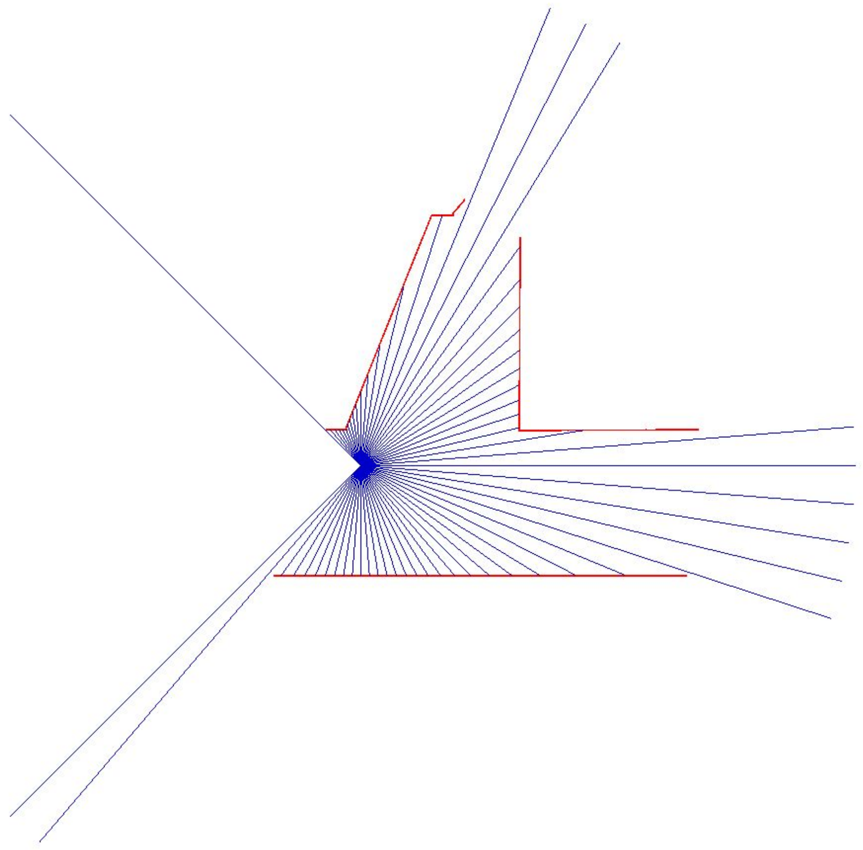
In practice evaluation of your sensor model might look something like this:

```
ranges = np.zeros(num_rays*num_particles)
weights = np.zeros(num_particles)
calc_range_repeated_angles(particles, ray_angles, ranges)
eval_sensor_model(observations, ranges, weights, num_rays, num_particles)
```

Note: the ranges array is one dimensional of size N*M, not two dimensional NxM. Notice the indexing arithmetic bolded in the above pseudocode. All ranges for a single particle should be in a contiguous section of memory. This array is one dimensional simply because the outputs of the other batched range methods are also one dimensional, using the same ordering.

You will definitely find that computing your sensor model in this way is orders of magnitude more efficient than using nested for loops over your particle distribution - it is the (not so) secret sauce which allows the TA Python MCL implementation to maintain >10000 particles in real time.

```
void saveTrace(string filename)
```



This function is included to help debugging. If you compile RangeLibc with TRACE=ON it will track map accesses in both the Bresenham's Line and Ray Marching methods. This can be very useful for finding issues with coordinate space conversions. The above figure was generated with this code:

```
# n_ranges, x, y, theta = int, float, float, float
bl = range_libc.PyBresenhamsLine(testMap, 500)
queries = np.zeros((n_ranges,3),dtype=np.float32)
ranges = np.zeros(n_ranges,dtype=np.float32)
queries[:,0] = x
queries[:,1] = y
queries[:,2] = theta + np.linspace(-MAX_SCAN_ANGLE, MAX_SCAN_ANGLE, n_ranges)
bl.calc_range_many(queries,ranges)
bl.saveTrace("./test.png")
```

To compile:

```
sudo TRACE=ON python setup.py install
```