Assignment 1: Utilising Abstraction for a Range of Sensor Classes

22 Aug 2020 – Initial Release
04 Sep 2021 – Updates (in red) and added FAQ
07 Sep 2021 – Updates (in red), reweighted sonar coverage as BONUS mark, added to FAQ

Intent: Skills in utilising, classes, functions, pointers and utilising abstraction, encapsulation, inheritance, polymorphism with appropriate documentation will be assessed.

Individual Task

Weight: 20%

Task: Write a program in C++ using object oriented paradigms that embodies a range of Mechatronics sensors, utilising abstraction, encapsulation, inheritance and polymorphism. Supply appropriate auto-generated documentation utilising inline source mark-up.

Rationale: In a Mechatronics System we would use a class to represent a real-world object such as a Sensor. In large projects to facilitate testing it is common to develop a mock (fake) sensor class that behaves the same as the real sensor. This allows for some system testing without the presence of all hardware.

Sensors producing data of similar nature are often abstracted from a base sensor (ie. sensor that produce range to the sensor surrounding such as a radar/laser/sonar). This allows treating a suite of sensors in an abstract manner when processing the data, the remaining of our processing code can be agnostic to the sensor type.

Your task is to:
(a) Create a base sensor class and expand it to a range of sensors
(b) Create a class to process this data that is agnostic to the sensor type and process that data to produce fused from sensor output data.

Due: As per Subject Outline

Contents

Assignment Specifics

Students are provided skeleton code in a1_skeleton folder that they need to use and develop from.

The physical sensors are collocated; the spatial separation of sensors can be disregarded. For the purpose of this assignment assume that all sensor data is obtained instantaneously (disregard acquisition time).

A) Create a Base Sensor Class (called Ranger) and two derived Sensor Classes (Laser and Sonar) that contain:

1. Pose Offset of Sensor (X,Y and angle relative to centre [Y axis] – anticlockwise positive)
2. Field of View of Sensor
3. Angular Resolution
4. Sequence Number (this is a count for the number of times the sensor has generated data)
5. Sensing Type
6. Data

The Ranger will need to inherit (use as base class) the **RangerInterface** header and implement its virtual functions without changing the function declarations (signatures).

B) Each Sensor will need to

1. Initialise all the required variables to enable using the sensor
2. Enable obtaining all the hardware specific fixed parameters of the sensor
3. Enable setting configurable parameters of the sensor
4. Inform if the values to be set are sane (supported), otherwise use previous values
5. Obtain sample sensor data at specific sensor angular resolution and within sensing range (Generated as random numbers within sensor specifications, normal distribution: mean 5m standard deviation 6m).

C) The RangerFusion Class will need to

1. Accept a STL container of Sensors
2. Accept a STL container of Cells (cells are essentially squares)
3. Determines the area to be covered by cone type sensors (as a union of sensing area). Solely for this task approximating the cone with a triangle is acceptable.
4. Produce a fusion of sensor readings at the supplied cells, dealing with readings that are on the boundary of the sensing range (max range). Cell Fusion is discussed in subsequent pages of this document. For this task the cone-based sensors can NOT be approximated as a triangle.
5. Return an STL container of raw unfused data (data must be that of raw data used for fusion)
6. You will need to inherit (use as base class) the **RangerFusionInterface** header and implement its virtual functions without changing the function declarations (signatures)

D) Create a static library called **ranger_lib,** from the classes specified.

E) Create a Main that

1. Initialises the sensors (for instance 1 Laser, 3 Sonars)
2. Queries the fixed sensor parameters
3. Sets sensor parameters (variable parameters, and offset)
4. Asks the user for number of cells and generates them
5. Reports the area to be covered by cone type sensors
6. Continuously calls (each second) the RangerFusion class grabAndFuse method (which that invokes generating data and fusion) and prints to screen the status of the cells (free , occupied, unknown)

Assessment Criteria

Criteria	Weight (%)	Description / Evidence	Further Breakdown	
Sensor classes exploits abstraction (encapsulation, inheritance and polymorphism) to cover a range of sensors	20	Use of special member functions for initialisation of classes such that they can be used with default settings. Correct use of access specifiers. Inheritance from base class, common data stored and generic functionality implemented solely in base class (no duplication). Classes that should not be available for instantiation are aptly protected.	Use of special member functions for initialisation of classes such that they can be used with default settings.	25
			I/O only in Main	15
			Correct use of access specifiers	15
			Functions take care of correlated dependencies (coupled settings)	15
			Classes that should not be available for instantiation are aptly protected.	15
			Inheritance from base class, common data stored and generic functionality implemented solely in base class (no duplication).	15
Proper code execution	35	User input of parameters works as expected. Data values reported as per sensor description. Data fusion reports correct number of readings, fused according to **specification**.	BONUS MARKS: Reports correctly the area covered by Sonars	10
			Fused data continuously reported ever second	10
			Sensor data reported as per sensor description	10
			Data fusion reports correct number of readings	10
			Laser behaves correctly with Cells	20
			Sonar behaves correctly with Cells	20
			Sensor Fusion performs with several different sensor configurations and Cells	40
Unit Testing	15	Unit tests created that guarantee performance of underlying algorithms for Sensor to Cell Interaction	BONUS MARKS Reports correctly the area covered by Sonars (at least 5 different tests with varying number of sonars – must be at least 2 sonars and there must be a test with 3 sonars).	20
			Laser behaves correctly with Cell (at least 5 different tests, varying sensor orientations, cell location and size)	50
			Sonar behaves correctly with Cell (at least 5 different tests, varying sensor orientations, cell location and size)	50
Documentation	10	ALL classes contain comments to understand methods, members and inner working (ie border case handling of fusion, method of fusing readings)	ALL source files contain useful comments to allow understand inner working (and description consistent with code)	80
			In addition to previous point, contains index.dox cover file which produces description of usage and anticipated output.	20
Modularity of software	20	Creates a library, links executable to the library.	Methods from rangerinterface implemented (NO CHANGES TO INTERFACE)	10

		No implicit coupling between classes that disables reuse. Appropriate use class declarations, definitions, default values and naming convention allowing for reuse. Sensor classes interface in ways allowing use of class in others contexts and expansion (more sensors can be added, sensors rearranged, can handle other offsets). No dependency on ordering of sensors, no “hard coded” values or assumptions of sensor order in fusion class.	Methods from rangerfusioninterface implemented (NO CHANGES TO INTERFACE)	10
			Data fusion class has no knowledge of ranger sub classes	10
			Invoking generating sensor data done consistent with skeleton code description	10
			Data fusion class can handle different ordering of sensors (no dependency on the ordering of sensors)	15
			Data fusion class can handle any angle offset	15
			Data Fusion class can handle additional sensors (Laser or Sonar)	15
			No hard coded values, constants used and values obtained by using getters and STL container sizes.	15

Sensor: Laser Rangefinder Scanner

Model	SICK-XL
Field of View	180 degrees
Angular Resolution	10 or 30 degrees
Max Distance	8.0m
Min Distance	0.2m

The laser returns N_L number of measurements (distances) which are related to the specified angular resolution. Each measurement is a single point in space. The laser scans anticlockwise.

For the purpose of this assignment assume that the measurement of the entire scan is obtained instantaneously (disregard acquisition time).

Sensor: Sonar

Model	SN-001
Field of View	20 degrees
Max Distance	10.0m
Min Distance	0.2m

The sonar returns 1 measurement (distance). The measurement is from an area of space (a cone in space). The distance returned is to the closest object within that area of space; therefore the return is a cone.

An Example Configuration of Sensors

Sensor	Field of View [°]	Angular Resolution [°]	Orientation [°]	Position (x,y) [m]
Laser	180	10	-30.0	1.0 , 2.0
Sonar1	20		+10.0	-2.0 , -0.5
Sonar2	20		+10.0	2.0 , -0.5

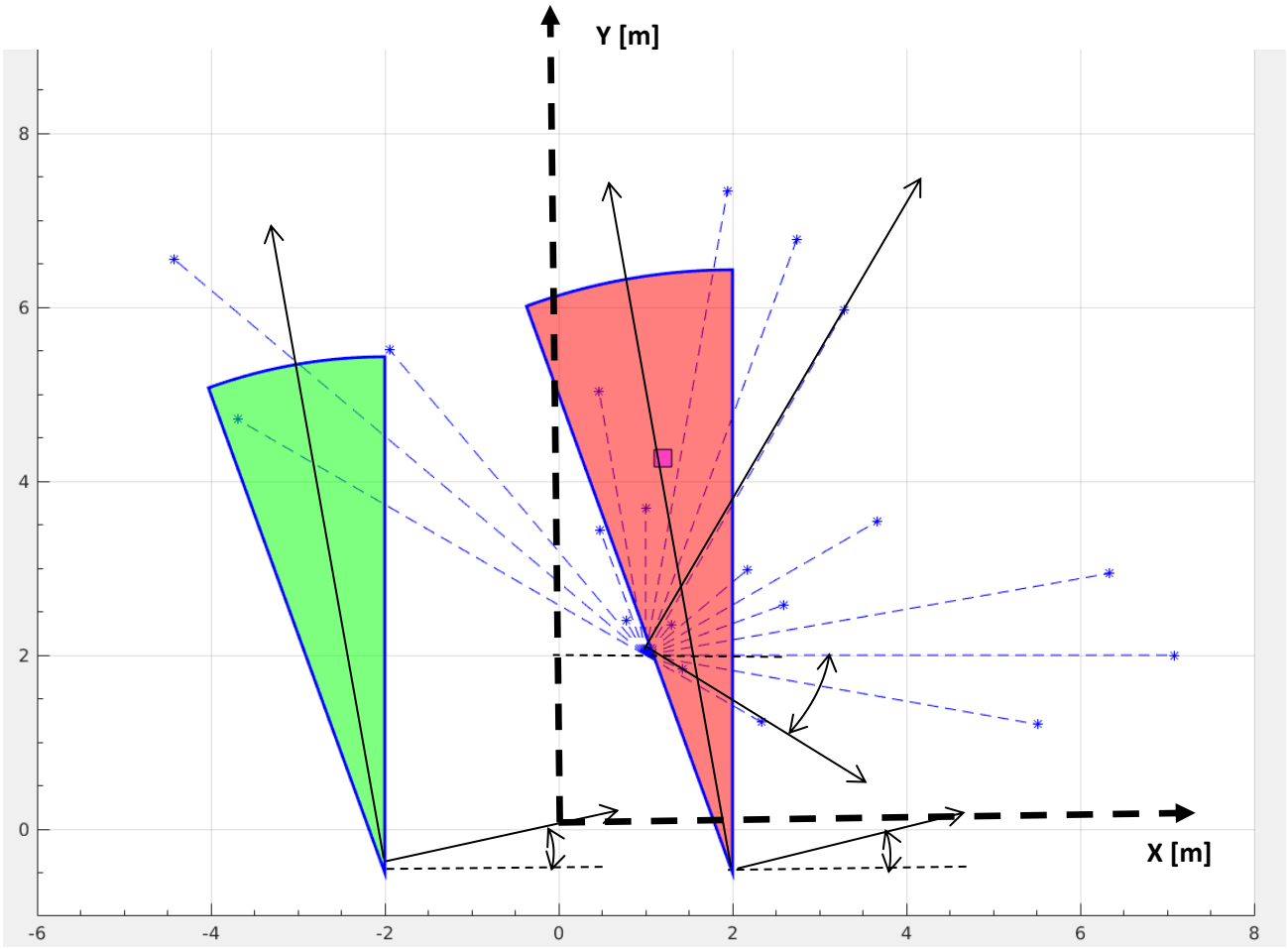
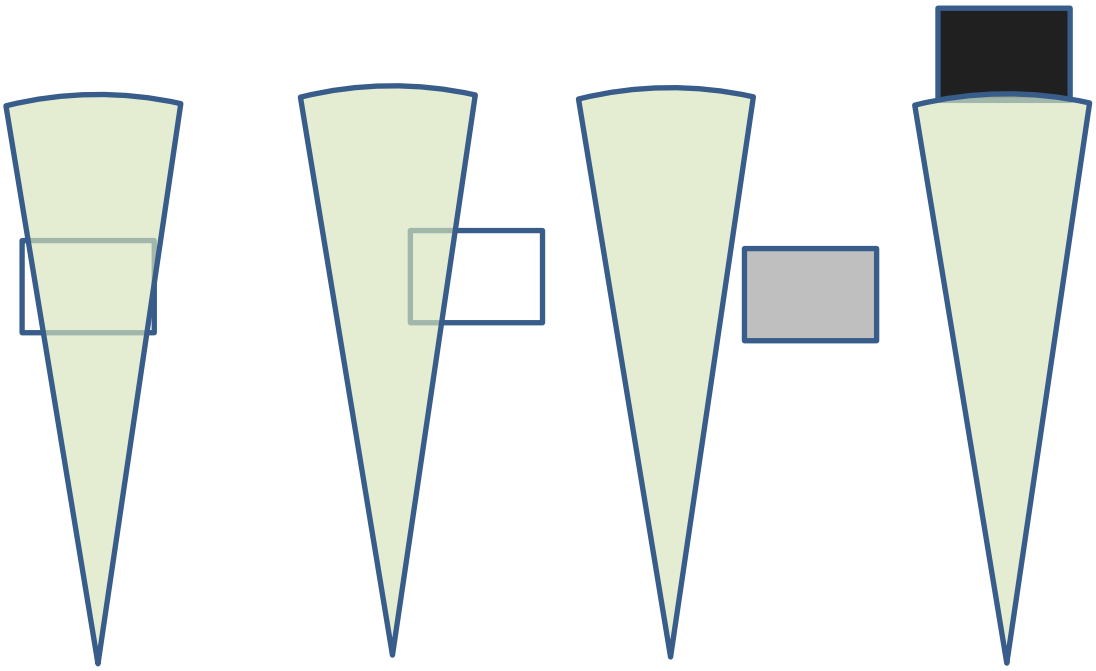


Figure 1 - Configuration 1 with sample readings; blue - laser; green - sonar 1; red - sonar 2. Each sensor has a coordinate frame associated with it (arrows noted) and therefore a position offset and an angular offset with respect to the global coordinate frame at x=0 y=0. To fuse data we have to transform any processing to the global coordinate frame.

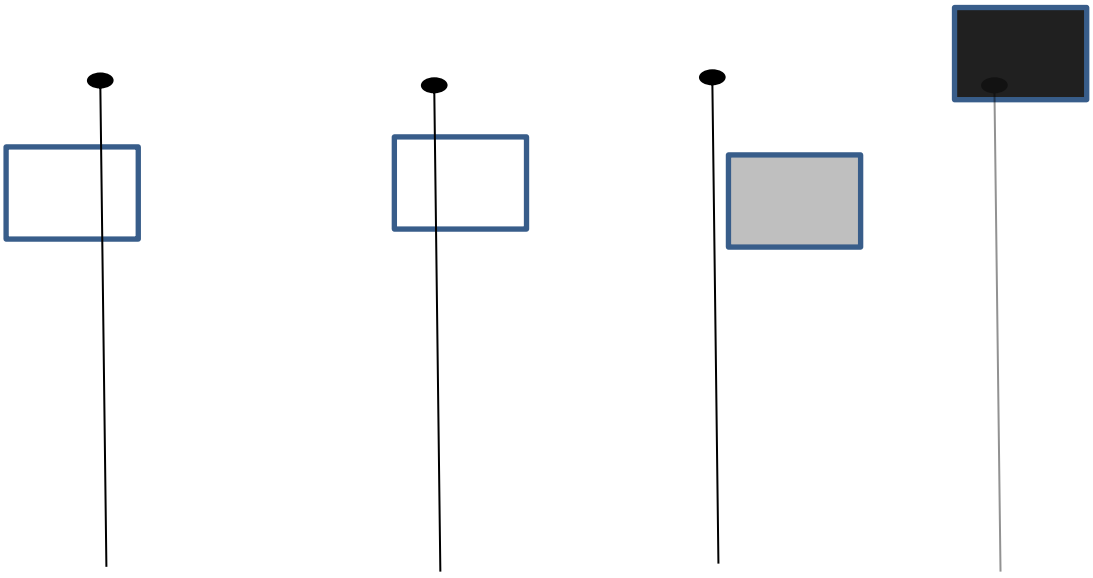
Cell Fusion Methodology

Cells can only have one of three states : Free , Unknown and Occupied (as per the typedef for State in cell.h). We will use the following colour scheme: White (Free), Grey (Unknown) and Black (Occupied) for the visualisation below.

When the sonar sensor interacts with the cell it changes the cells state, if the end of the sonar return lies inside the cell then the cell is occupied, if it goes through the cell then it is free. If it does not intersect with the cell then it is unaffected. The sonar must be treated as a cone shape (sector), it can not be treated as a triangle.



Similarly, when the laser sensor interacts with the cell it can change its state. If any single laser reading lies inside the cell then the cell is occupied, if a ray goes through the cell then it is free. If it does not intersect the cell then it is unaffected.

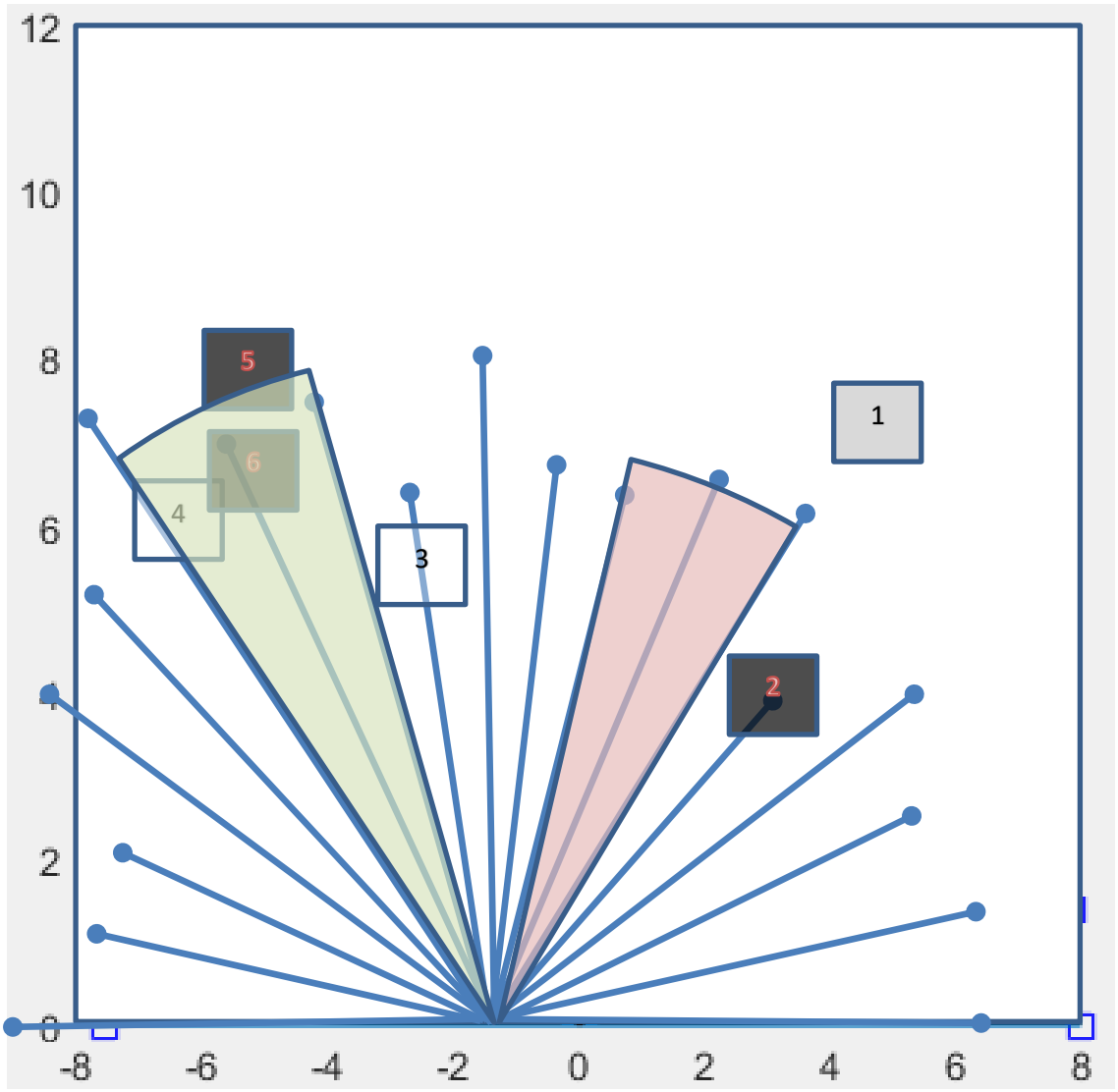


Occupancy takes precedence, if one sensor detects the cell as occupied then the cell is occupied.

Examples

If the Cell was placed in a few locations the resulting status will be:

Cell(s)	Value
1	Unknown
2, 5 and 6	Occupied (cell 6 is free by sonar and occupied by laser sensor – thus remains occupied)
3 and 4	Fee



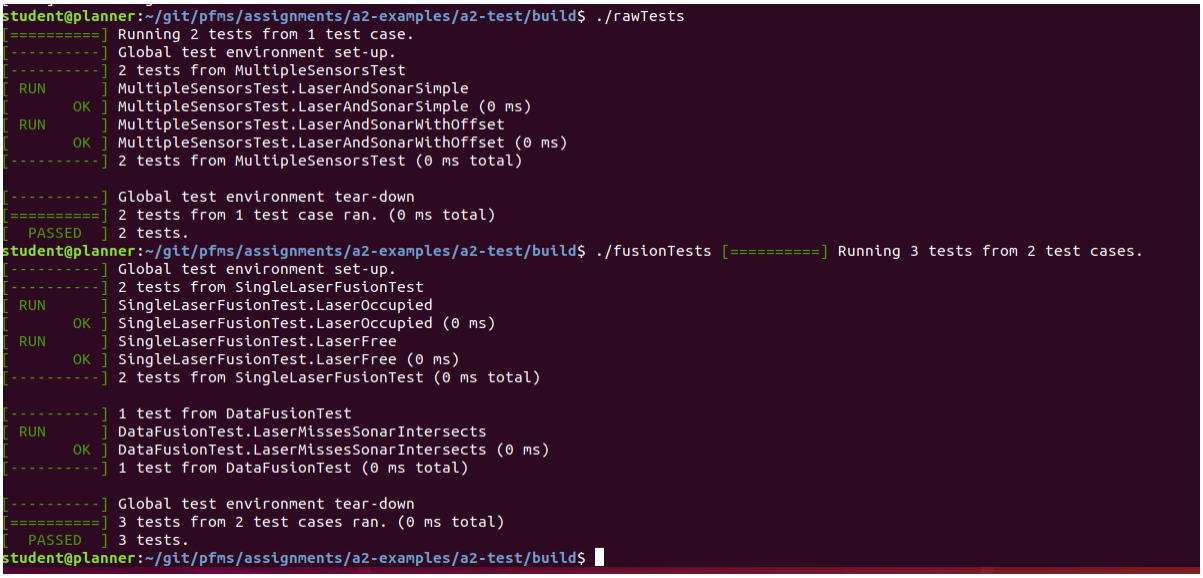
Automated testing of code

We have provided a test stub in the test folder of a2_skeleton. The testing will attempt to test the Sensor Classes developed, as well as RangerFusion. These ARE NOT THE ONLY tests we will be undertaking on your code, but they provide some indication of how your code will be thoroughly evaluated.

To compile and install testing students must install google testing framework (refer canvas for instructions).

Instructions to compile and run the tests on your own code are in the README.md file located within the test folder. Whenever testing, after making changed to your code, you should recompile the **test folder** and run the two test scripts.

If your tests are succeeding you should get all green in execution of tests.



FAQ (extract from Teams Discussion)

Do we need to implement the setAngularResolution function. From the specification the Sonar sensor doesn't need the resolution?

- All the setters need to be implemented, and they have a return value.
- The return value should be false if the setting is not supported (the sensor can not achieve that setting) OR the sensor does not have that setting (for instance sonar does not have an angular resolution).

Should I implement this function in ranger or laser and sonar?

Students need to decide whether to implement the function, whether in base class or derived class (whether it is pure virtual in ranger, or virtual in ranger or fully implemented in ranger). The base class SHOULD not know anything about the derived classes.

What should the getter for angular resolution return for the Sonar, since it does not have an angular resolution?

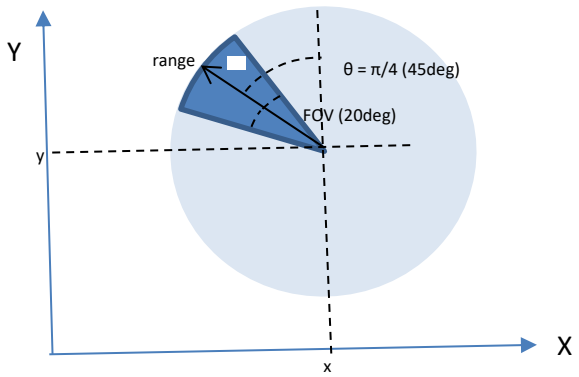
While FOV exists for both sensors, angularResolution does not. Getter for angular resolution of cone sensors can return zero. When your implementing the fusion strategy you should disregard angularResolution for CONE based sensors.

Is there a limitation for the sensor pose, for instance should position be limited to -10 to 10 or angle limited to some range?

The sensorPose should always be accepted and return a true. There is no restriction on the sensorPose: position can be anywhere in space (x,y) and orientation. I would strongly suggest to wrap (normalise) the angle. It is commonly normalised between -180 to 180 (so 270 gets wrapped to -90).

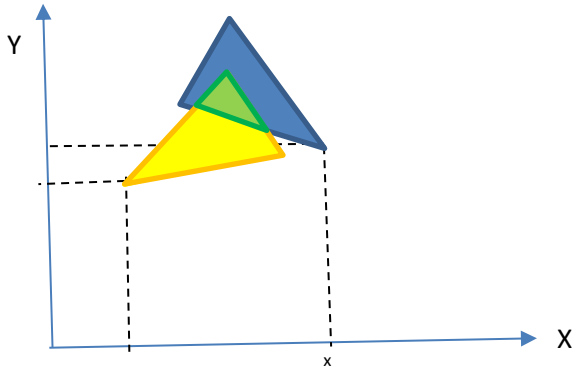
For cell interaction with Sonar, should we approximate the Sonar reading as a triangle instead of a cone (technically in 2d it's a sector)

NO, you can not approximate the sonar reading as triangle for the purpose of cell interaction. If your struggling conceptually with cell to cone interaction, examine below. Look at whether the cell is within (or at the) range reading returned by sonar. Then check the cell is within the sonar cone (sector), bound by the angle (theta) and it's FOV.



For area coverage with Sonar (union of area), can we approximate the Sonar reading as a triangle instead of a cone .

ONLY for the task of area of coverage can we resort to using triangles. The process for sectors is not straightforward and we might have to resort to a simpler process. Thereafter you can use area of convex polygon: http://www.mathwords.com/a/area_convex_polygon.htm. The area as union of two sensors is: area yellow + area blue – area green [green is overlap]



Should we consider these specific cases (1) the sensor is inside a cell (2) If a cell/part of a cell lies within/less than a sensor's min range

Please clearly articulate in your documentation whether you are looking at cell to ranger interaction from the origin of the sensor or from the minimum range. This is your prerogative, we will evaluate your specific implementation against these conditions. Note, the sensors can not return a reading below minimum range.

What is the role of getRawRangeData

getRawRangeData DOES NOT do any acquisition, it simply returns the data being used for fusion in vector of vectors

```
    /**
     * @brief Returns the raw data from all sensors in the ranger container within a vector of vectors
     * The raw data is updated every time a new fusion is requested (grabAndFuseDat). The raw data is the data used for collision checking. If
no fusion has occured the vector shall be empty.
     *
     * @return std::vector<std::vector<double>>  the outer elements of the vector related to the rangers, the inner elements of vector are the
respective range readings

     *
     * @sa grabAndFuseData
     */
    virtual std::vector<std::vector<double>> getRawRangeData() = 0;
```

grabAndFuseData both queries sensors for data and performs fusion

```
    /**
     * @brief Does two operations (1) Calls each ranger to generate data and uses this data to determine colisssions with provided container of
cells (2) Generates a 'fusion' of the data based on collision conditions as descibed in Assignment 1 specification
     *
     */
    virtual void grabAndFuseData() = 0;
```

Can we create additional functions and classes?

Certainly, yes, you can create own functions/classes and this is good practice. This is subject to following

- You cannot add functions to interface classes
- Only the functions in rangerinterface are accessed by fusion
- Only the functions in fusioninterface are accesed by main (unit tests we have supplied show these two things and marking criteria is also showing this)
- Additional classes can be added, long as above constraints are applied (that is you cannot expect someone using your code to have to create objects of the classes or functions from this class. The existing classes and functions can interact with them. (Check that our supplied unit tests have to pass).

The RangerFusion constructor seems to need to interact with cells

That is a typo, that comment can not stand as the RangerFusion in the constructor does not have any cells (none are passed to it at that stage) as per constructor declaration below.

```
class RangerFusion: public RangerFusionInterface
{
public:
    /**
    The Default constructor sets the cell centre to values within the #MAP_SIZE\n
    @sa RangerFusionInterface and @sa RangerInterface for more information
    */
  RangerFusion(std::vector<RangerInterface*> rangers);
```

Can I have additional constructors in the Classes supplied

You absolutely NEEED a default constructor, which takes no values) which initialises the sensor to default values (in case of angular resolution you need to decide what the default value is and articulate this in your code). If you do not have a default constructor none of the unit tests will pass (including those supplied) and we will not be able to mark execution.