Newton Method Based TDOA Solver

Author: Hao Wu, Date: 12/29/2018

Introduction

This application works as a solver for the TDOA locating problem by using Newton Method and sorts of improved Newton Method. It can generate the figures of your TDOA setup, like the contour of objective function, the iteration trajectory and the convergence map of multiple start points. Due to the lack of UI and some math problems unresolved, this application is still under construction. But two classes have been finished for correct and reliable usage. The following instruction will tell you how to build more things with this app.

Project Structure

This app has a simple structure like many tiny Makefile projects. The C++ header files are in /include, C++ source files are under /src and the main.cpp under root directory is the entrance of this whole project. The Makefile under root contains the compile instructions, you can modify it with your own preference compiler and filename style. All compiled object files will be listed under /bin. For the /test folder, you can write your testing source code here, but remember to add the compile info for it in Makefile. The folder /py contains two python scripts used for drawing figures. For how to use them please refer here.

Program Design

1. C++ Classes

`MyPoint2D`

A point class containing 2 coordinates x and y. You can interpret it as a two-dimension vector.

Attributes:

`x, y`

The coordinates of x and y

Methods:

`MyPoint2D()`

Constructor, initialize `x = 0, y = 0`

`MyPoint2D(a, b)`

Constructor, initialize `x = a, y = b`

`Operator +`

Add two `MyPoint2D` objects, equal to vector addition, return a new `MyPoint2D` object

`Operator –`

Vector subtraction, return a new `MyPoint2D` object

`Operator *`

Multiply a parameter for the vector, return a new `MyPoint2D` object

`norm()`

Return the norm of a vector, `double` type

`dist_square(p)`

Return the squared distance of current point object to point p, `double` type

`randomGenerate(a, b)`

Randomly set x, y in range [a, b], a must smaller than b

`randomGenerate(a, b, c, d)`

Randomly set x in [a, b], y in [c, d]

`Setting2`

This is the class designed for our 2nd TDOA setup, which has one target emitting signal and multiple sensors receiving the signal.

Attributes:

`sensors`

The position vectors of multiple receivers, implemented as a CPP `vector` of `MyPoint2D`

`timeDiffs`

The time differences of signal's arrival at sensors other than the first sensor. This means we use the first sensor (`sensors[0]`) as the reference receiver. Implemented as a CPP `vector` of type `double`.

`realTarget`

The point of real target (emitter, signal source), implemented as a `MyPoint2D` object

`startPoint`

The start point of iteration, `MyPoint2D`

`p`

Used as temporary container for iteration point, `MyPoint2D`

`c`

Signal speed, `double`

`noise`

Noise level, `double`

`iterTimes`

Record of the iteration times, `int`

`iterLimit`

The limitation on iteration times, `int`

`distLimit`

The threshold for norm difference between two adjacent iterations. If smaller than this threshold, we should terminate the iteration and claim finding a convergence answer. `double`

`funcValLimit`

The threshold of objective function value. If smaller than this threshold, we say our function achieves the optimal point. `double`

Methods:

`Setting2()`

Constructor, initialize an empty object of `Setting2`

`Setting2(arr_sensor, rt, speed)`

Constructor, initialize `sensors` with `arr_sensor`, `realTarget` with `rt` and `c` with `speed`

`setStart(sp)`

Set the start point as `sp`

`setLimits(iter_limit, dist_limit, func_limit)`

Set the limitations

`timeCompute()`

Compute the time differences of arrival and assign the answer to `timeDiffs`. This method is used for simulation. DO NOT use this method if you have the real time differences of arrival data, use `setTimeDiffs(tds)` instead.

`setTimeDiffs(tds)`

Set `timeDiffs` with `tds`.

`locate()`

The pure Newton method solver to locate the real target. Return a `MyPoint2D` object.

`locate_TR(ita_s, ita_v, miu_shrk, miu_xpnd, tr_upper)`

The trust region Newton method to locate the real target. Return a `MyPoint2D` object. Augments are:

> `ita_s`: the lower bound of rho which is satisfied
>
> `ita_v`: the lower bound of rho which is super-satisfied
>
> `miu_shrk`: the ratio we want to shrink our trust region
>
> `miu_xpnd`: the ratio we want to expand our trust region

`tr_upper`: the upper bound of the size of our trust region

`objectFunc(point)`

Return a `double` type value of our objective function with the `point` as input.

`modelFunc(point, obj_func_val, grad_x, grad_y, H00, H01, H11)`

Return a `double` type value of our model function with current iteration's `point`, objective function value `obj_func_val`, gradient in x `grad_x`, gradient in y `grad_y` and Hessian matrix's elements `H00, H01, H11`.

`isPosDef(H00, H01, H11)`

Return whether the Hessian is positive definite.

`cg_steihaug(H00, H01, H11, grad_x, grad_y, maxit, radius, errtol)`

The conjugate gradient (Steihaug) method for solving trust region subproblem. Return the next moving step as a `MyPoint2D` object. Augments are:

`H00, H01, H11`: Hessian

`grad_x, grad_y`: gradient

`maxit`: maximum iteration for conjugate gradient method

`radius`: trust region radius

`errtol`: the tolerance of error

Please refer more details by reading Steihaug's paper.

`cauchy_point(H00, H01, H11, grad_x, grad_y, radius)`

The Cauchy point solver for trust region subproblem. Return the next moving step as a `MyPoint2D` object. Augments are:

`H00, H01, H11`: Hessian

`grad_x, grad_y`: gradient

`radius`: trust region radius

2. C++ Main Function

The main.cpp contains the main function of this whole project. Currently it works only for a single run, which is that you must change some pieces of code in main function to make it do different missions. For example, the current main function only has one function called `func_11_03(string, bool)`, and the string is used to choose the arrangement of sensors, like arranging sensors in a line or in a circle, while the bool is used to determine whether you want to get the convergence map or the iteration record. To be specific, the `func_11_03("circle", false)` will compute the data needed for building a convergence map of circle arranged sensors.

When you infer the `func_11_03(string, bool)`, you can find the settings of many parameters like the sensor arrangement, target position, the start points position etc. You can try out the parameters to see the different answers of this Newton based solver, but you have to change them directly in the main.cpp file, and then compile the project using cmake (or nmake on Windows), and then run the main executable file. The `func_11_03(string, bool)` will generate sorts of csv files for drawing figures, you can read them but DO NOT write them.

3. Python Scripts

The python scripts are used to draw figures. Although C++ has some figure drawing library, but I am not familiar with them. So, for simplify the project, I just use python to do the drawing thing.

`conv_map.py`

Draw the convergence map. After you run the main function for getting the convergence map, run this script in your terminal with `python conv_map.py "figure_name_as_you_wish.png"`.

`iter_rec.py`

Draw the iteration record of one iteration. After you run the main function for getting the iteration record, run this script in your terminal with `python iter_rec.py`.