100/100

Robotics

Group 2_Fri_E
Assignment 4

Garv Asrani (0494851) Nicolas Hahn (501703)
Björn Altmann (351352)

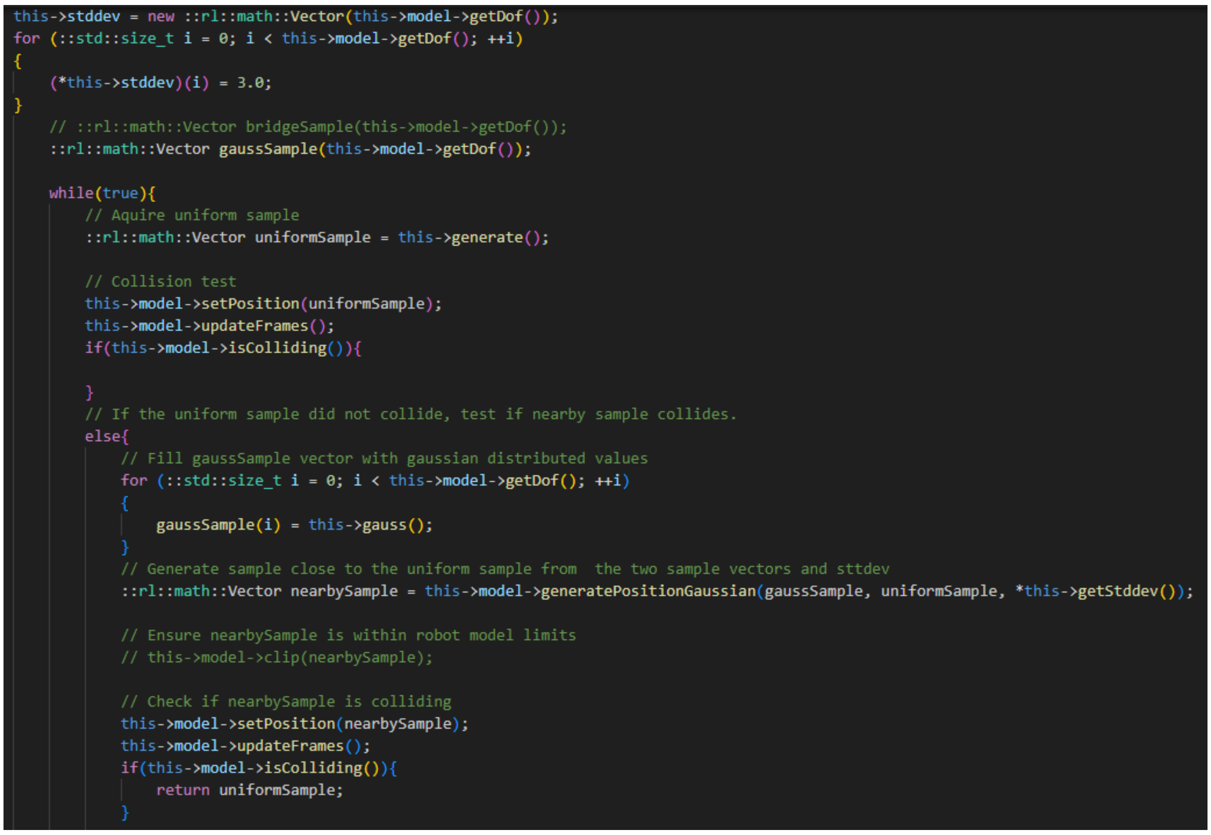
February 18, 2024

Contents

A. Proposed Extension - Gauss Sampling

A.1. Extension Description

First, we propose a modification to the random-sampler in "YourSampler.cpp" (For the code snippet, see Figure 1. We keep the method to generate a uniform sample the same. Then we overwrite the other sampler-function "generateCollisionFree()". In it, we first find a non-colliding, uniform sample. Then we generate a nearby sample using normal distribution, and check if this sample collides with the model. If it does, we have found a sample near an obstacle and return it to be added to the tree. Otherwise we repeat the process by generating a new, non-colliding sample.



```
this->stddev = new ::rl::math::Vector(this->model->getDof());
for (::std::size_t i = 0; i < this->model->getDof(); ++i)
{
    (*this->stddev)(i) = 3.0;
}
    // ::rl::math::Vector bridgeSample(this->model->getDof());
    ::rl::math::Vector gaussSample(this->model->getDof());

    while(true){
        // Aquire uniform sample
        ::rl::math::Vector uniformSample = this->generate();

        // Collision test
        this->model->setPosition(uniformSample);
        this->model->updateFrames();
        if(this->model->isColliding()){

        }
        // If the uniform sample did not collide, test if nearby sample collides.
        else{
            // Fill gaussSample vector with gaussian distributed values
            for (::std::size_t i = 0; i < this->model->getDof(); ++i)
            {
                gaussSample(i) = this->gauss();
            }
            // Generate sample close to the uniform sample from  the two sample vectors and sttdev
            ::rl::math::Vector nearbySample = this->model->generatePositionGaussian(gaussSample, uniformSample, *this->getStddev());

            // Ensure nearbySample is within robot model limits
            // this->model->clip(nearbySample);

            // Check if nearbySample is colliding
            this->model->setPosition(nearbySample);
            this->model->updateFrames();
            if(this->model->isColliding()){
                return uniformSample;
            }
```

Figure 1: CodeSnippet for the Gaussian Sampler in YourSampler.cpp

A.2. Expected Result

By only adding nodes to the tree that are near collisions, and thus potentially near our narrow gap, we expected an improvement in runtime. Adding nodes and vertexes to the tree is the most expensive operation, while resampling and collision checking can be done in an efficient manner.

A.3. Actual Result

With a standard deviation of 3.0, there was no difference in average runtime. While the tests with RrtConConBase sometimes generated an outlier where a solution took from 1.5 to 2 minutes to find and that behaviour could not be replicated with the Gauss-sampler under this standard deviation, this more likely than not can be attributed to random chance. The same can be said for outliers in the opposite direction, where RrtConconBase generated run-times as low as 11s. The average number of vertices in the tree goes down as expected, while the number of collision queries is increased.

A.4. Result Explanation

One possible explanation for the fact that there is no reduction in run-time is that the collision-checking of robLib is not as efficient as it could be, and the large increase in needed collision checks balance out the expensive edge creation. Another explanation is that the standard deviation chosen to generate the Gauss sample is not optimal. We ran tests with the standard deviations of 0.5, 3.0 and 7.0. A deviation of 0.5 led to an even higher runtime, on average exceeding the runtime of RrtConConBase as well as reducing the number of nodes in the tree even further.

B. Proposed Extension - Adaptive Dynamic Domain

B.1. Extension Description

We implemented a dynamic domain adjustment for nodes to the solve() function in YourPlanner.cpp. A code-snippet of the implementation can be found seen in Figure 2. We add a "radius" field to the VertexBundle structure. Then we randomly choose a sample, generate the node aNearest, and check if the calculated distance to the nearest node is larger than its radius field. If it is, we sample again. If it is not, we have found a sample within the nodes' "domain" and attempt the connect step.

If we successfully connect the nodes we multiply our radius by 1 plus a set radius expansion factor, and from there proceed to attempt to connect to the other tree and potentially solve.

If we instead do not manage to connect the nodes, we reduce the radius (or domain) of the node and then make sure it is not below a lower boundary.

B.2. Expected Result

We expected a reduction in runtime as we further reduce the number of nodes that will be checked - and added - to the tree, as well as reducing the number of collision checks needed by reducing the chance nodes that have collided often will be expanded.

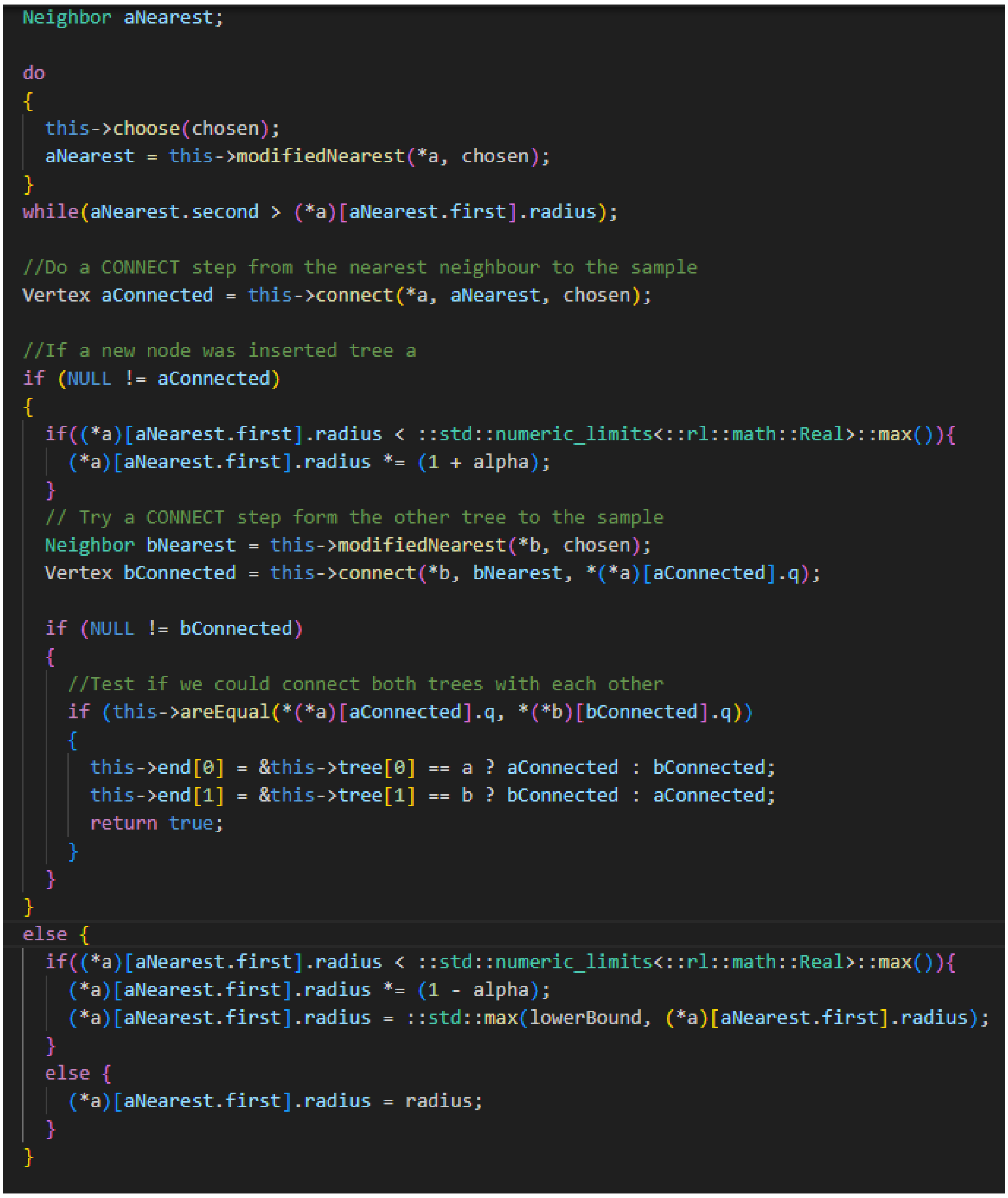
B.3. Actual Result

The average runtime actually increased compared to RrtConConBase and using only Gauss-sampling. The tests were run with a initial radius of 20.0, an expansion factor of 0.05, and a lower boundary for the radius of 2.0.

Comparing two tests with similar runtime, one with Gauss-sampling (hereinafter GS) at 30.135 s and one with Dynamic Domain + Gauss-sampling (hereinafter DD+GS) at 30.880 s, DD+GS added $\tilde{2}600$ less nodes to the tree and needed $\tilde{1}10,000$ less collision checks. Compared to a run of RrtConConBase which found a solution in 31.674 s, DD+GS added $\tilde{7}500$ less nodes to the tree. RrtConConBase performed over 722,000 collision checks during that run, exceeding the number of checks needed for DG+GS by $\tilde{1}20,000$.

B.4. Result Explanation

A possibility for the observed increased runtime of our Planner compared to RrtConConBase despite improvement in the other metrics is the choice of the radius, expansion factor and lower boundary variables used to compute the domain modification. Reducing the radius to 10 led to a lower runtime on average, though the variance - at least in the ten tests we performed - was higher,ranging from $\tilde{1}7$ s at the lowest to $\tilde{6}3$ s at the highest. In return, many more node additions and collision checks were performed. Another issue is that we potentially run through the "nearest" function a lot more often than without dynamic domain, which in turn iterates through all vertices each time. While we sharply reduce the amount vertices, this still gets computationally expensive and could lead to the observed worsening of the runtime.



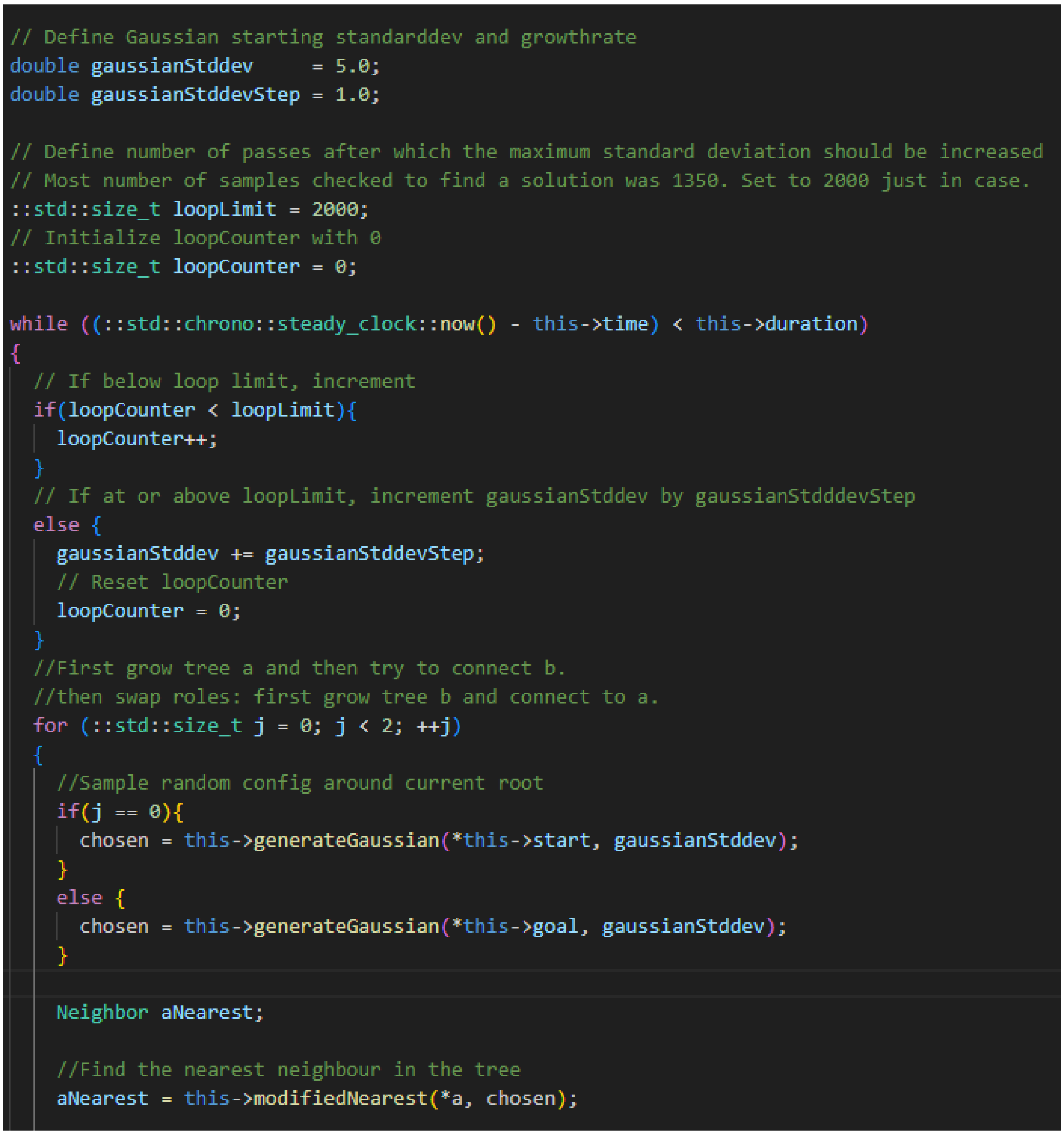
```
Neighbor aNearest;

do
{
  this->choose(chosen);
  aNearest = this->modifiedNearest(*a, chosen);
}
while(aNearest.second > (*a)[aNearest.first].radius);

//Do a CONNECT step from the nearest neighbour to the sample
Vertex aConnected = this->connect(*a, aNearest, chosen);

//If a new node was inserted tree a
if (NULL != aConnected)
{
  if((*a)[aNearest.first].radius < ::std::numeric_limits<::rl::math::Real>::max()){
    (*a)[aNearest.first].radius *= (1 + alpha);
  }
  // Try a CONNECT step form the other tree to the sample
  Neighbor bNearest = this->modifiedNearest(*b, chosen);
  Vertex bConnected = this->connect(*b, bNearest, *(*a)[aConnected].q);

  if (NULL != bConnected)
  {
    //Test if we could connect both trees with each other
    if (this->areEqual(*(*a)[aConnected].q, *(*b)[bConnected].q))
    {
      this->end[0] = &this->tree[0] == a ? aConnected : bConnected;
      this->end[1] = &this->tree[1] == b ? bConnected : aConnected;
      return true;
    }
  }
}
else {
  if((*a)[aNearest.first].radius < ::std::numeric_limits<::rl::math::Real>::max()){
    (*a)[aNearest.first].radius *= (1 - alpha);
    (*a)[aNearest.first].radius = ::std::max(lowerBound, (*a)[aNearest.first].radius);
  }
  else {
    (*a)[aNearest.first].radius = radius;
  }
}
```

Figure 2: Code Snippet of the solve() function modified for adaptive dynamic domain

C. Proposed Extension - Gauss-Sampling around Root Nodes

C.1. Extension Description

We seek to exploit the physical limits of the Puma robot which should cause either the goal or start to be close to the narrow passage for most applications. By generating Gauss-Samples with a given starting standard deviation around the root of the current tree we attempt to find such a passage. To avoid the case of being unable to find a solution for planning problems where the narrow passage is not near either goal or start, we increase the standard deviation every X number of loops until we are back to sampling in the entirety of the robots' operational space. A code snippet of this extension can be seen in Figure 3.



```
// Define Gaussian starting standarddev and growthrate
double gaussianStddev     = 5.0;
double gaussianStddevStep = 1.0;

// Define number of passes after which the maximum standard deviation should be increased
// Most number of samples checked to find a solution was 1350. Set to 2000 just in case.
::std::size_t loopLimit = 2000;
// Initialize loopCounter with 0
::std::size_t loopCounter = 0;

while ((::std::chrono::steady_clock::now() - this->time) < this->duration)
{
  // If below loop limit, increment
  if(loopCounter < loopLimit){
    loopCounter++;
  }
  // If at or above loopLimit, increment gaussianStddev by gaussianStdddevStep
  else {
    gaussianStddev += gaussianStddevStep;
    // Reset loopCounter
    loopCounter = 0;
  }
  //First grow tree a and then try to connect b.
  //then swap roles: first grow tree b and connect to a.
  for (::std::size_t j = 0; j < 2; ++j)
  {
    //Sample random config around current root
    if(j == 0){
      chosen = this->generateGaussian(*this->start, gaussianStddev);
    }
    else {
      chosen = this->generateGaussian(*this->goal, gaussianStddev);
    }

    Neighbor aNearest;

    //Find the nearest neighbour in the tree
    aNearest = this->modifiedNearest(*a, chosen);
```

Figure 3: Code Snippet of the solve() function modified to sample around root nodes with increasing standard deviation

C.2. Expected Result

We expected a reduction in runtime for cases where a solution can be found by only sampling in a growing area in the vicinity of the start and goal (see Figure 4. Due to the physical limitations of the puma robot, most narrow passage problems should have either the start or the goal (or both) near a narrow passage. In the worst case we expect the runtime to be at most marginally slower than RrtConConBase since we increase the search radius back to include the whole workspace.

C.3. Actual Result

For our given start and stop, as well as narrow passage location, this method leads to a vastly reduced runtime. With a starting standard deviation of 0.5 and a growth rate of 1 per 2000 loops we got results between$\tilde{5}$ s and $\tilde{1}0$ s. We also tried with a higher starting standard deviation, 5.0, which resulted in incredibly fast results between as low as $\tilde{0}.3$ s to $\tilde{7}$ s. However, the price paid for this speed is an incredibly convoluted path from start to goal due to the low number of added vertices, even after optimization. We decided that a slightly longer runtime of the path finding algorithm was a reasonable trade-off for getting reasonably good, smooth and fast paths most of the time.

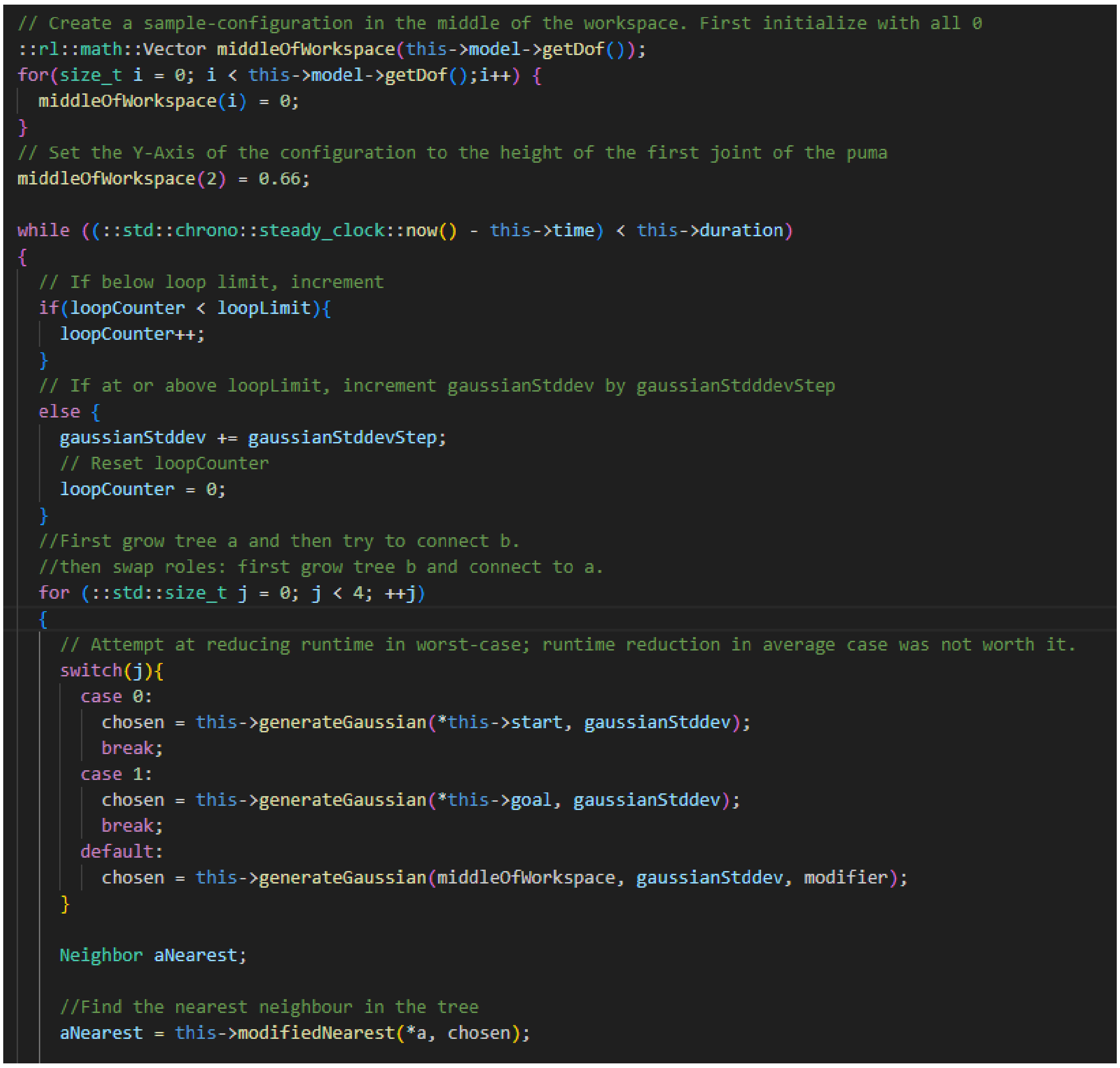
C.4. Result Explanation

In our given problem, the narrow passage that needs to be navigated is indeed close to our goal node. As a result, we quickly find samples inside the narrow passage and are able to "escape" and connect with the opposite tree. In the worst case where the narrow passage is both equidistant between the start and goal node and the distance is maximized for the limits of the robot-model, the average runtime should be equal to, or slightly longer than RrtConConBase, as we increase the space in which we sample configurations every 2000 loops. If the worst case is known, the parameters of starting standard deviation, deviation growth and loops before increasing the deviation can be tweaked. In our opinion, sacrificing some planning speed when dealing with the worst case for a large increase in performance for average cases is worth it.

D. Proposed Extension - Additional significant point for sampling

D.1. Extension Description

We attempted to potentially reduce the time to find a path with Extension C by introducing another significant point to sample around. We decided on the middle of the workspace. To get a configuration to use as the mean for our Gauss sampling, we crated a configuration of size this-¿model-¿Dof() and set the y-axis to 0.66 (the height of the robot limb). We multiplied the sampled configuration by 1 + a small constant to mostly avoid generating samples inside the robot, which would automatically collide.



```
// Create a sample-configuration in the middle of the workspace. First initialize with all 0
::rl::math::Vector middleOfWorkspace(this->model->getDof());
for(size_t i = 0; i < this->model->getDof();i++) {
  middleOfWorkspace(i) = 0;
}
// Set the Y-Axis of the configuration to the height of the first joint of the puma
middleOfWorkspace(2) = 0.66;

while ((::std::chrono::steady_clock::now() - this->time) < this->duration)
{
  // If below loop limit, increment
  if(loopCounter < loopLimit){
    loopCounter++;
  }
  // If at or above loopLimit, increment gaussianStddev by gaussianStdddevStep
  else {
    gaussianStddev += gaussianStddevStep;
    // Reset loopCounter
    loopCounter = 0;
  }
  //First grow tree a and then try to connect b.
  //then swap roles: first grow tree b and connect to a.
  for (::std::size_t j = 0; j < 4; ++j)
  {
    // Attempt at reducing runtime in worst-case; runtime reduction in average case was not worth it.
    switch(j){
      case 0:
        chosen = this->generateGaussian(*this->start, gaussianStddev);
        break;
      case 1:
        chosen = this->generateGaussian(*this->goal, gaussianStddev);
        break;
      default:
        chosen = this->generateGaussian(middleOfWorkspace, gaussianStddev, modifier);
    }

    Neighbor aNearest;

    //Find the nearest neighbour in the tree
    aNearest = this->modifiedNearest(*a, chosen);
```

Figure 4: Code Snippet of the solve() function modified with the additional significant node for sampling

D.2. Expected Result

We expected a potential reduction in runtime efficiency.

D.3. Actual Result

The expected increase in average runtime did occur, however it increased from an average of 5359.35 ms to over 10.00 ms. We decided that almost doubling the average runtime for most narrow-passage problems the Puma could face was not worth a potential reduction in worst case runtime. If the planner was to be used for a worst case scenario, the variables for the greedy gauss sampling around root nodes can be modified.

E. RrtConConBase vs. YourPlanner

	RRTConConBase	RRTConConBase (reversed)	YourPlanner	Your Planner (reversed)
avgT	21311.56	27964.01	5359.35	6011.93
stdT	7223.10	22034.33	1631.97	2100.16
avgNodes	10021.60	11462.90	3621.90	3901.70
avgQueries	513993.30	571474.10	126450.60	140258.80

Table 1: Comparison of average metrics for given and modified planner, run both forward and in reverse.

F. Conclusion

In the end, we attempted five extensions to try and increase the runtime of the program: Bridge-Sampling, in which the robot did not which to move despite following the steps shown in the lecture; my assumption is that we did not set the standard deviation correctly to ever find two points that collide and have a non-colliding sample in the middle. Potentially we should first have sampled a number of samples near colliding points with Gauss, and then used a saved array of the collision points to try and find "bridge" nodes.

Next, we went back a step and instead implemented Gauss-Sampling, as well as Adaptive Dynamic Domain, however, the result was simply less collision tests and added nodes, no reduction in runtime. Instead, with both active, we even saw it increase. While these Extensions are no longer used in our final code, they can still be seen and tested by commenting out the "live" solve() in YourPlanner.cpp, and un-commenting the solve() function following the comment "SOLVE WITH GAUSS-SAMPLER AND ADAPTIVE DYNAMIC DOMAIN".

Third, we implemented a greedy sampling strategy only sampling around the roots of our two trees independent of our first to Extensions. We were seeking to epxloit the fact that, due to the physical constraints of the Puma robot, most narrow-passage problems would have either the start or the goal nodes relatively close by. This resulted in a sharp drop in runtime, and correspondingly expanded nodes and collision tests. However, we assume there are some worst-case configurations of the planning problem that will result in a slightly larger runtime than RrtConConBase.

Finally, we attempted to introduce an additional node to sample around in our greedy sampling strategy. The hope was to reduce or prevent the predicted loss in runtime compared to RrtConConBase during a scenario where both goal and start were far away from the narrow passage. However, this almost doubled the runtime, and we decided that increase was not worth a potential reduction during our worst-case prediction.

G. Appendix

Student Name	Ext. A-Impl.	Ext. B-Impl.	Ext. C-Impl.	Ext. D-Impl.	Documentation
Nicolas Hahn	x		x		
Garv Asrani	x				
Björn Altmann	x	x	x	x	x

Table 2: Task done by which student