Rube Goldberg Machine : Project Report by Group 05

Hardik Kothari
110050029
khardik84@gmail.com

Navin Chandak
110050047
navinchandak92@gmail.com

Preetham Sreenivas
110050073
preethamsreenivas@gmail.com

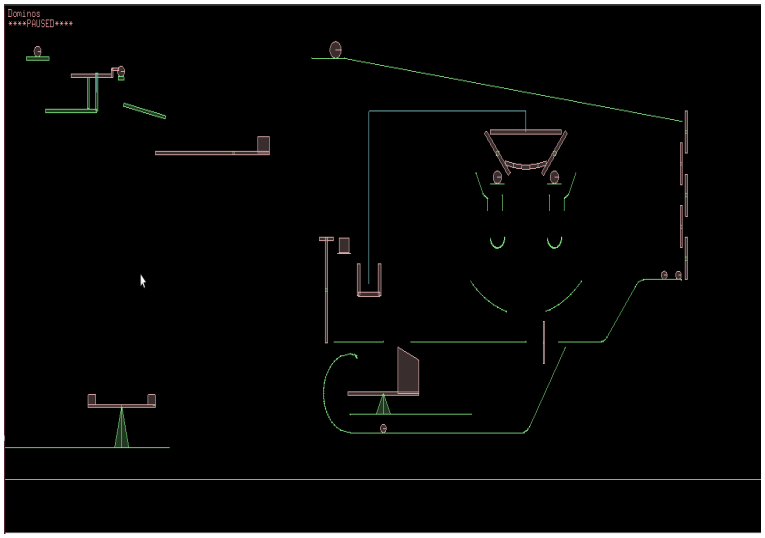
April 10, 2013

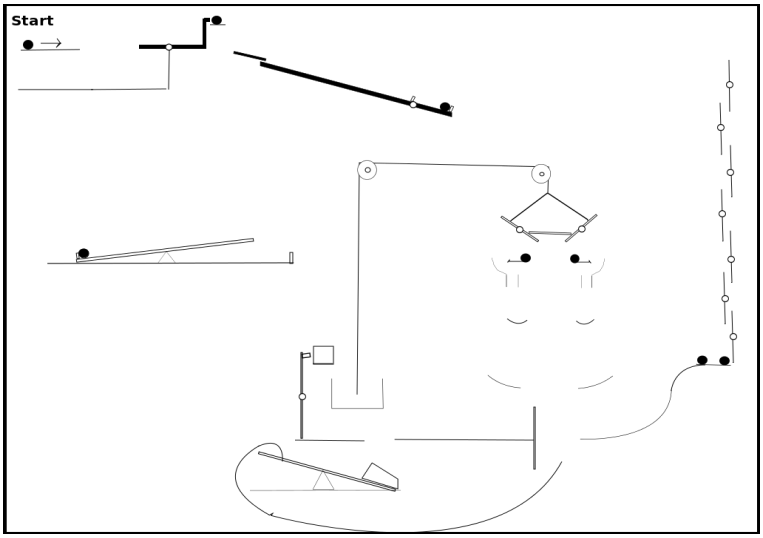
1 Introduction

This report explains the rube goldberg machine [3] designed by us, its difference from the original design and the interesting parts in our design. We also analyse our code using various techniques.

2 Present and Original Design

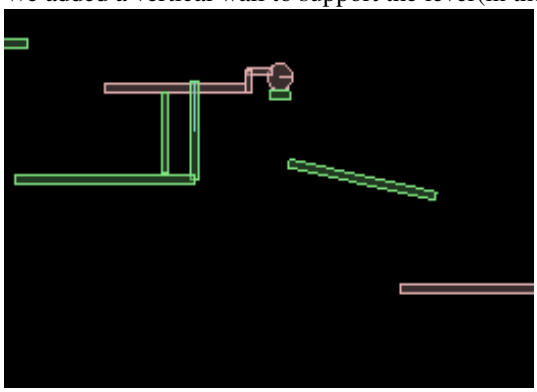
Following are the images of the present design(left) and the original design(right) respectively :

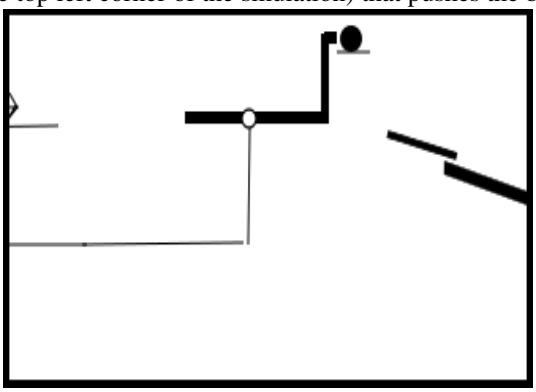




There are some minor additions to our original design to make our simulation work better. There are just three of them which are listed below:

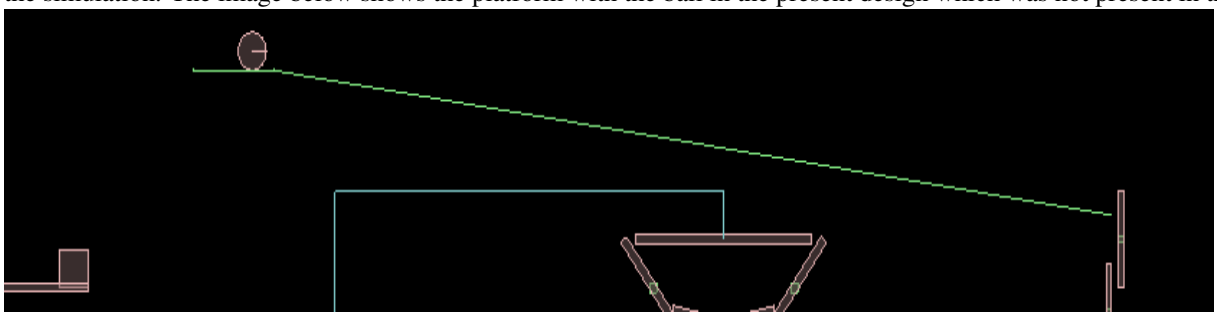
- We added a vertical wall to support the lever(in the top left corner of the smulation) that pushes the ball.



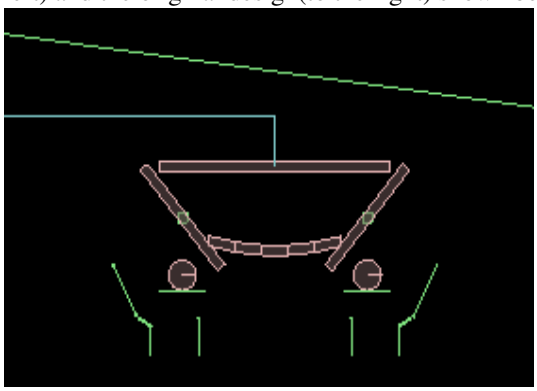
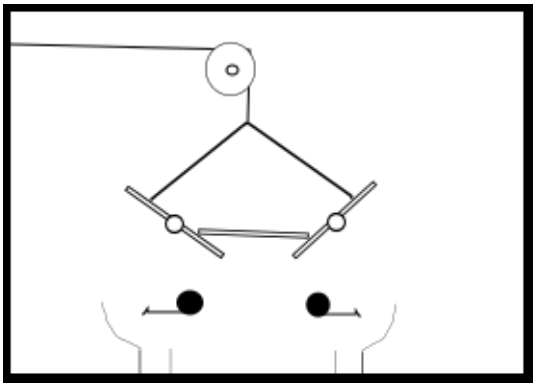


- In our original design, a ball falls on the wedge(in the bottom left corner) that pushes a block placed on it which takes a projectile path and finally hits a set of multiple levers(on the rightmost part of the simulation). But a part of that motion of the block would not be visible as it would go above the visible area of the simulation. So we included a platform with a ball on it in the topmost part of the

simulation. Now the projectile motion of that block becomes small and it hits the ball on the platform which takes forward the rest of the simulation. The image below shows the platform with the ball in the present design which was not present in the original design.



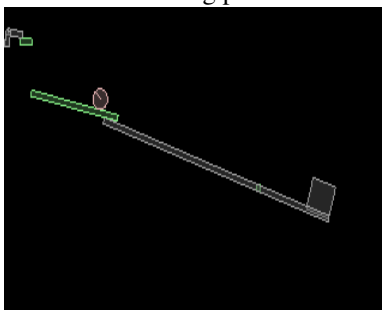
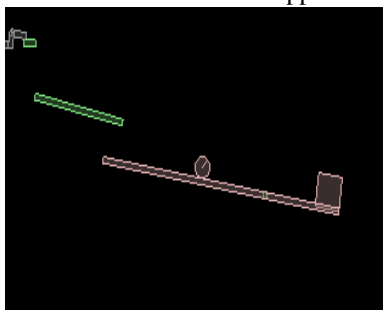
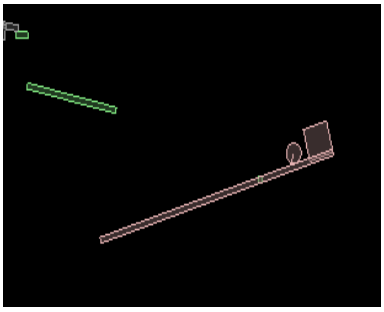
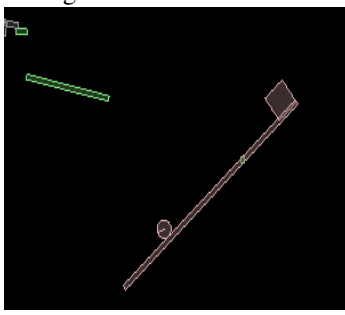
- This one is a very minor change in the rod connected to the pulley which you can figure out by the images of the present design(to the left) and the original design(to the right) shown below:

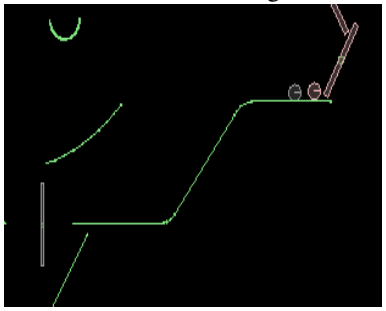
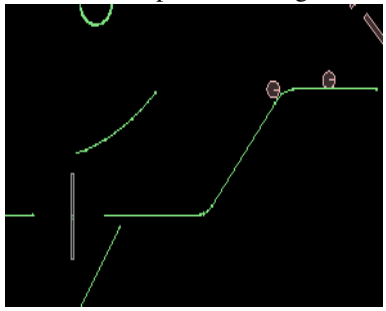
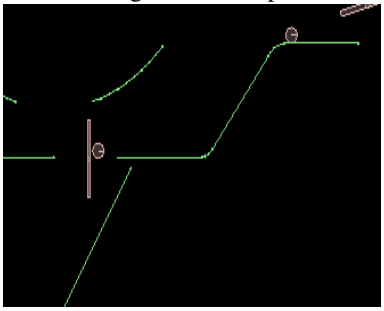
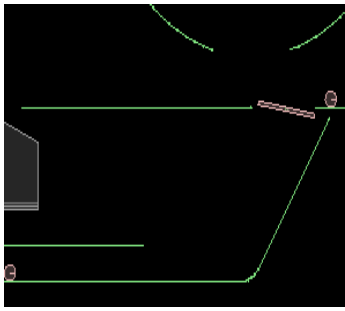
3 Interesting Parts of our Design

We would like to point out some parts of our simulation design that make it interesting. [5] [4]

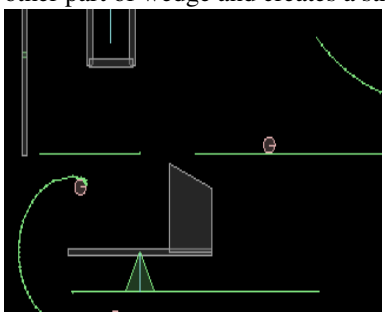
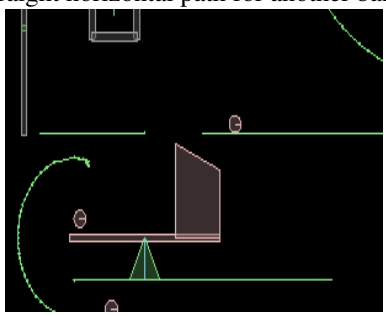
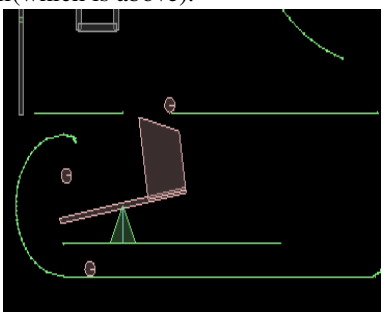
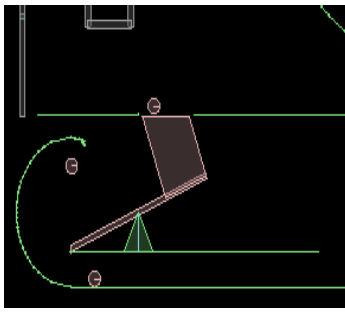
- There is a revolving platform on which a ball comes from its upper left side and it releases it towards its bottom right side.

- We think that this is the most interesting part of our simulation. There are two balls kept nearby each other on a horizontal surface and there is a vertical lever to the right. When the lever hits one of the balls, that ball(which is towards the right side) gets into motion and hits the other ball which goes forward and creates path for it to go forward. The images below explain the scenario:

- This is also an interesting part of our design. The ball(which is below) hits the wedge at a perfectly appropriate time which lifts the other part of wedge and creates a straight horizontal path for another ball(which is above).

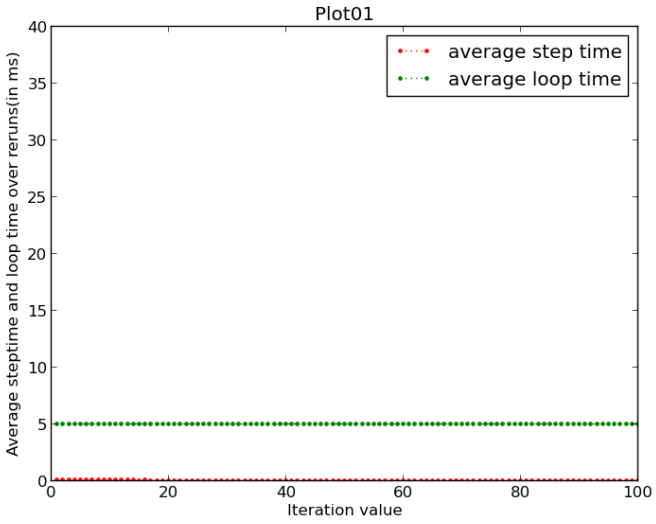
- Our Rube Goldberg Machine ends by creating a smiley. This, we think, is another interesting aspect of our simulation.

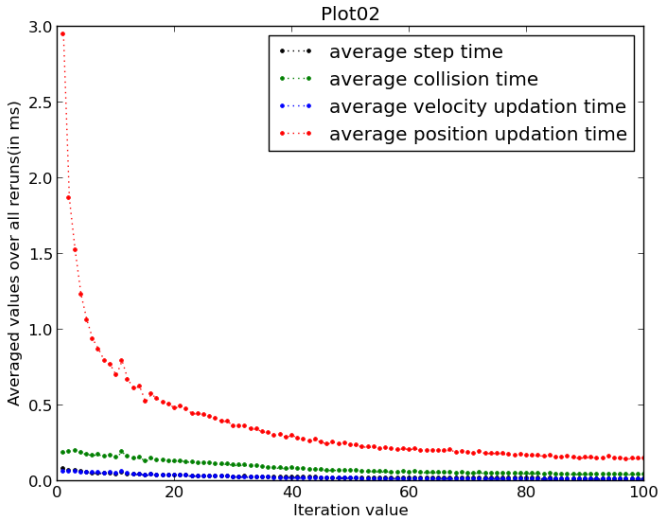
4 Analysis of plots

[1]

4.1 Analysis of plot01(image below) : [Average (StepTime and LoopTime over reruns) vs number of iterations]

- The stepTime and looptime are very close to each other. This is meaningful because the only major process that is happening inside the loop is Step function and apart from the step function, only a few addition, subtraction and reading is happening.



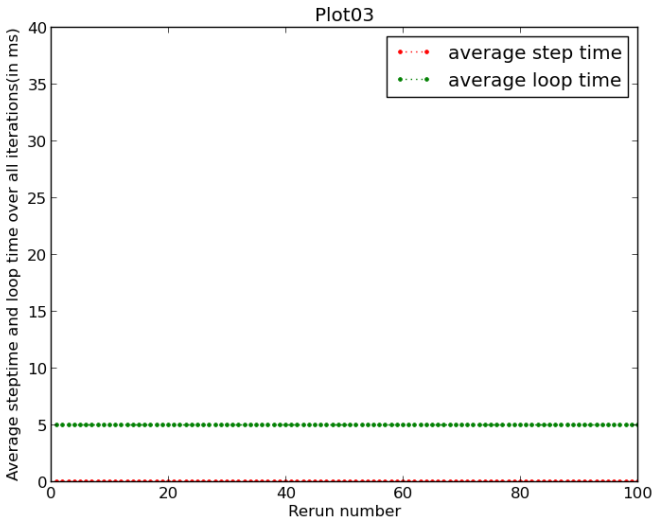


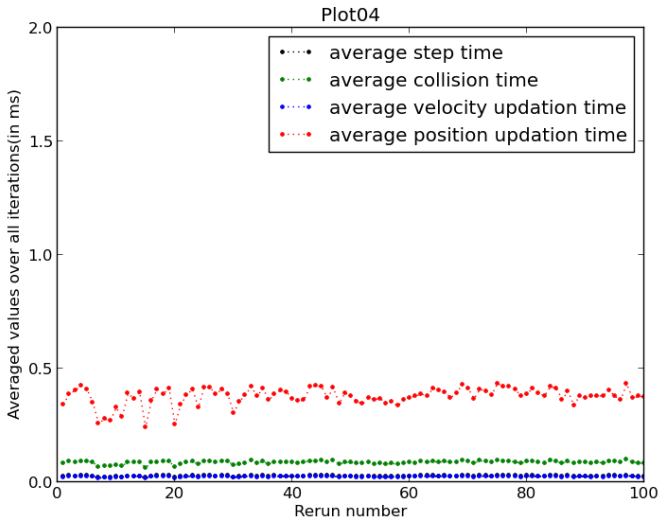
4.2 Analysis of plot02(image above) : [Average (StepTime, CollisionTime, Velocity Time, Position update time over reruns) vs number of iterations]

- In this case, an important thing to notice is that the collision time, velocity time and position update time add upto the step time. This proves that major things which happens inside the step function are collision handling, velocity update handling and position update handling.
- The ratio of the time taken by collision handling, velocity update handling, and position update handling is approximately same for different values of iteration number. Which is to say that, if more collisions are happening, then there is a greater need to update velocities and positions, which makes sense.

4.3 Analysis of plot03(image below) : [Average of step time and looptime over iteration values vs rerun number]

- The averages of looptime and steptime are same because of reasons mentioned earlier.
- The avg looptime over iteration values changes slightly for different rerun index. The important thing to notice that it is not constant. And the most likely reason for this is the other processes that runs in CPU during the execution of this program. So say when the nth iteration is running, the load on CPU might increase because of which the avg of step time over iteration values might increase for that particular rerun number. This justifies rerunning the program many times.



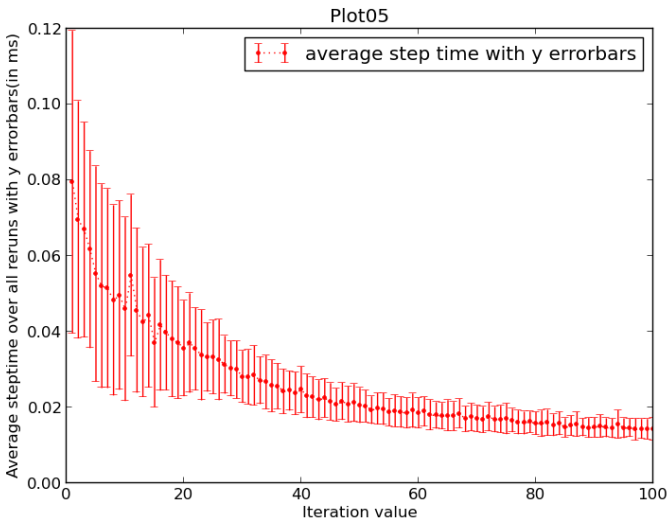


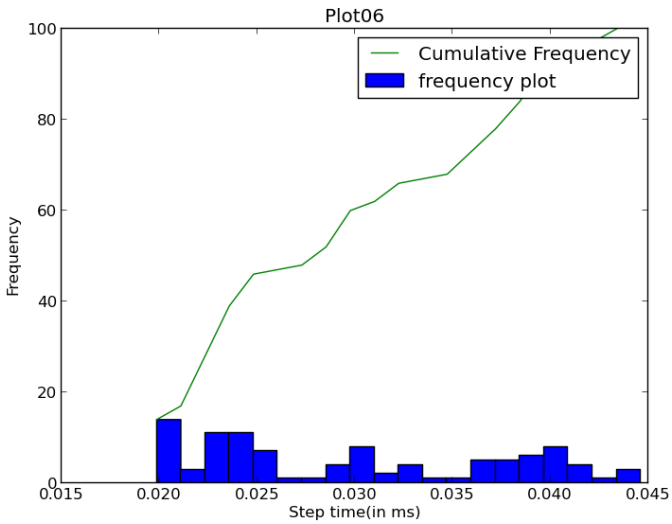
4.4 Analysis of plot04(image above) : [Average of (step time, collision time, position update time, velocity update time over iteration values) vs the rerun number]

- The analysis is very similar to the previous case. Again here the point that the sum of position update time, velocity update time, and collision update time is almost equal to step time and that the ratios are similar.

4.5 Analysis of plot05(image below) : [Average of iteration values over reruns (with error bars) vs the iteration values]

- The new feature is the error bars over plot1. The error decreases as the iteration values increases. On closely observing the step times from the actual data files, many a times the step times initially are very high and many times they are pretty low. This is perhaps because of the way in which the OS allocates CPU to the program. If the CPU is busy then the program gets less CPU initially and it increases into the program. However if the CPU is free then the program quickly gets sufficient CPU power. However when we consider higher iteration values, then the initial disturbance gets averaged out due to which the error is not so visible.





4.6 Analysis of plot06(image above) : [frequency of a particular step time vs the step time for iteration value 29]

- The step time for different reruns are almost in a close range of 0.045ms to 0.050 ms. Approximately 70% of values are in this range. However in some exceptional cases, the step time came out to be upto 0.1 ms where the CPU load might have increased.

5 Code Profiling Report

We profiled the code using gprof for 1L iterations in both release-mode and debug-mode.
First, let us analyze the report of debug-mode.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  8.87      0.91     0.91 72468736     0.00     0.00  operator-(b2Vec2 const&, b2Vec2 const&)
  8.73      1.80     0.89 290849558     0.00     0.00  b2Vec2::b2Vec2(float, float)
  7.26      2.54     0.74 69275049     0.00     0.00  operator*(float, b2Vec2 const&)
  5.88      3.14     0.60                             b2ContactSolver::SolveVelocityConstraints()
  3.28      3.47     0.34 32146717     0.00     0.00  operator+(b2Vec2 const&, b2Vec2 const&)
  3.14      3.79     0.32                             b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&)
  3.04      4.10     0.31                             b2World::Solve(b2TimeStep const&)
  2.94      4.40     0.30                             b2World::SolveTOI(b2TimeStep const&)
  2.94      4.70     0.30                             b2Mul(b2Transform const&, b2Vec2 const&)
  2.55      4.96     0.26                             b2Island::Solve(b2Profile*, b2TimeStep const&, b2Vec2 const&, bool)
  2.45      5.21     0.25                             b2Cross(float, b2Vec2 const&)
  2.35      5.45     0.24                             b2Dot(b2Vec2 const&, b2Vec2 const&)
  2.16      5.67     0.22                             b2Mul(b2Rot const&, b2Vec2 const&)
  1.57      5.83     0.16 39512189     0.00     0.00  b2Cross(b2Vec2 const&, b2Vec2 const&)
  1.37      5.97     0.14 37598186     0.00     0.00  float b2Max<float>(float, float)
  1.37      6.11     0.14                             b2TestOverlap(b2AABB const&, b2AABB const&)
  1.37      6.25     0.14                             b2Max(b2Vec2 const&, b2Vec2 const&)
  1.32      6.39     0.14 34351218     0.00     0.00  float b2Min<float>(float, float)
  0.98      6.49     0.10                             b2Sweep::GetTransform(b2Transform*, float) const
  0.93      6.58     0.10                             b2Cross(b2Vec2 const&, float)
  0.88      6.67     0.09                             b2MulT(b2Rot const&, b2Vec2 const&)
  0.88      6.76     0.09                             b2ContactSolver::SolvePositionConstraints()
  0.88      6.85     0.09                             b2ContactSolver::InitializeVelocityConstraints()
  0.88      6.94     0.09                             b2PolygonShape::ComputeAABB(b2AABB*, b2Transform const&, int) const
  0.83      7.03     0.09                             b2EdgeSeparation(b2PolygonShape const*, b2Transform const&, int, b2PolygonShape cons
    t*, b2Transform const&)
  0.78      7.11     0.08 35321736     0.00     0.00  b2Body::IsAwake() const
  0.78      7.19     0.08 19602750     0.00     0.00  b2Vec2::operator+=(b2Vec2 const&)
  0.78      7.27     0.08                             b2Distance(b2DistanceOutput*, b2SimplexCache*, b2DistanceInput const*)
  0.78      7.35     0.08                             b2DynamicTree::InsertLeaf(int)
  0.78      7.43     0.08                             b2Vec2::operator-() const
  0.78      7.51     0.08                             b2Mat33::Solve22(b2Vec2 const&) const
  0.69      7.58     0.07 83893869     0.00     0.00  b2Vec2::b2Vec2()
"debug_1L" 1748L, 146349C                                                                1,1           Top
```

- First of all, we observe that simple functions like operator-, operator *, b2Vec2 (function for creating a new vector) are taking the maximum percentage of time. This shows that Box2D is a very computationally heavy process which relies on a lot of computation. In 1L steps , the b2Vec2 constructor is called a whopping 374743427 times, which means that it is called 3.7K times in each call of Step function.

- SolveVelocityConstraints of the b2RevoluteJoint class takes up a lot of time, perhaps because implementing joints are computationally expensive and required continued operations to find out their position. Morover our simulation uses revoluteJoints in many places
- Functions like solve of b2World class (which finds islands, integrates and solves constraints and solves position constraints) , solveVelocityConstraints of b2ContactSolver(which solves the velocity constraints, the tangential forces on body, friction etc), solveTOI of b2World class (which finds Time of Impact contacts and solve them) also takes up a lot of time.
- Again operations like finding max, finding min, taking the cross product or taking dot products of vectors takes up quite a lot of time despite being small functions sheerly because of the large number of times it is being called.

Now let us consider the report of release-mode and consider the differences and similarities. Note that in the release-mode we have compiled Box2d in release-mode and also applied O3 optimizations to the code which lies in src folder. So we expect the gcc compiler to make many modifications to the code, making code inline(where possible), replacing recursive functions with a loop (where possible easily), moving variables from the stack to the register, etc. So lets analyze the release-profile clearly.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 20.29      0.28     0.28                             b2ContactSolver::SolveVelocityConstraints()
 10.15      0.42     0.14                             b2World::Step(float, int, int)
  7.97      0.53     0.11                             b2World::SolveTOI(b2TimeStep const&)
  7.97      0.64     0.11                             b2TimeOfImpact(b2TOIOutput*, b2TOIInput const*)
  5.80      0.72     0.08                             b2Island::Solve(b2Profile*, b2TimeStep const&, b2Vec2 const&, bool)
  4.35      0.78     0.06                             b2PolygonShape::ComputeAABB(b2AABB*, b2Transform const&, int) const
  4.35      0.84     0.06                             b2EdgeSeparation(b2PolygonShape const*, b2Transform const&, int, b2PolygonShape const
*, b2Transform const&)
  3.62      0.89     0.05                             b2DynamicTree::InsertLeaf(int)
  2.90      0.93     0.04                             b2Distance(b2DistanceOutput*, b2SimplexCache*, b2DistanceInput const*)
  2.90      0.97     0.04                             void b2DynamicTree::Query<b2BroadPhase>(b2BroadPhase*, b2AABB const&) const
  2.17      1.00     0.03                             b2ContactSolver::SolvePositionConstraints()
  2.17      1.03     0.03                             b2ContactSolver::b2ContactSolver(b2ContactSolverDef*)
  2.17      1.06     0.03                             b2Mat33::Solve22(b2Vec2 const&) const
  1.45      1.08     0.02                             b2DynamicTree::MoveProxy(int, b2AABB const&, b2Vec2 const&)
  1.45      1.10     0.02                             b2PulleyJoint::SolvePositionConstraints(b2SolverData const&)
  1.45      1.12     0.02                             b2RevoluteJoint::SolveVelocityConstraints(b2SolverData const&)
  1.45      1.14     0.02                             b2ContactManager::Collide()
  1.45      1.16     0.02                             b2StackAllocator::Free(void*)
  1.45      1.18     0.02                             b2StackAllocator::Allocate(int)
  1.45      1.20     0.02                             b2Body::SynchronizeFixtures()
  1.45      1.22     0.02                             b2Timer::GetMilliseconds() const
  0.72      1.23     0.01                             b2TestOverlap(b2Shape const*, int, b2Shape const*, int, b2Transform const&, b2Transfo
rm const&)
  0.72      1.24     0.01                             b2CollidePolygons(b2Manifold*, b2PolygonShape const*, b2Transform const&, b2PolygonSh
ape const*, b2Transform const&)
  0.72      1.25     0.01                             b2CollideEdgeAndPolygon(b2Manifold*, b2EdgeShape const*, b2Transform const&, b2Polygo
nShape const*, b2Transform const&)
  0.72      1.26     0.01                             b2CollidePolygonAndCircle(b2Manifold*, b2PolygonShape const*, b2Transform const&, b2C
ircleShape const*, b2Transform const&)
  0.72      1.27     0.01                             b2FindMaxSeparation(int*, b2PolygonShape const*, b2Transform const&, b2PolygonShape c
onst*, b2Transform const&)
  0.72      1.28     0.01                             void b2BroadPhase::UpdatePairs<b2ContactManager>(b2ContactManager*)
"release_1L" 371L, 27430C                                                                                    1,1           Top
```

- The first thing we notice is that most of the functions which consumed a lot of time in debug-mode are absent in the release-mode profiling. So operator-, operator*, b2Vec2, b2Max have most likely been made inline by the compiler.
- The level of inlining is clear from the fact that 186 functions were detected by gprof in the debug-mode, while only 53 functions were detected by gprof in the release-mode.
- The percentage of time taken by SolveVelocityConstraints of b2ContactSolver is here the most time consuming function and it is taking 20% time in release-mode ; It was taki/ng just 5% time in debug-mode. This is most probably because when functions are made inline, then the time taken taken by those functions earlier are now counted in the function from where these functions were being called. The same reasoning perhaps applies for all the functions which are now new in the list of functions which are consuming high percentages of time taken by the program. Such things happens for many functions.
- It is notable that the Step function in the release-mode is not present at all in the debug-mode. On seeing the code for step function, we find out that step function calls many functions and does nothing else. Due to which its presence in debug-version was negligible; However it appears in the release-mode because of inlining of the functions which step function calls.

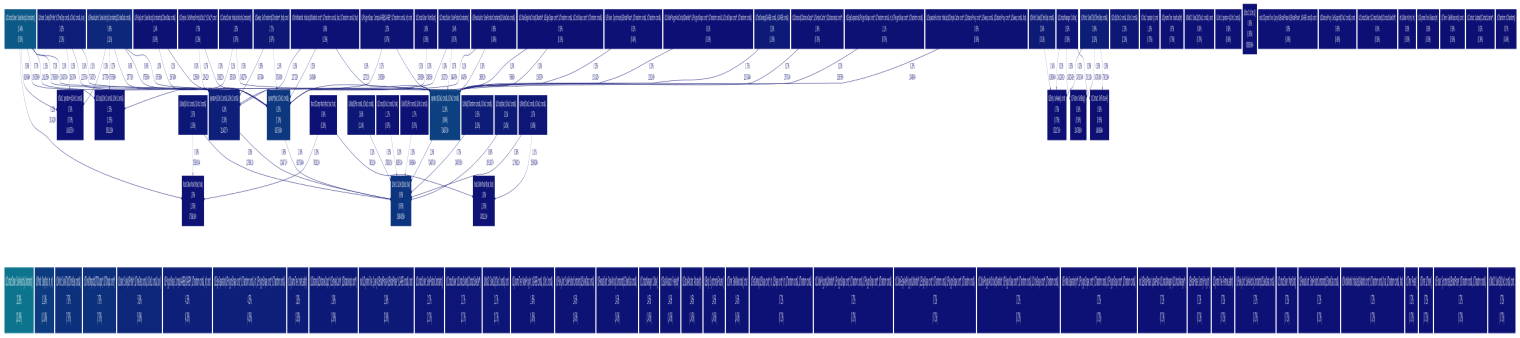
6 Other differences in release and debug modes

There is a marked difference in the time taken by code in the release and debug mode. The time taken by the code in the debug-mode is 18.29s, while the time taken in release-mode is just 1.942s. (These are values for 100K steps). This indicates that the code written by us are

really inefficient as compared to what the compiler gives us.
The size of the executable in debug-mode is 1.4M, while in release-mode it is just 690K.

7 Call Graphs

[2] The callgraphs for debug and release mode respectively :



The call graphs represent the callee-caller relations. It shows all the functions that called a particular function and also all the functions that are called by it.

For release mode, we can see that there are no arcs coming out of any functions which implies that all the callee functions are made inline to the caller functions , where as there are a large number of function calls which are present in callgraphs for debug mode.

References

[1] Matplot lib for generation of plots. `http://matplotlib.org/`.

[2] Used for creation of call graphs. `https://code.google.com/p/jrfonseca/wiki/Gprof2Dot`.

[3] Wikipedia page for rube goldberg. `http://en.wikipedia.org/wiki/Rube_Goldberg`.

[4] Walker Halliday, Resnick. *fundamentals of physics, 9th Ed.* John Wiley and Sons, 2010.

[5] H.C.Verma. *Concepts of Physics,Part1*. Bharati Bhawan, 2006.