ImperiSim 2.0: Bidirectional Communication Between SolidWorks and KISS-Sim
for 3D Robot Simulation with Motors, Sensors, and Physics (Part 1)
Jeremy Rand and Marty Rand
Norman Advanced Robotics
jeremy.rand@ou.edu, mario.tech.boy@gmail.com

# ImperiSim 2.0: Bidirectional Communication Between SolidWorks and KISS-Sim for 3D Robot Simulation with Motors, Sensors, and Physics (Part 1)

## 1 Introduction

When Jeremy developed ImperiSim in 2010 [1], it received significant attention, even receiving a mention by Botball founder and NASA Mars rover engineer Dr. David Miller at GCER 2010. The ability to use SolidWorks's physics engine for 3D autonomous robot simulation was certainly a new concept in Botball. But there were some serious shortcomings in the 2010 release. Most notably, the entire program had to finish executing in KISS-Sim before SolidWorks could start the physics simulation. This meant that the program's sensors couldn't react to the physics from SolidWorks; they could only see the primitive KISS-Sim physics, effectively making the robot blind. Since Botball encourages intelligent robots with robust sensor systems, this severely limited ImperiSim's practical usefulness in Botball.

But we don't give up easily. In this paper, we discuss how we rewrote ImperiSim to support sensors, including touch, rangefinder, and camera sensors, in full 3D with the SolidWorks physics engine. But first, the obligatory disclaimer.

DISCLAIMER: Simulators operate using a simplified version of real-life physics, and therefore do not perfectly reflect a robot's behavior, particularly in complex scenarios. While simulators are a convenient tool, the most accurate way to assess whether your robot will function correctly is to run it on a well-built physical game board. Teams which rely on simulators until a week before the tournament are just about guaranteed to fail. We take no responsibility for poor tournament performance due to differences between simulated and actual robot behavior.

## 2 Why Bidirectional Communication is Problematic

SolidWorks has three Motion Study types: Animation, Basic Motion, and Motion Analysis. ImperiSim uses Basic Motion, as it is the most advanced Motion Study type which is supported in the SolidWorks Student Design Kit, which is free for Botball students. All three Motion Study types have API functionality, but their level of feature support varies. The problem we encountered is that when a Motion Study begins calculating, the API stops responding until the calculation has completed. There is an API feature called External Motors, which permits motor behavior to be implemented in a VBA routine; the VBA routine is called every time the motor

moves. We wanted to utilize this with a VBA routine which communicated with KISS-Sim, but after our attempts failed, we checked with Dassault Systemes tech support and were told that External Motors only work with Motion Analysis, and there is no API feature which allows motors to be modified during a Basic Motion calculation.

## 3 A Solution: Simulation Segmentation

Our method for circumventing the bidirectional issue involves splitting the simulation into very small segments. For a 500ms segment length, SolidWorks receives motor data from KISS-Sim, runs a 500ms simulation, and then sends sensor data back to KISS-Sim. The resulting simulation state is then copied to the beginning of a new simulation, and the process repeats. While this isn't quite as temporally precise as External Motors, the system works well. A good bit of API code is necessary to handle the segmentation, however.

## 4 Communication Methods

We initially considered using sockets to transfer data between SolidWorks and KISS-Sim, but VBA doesn't have socket libraries, and most of the sample code available online looked difficult to implement. We instead used a little-known feature of Windows called Named Pipes. A Named Pipe is a bidirectional communication medium which permits any two Windows applications to exchange data. VBA and C sample code for Named Pipes was readily available online.

We initially considered a complex protocol, but decided that simply transferring a packet of data bytes back and forth with no significant protocol overhead was the best way to go, given our limited available time. Each segment of simulation, KISS-Sim transfers a Motors packet, and SolidWorks replies with a Sensors packet. The ability to detect errors is limited with this method, as there is no checksumming or handshaking involved. If bytes are dropped, all the following bytes in the packet will be corrupted, and the simulation will likely freeze as it waits for a byte that never arrives. Since Named Pipes aren't subject to data loss, this isn't usually an issue, but if we were to make a mistake in our code, forming a mismatch in the number of bytes sent and received, a frozen SolidWorks or KISS-Sim would be the only signal we had regarding what went wrong. So far, we have not yet had a mismatch error during development.

## 5 Packet Format: Motors

The motor packet format consists of a struct for each motor, each of which contains a velocity, a goal position, and a flag indittaing whether the motor has a goal or if it is turning indefinitely. The mav() function, for example, would set the velocity and disable the “has goal” flag, while the mrp() function would set the velocity and goal and enable the “has goal” flag. The protocol currently calls for 14 motors: 4 CBC motors, 4 CBC servos, 3 locomotion motors, and 3 Create LSD motors. The locomotion motors were originally intended to correspond to X, Y, and Theta positions, but it was later decided that such kinematics would be handled on the SolidWorks side (so that they could integrate with the trajectory generator), so they are now simply used for the two Create drive motors, with the last locomotion motor being unused/deprecated.

A “Terminate” flag is also sent; this allows SolidWorks to request that KISS-Sim end the simulation immediately.

```
// KISS-Sim to SolidWorks Packet Format

// 239 bytes
struct KissSimToSolidWorksPacket
{
    unsigned char KissSimTerminate;
    KissSimToSolidWorksMotor Motors[14];
    // 0-3 is CBC motors, 4-7 is CBC servos,
    // 8-10 is locomotion, 11-13 is Create LSD's
}

// 17 bytes
struct KissSimToSolidWorksMotor
{
    double MotorVelocity;
    double MotorGoal;
    unsigned char MotorHasGoal; }
```

## 6 Packet Format: Sensors

The sensor packet format consists of analogs and digitals for the various CBC and Create inputs. It also includes the result for track_x(0,0); in the future multiple blobs and models will be supported. Locomotion data is also sent, as is a Terminate flag similar to the one send with motors.

```
// SolidWorks to KISS-Sim Packet Format

// 95 bytes
struct SolidWorksToKissSimPacket
{
    unsigned char SolidWorksTerminate;
    short Analogs[14]; // 0-7 is CBC analogs; 8 is Create analog; 9-12 is
Create cliffs, 13 is Create wall
    long Digitals[14]; // 0-7 is CBC digitals; 8-11 is Create digitals; 12-13
is Create bumps
    short CameraX;
    double XDelta;
    double YDelta;
    double ThetaDelta;
}
```

## 7 Identifying Assembly Components

We match SolidWorks model components to simulated motors/sensors using a recursive search

through the assembly to find components based on keyword in their name. Using recursive search allows subassemblies to be used freely. The following names are currently matched:

- pvc
    - Any component with this keyword can trigger a touch sensor. To add other keywords, modify the TraverseComponent Sub in the VBA macro. (Note that adding touch sensor triggers will slow down the simulation accordingly.)
- RTBot[X]
    - A robot subassembly.
    - All sensors must be members of a robot assembly to be recognized.
    - [X] is the number of the robot (currently must be 1).
- Leversensor[X], lgtouch[X], smtouch[X]
    - The various touch sensors from the Botball kit.
    - Sensor is plugged into port [X].
    - (These are still being worked on as of this writing, but should be fully functional by the time of GCER.)
- create
    - An iRobot Create chassis.
    - The Create's front bumper is automatically assigned as a touch sensor.
    - A robot can have only one Create (but we can't imagine why this would be a serious limitation).
- ETCollide[X]
    - Collision definition for an ET rangefinder.
    - ET is plugged into port [X].
    - All analog ports can be used simultaneously.
- CameraCollide[X]
    - Collision definition for a camera.
    - [X] is the camera number (range 0 to 2)
    - All three cameras can be used simultaneously.
- RTBot1[M]
    - A motor, where [M] is one of the following:
        - M0
        - M1
        - M2
        - M3
        - S0
        - S1
        - S2
        - S3
        - X
        - Y
        - Theta
        - LSD0
        - LSD1
        - LSD2

## 8 Implementation: DC Motors

Once DC motor data is received by SolidWorks, it is run through a trajectory generator. The generator iterates through times, using the slope-intercept form of a line to calculate positions (velocity is slope; initial position is y-intercept). If the position crosses the goal position while the "Has Goal" flag is enabled, the velocity is set to 0 until further motor commands dictate otherwise, and the position is frozen at the goal position. The end result is a table of times and motor positions.

We wanted to directly access the motor properties from VBA to import this table, but there was insufficient documentation for this API feature. Specifically, the API documentation claims that it accepts an array of doubles, but does not specify the format. We tried to reverse-engineer the format by trial and error, but had no luck.

So, we improvised. We still had the ImperiSim 1.0 CSV import functions, so we simply wrote VBA code to output a CSV file for each motor based on the trajectories, and then imported the file using ImperiSim 1.0. This worked, but there were performance issues, because now hard drive access was slowing us down.

Our solution was a little-known Windows filesystem feature, the "Temporary" flag. Temporary files show up as normal files in the filesystem, but the OS caches them in RAM and never writes them to the hard drive unless RAM runs low. Temporary files get wiped when the system reboots, but this isn't a problem for our uses. Writing code in VBA to access the Temporary flag was a bit of an adventure, since VBA doesn't normally expose this; we had to use the Windows API DLL's instead of the usual VBA file-writing commands.

We had some additional adventures due to Windows Vista/7's Permissions feature; we were initially developing this on Windows XP, where writing Temporary files to C:\ is acceptable. Vista/7 frowns on this practice, silently wiping the files, which caused us major debugging headaches as we tried to find out why our files weren't showing up on our newer computers (which ran Windows 7). Eventually we figured out the issue, and decided to write the files to the Public Documents folder, which appears to always be writable.

## 9 Implementation: Servo Motors

Servo motors are similar to DC motors, but KISS-C only sees a goal position, not a velocity or actual position. We handle servos as DC motors, but we hardcode a certain velocity so that the servos attempt to reach the requested goal. In the future, we'd like to incorporate a method for the user to specify the velocity for individual servos, which would produce more realistic simulations given unusual servo loads.

## 10 Implementation: Locomotion Motors

We initially treated locomotion motors like any other motor, with the exception that the KISS-C functions would perform kinematic conversions before sending the velocities and positions to SolidWorks. As it turned out, this was a bad idea, because it meant that the trajectory generator

was running after the kinematics.  We're currently in the process of redoing this, so that kinematics are run after every trajectory iteration on the SolidWorks side, and KISS-Sim simply sends the usual motor commands without regard to kinematics.

## 11 End of Part 1

That's it for Part 1.  Part 2 will continue where we leave off here, covering sensor implementation and providing an operational guide.  See you there!