CS 524 Task 3: Datacentric What If Analysis

Description

The goal of this task is to build a small QMSVA framework to make quantitative *what if* decisions about the best configurations of a data element for a hypothetical purpose, such as Task 2. As described in Lecture 21, the element is a representation of an unsigned real number that is perfect (i.e., no rounding errors, etc.) within its defined limits. You will build this data element and determine these limits experimentally. Your approach will use fully automated test generation to support sensitivity analysis by comparing the results of your solution to those of Java's `float` datatype.

Specifications

Model 1: Data Element

Create a Java class called `Real` as follows:

- The constructor takes (1) two integers, which define the number of bits for the value and shift parts of the numerical representation, and (2) the `float` value it encodes.
- Method `add` takes a `Real` and returns as a new `Real` the result of adding it to itself. Both arguments must have the same configuration.
- Method `subtract` takes a `Real` and returns as a new `Real` the result of subtracting it from itself. Both arguments must have the same configuration.
- Method `multiply` takes a `Real` and returns as a new `Real` the result of multiplying itself by it. Both arguments must have the same configuration.
- Method `divide` takes a `Real` and returns as a new `Real` the result of dividing itself by it. Both arguments must have the same configuration.
- Method `interpolate` takes two `Real`s (the other end and a percent) and returns as a new `Real` the result of performing linear interpolation. Both ends of the line must have the same configuration.
- Method `compare` takes a `Real` and returns an integer `-1`, `0`, or `+1` based on comparing it to itself, where these values correspond to the argument being greater than, equal to, and less than the object, respectively. Both arguments must have the same configuration.
- Method `getValue` returns the `float` value of the encoded value.

The implementation of your encoding representation is your choice, but it needs to correspond to the following definition:

- The number of bits for the value defines the maximum size of the decimal integer value.
- The number of bits for the shift defines the maximum number of base-10 digits the decimal point may shift from the right side of the value.

  For example, *pi* could be encoded as the decimal 314159 with a shift of 5. This representation would require 19 value bits and 3 shift bits. The maximum decimal value that 19 value bits can encode is 524,287 (2^{19}–1) from $\lceil \log_2(314159) \rceil$. The maximum number of shifts 3 shift bits can encode is 7 (2^3–1) from $\lceil \log_2(7) \rceil$. Therefore, a configuration of (19,3) allows for real values between 0 and 524,287. However, not all real values are available on this interval, which is why our "perfect" representation will yield imperfect results in some cases.

  Here are some other possible encodings, where only the shift differs:

Shift	Value
0	314159.
1	31415.9
2	3141.59
3	314.159
4	31.4159
5	3.14159
6	.314159
7	.0314159

The maximum number of bits (value plus shift) is 32. Your simulation will try all combinations of configurations except (0,0), which can hold no data at all.

The sign does not consume any bits. In a true compressed version mapped onto hardware-level bits, it would, but since you have the freedom to implement your representation however you want, we can pretend this aspect away.

Clamp the value to encode if it exceeds the limits. For example, 3 value bits and no shift bits can represent at most 7 (from binary 111). Trying to insert a larger number clamps to this limit.

Model 2: Random Walk

Create a Java class called `RandomWalk` as follows:

- The constructor takes three integers, which defines the number of steps to take, the number of value bits, and the number of shift bits, respectively.
- Method `execute` randomly walks the length of the path as follows:

  Start at position (0,0,0). The coordinate system is the standard mathematical one, but this information is irrelevant because we are interested in the total error over the path length, not specifically where we end up. We are not visualizing this virtual movement.

  Iterate over the path length such that at every step, you generate random signed x, y, and z offsets from the current position, which then becomes the new position. Use Java's `nextFloat()` method uniformly distributed over `[-1000000,+1000000]`.

  Maintain two simultaneous paths. The first uses Java's `float` datatype for the (x,y,z) position, whereas the second uses your `Real`. Use the same three random step triplet for each. At the end of execution, return the error between the final positions as a `float` Euclidean distance.

Simulation

Create a class called `RandomWalkSimulation` as follows:

- The constructor takes two integers, which define the number of iterations to execute and the random seed value.
- Method `simulate` executes the iterations on independent instances of `RandomWalk` and keeps track of the errors. At the end, report the average, standard deviation, minimum, and maximum of the errors.

Visualization

Generate informative, meaningful plots of your own choice to show the average and standard deviation for all combinations of value and shift bits at path lengths of 1, 10, and 100. The provided class `Generate` lists all 146 bit combinations. You may modify it however you want. The number of iterations should be 1,000. We will cover the visualization more in lecture.

Make sure you consider the value and shift bits separately. In other words, do not combine them in a way that convolutes or obscures the landscape that we are trying to produce. One factor is probably going to dominate.

Analysis

TBD: Present and discuss the results. The contents of this section will be addressed in more detail in lecture. We need to test the math functions independently and collectively.

Without running this experiment, do you think the results would differ if instead of three dimensions, we used two or one or more than three? Address this in general for both fewer and more dimensions.

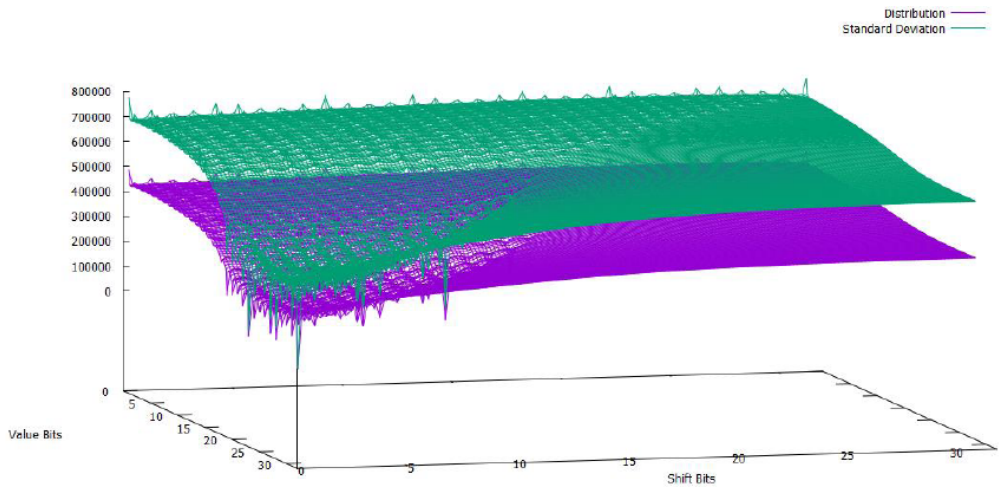
Deliverables

Submit your source code and a PDF report with your visualization and analysis.

Resources

The Gnuplot installation will vary depending on your platform. See `http://www.gnuplot.info/`

The best representation of the results captures relationships between the number of value and shift bits. Make a three-dimensional graph with each on a horizontal axis. On the vertical axis, include both the average and standard deviation of the error. For example:



This landscape has a clear region where the error is lower. Note that this example does not exactly correspond to the specifications above. We never have this many shift bits because the generator eliminates unnecessary combinations. Still, the bottom right supports this argument that they are unnecessary because there is no real change.

Repeat this type of graph for path lengths of 1, 10, and 100.

The Gnuplot commands are you choice, but this example was generated with a variant of this one. Be careful with copy and paste this because this contained some evil hidden character that Gnuplot hated when I first did it. You should not get an `invalid command` error.

```
reset
set title 'Path Length 1'
set xlabel 'Shift Bits'
set ylabel 'Value Bits'
set zlabel 'Error'
set grid

set hidden3d
set dgrid3d 32,32 qnorm 1

splot '-' with points pointtype 5 pointsize 1 palette linewidth 30 title 'Average Error'

# error-avg x,y->z
1 1 3.0
1 2 3.5
1 3 3.1

2 1 3.9
2 2 3.7
2 3 3.5

3 1 2.4
3 2 2.3
3 3 2.2
EOF
reset
set title 'Path Length 1'
set xlabel 'Shift Bits'
set ylabel 'Value Bits'
```

```
set zlabel 'Error'
splot '-' title 'Average Error' with lines, '-' title 'Standard Deviation' with lines

# error-avg x,y->z
1 2 3
2 7 5
5 3 2
9 8 2
8 0 9
EOF

# error-std x,y->z
9 7 1
2 5 2
9 7 7
2 4 6
0 2 5
EOF
```

To execute the script: `load 'myscript.gnu'`

To set the perspective, use the mouse or commands:

top:	`set view 0,0`
side:	`set view 90,0`
front:	`set view 90,90`
orthogonal:	`set view 45,45`