Bibliographic Citation Data:
Swan, M. S. and R. M. Brannon (2009). Statistical perturbation of material properties in Uintah. The University of Utah Undergraduate Research Abstracts.

STATISTICAL PERTURBATION OF MATERIAL PROPERTIES IN UINTAH

M. Scot Swan (Rebecca M. Brannon)
Department of Mechanical Engineering
University of Utah, Salt Lake City, UT 84112

May 2009

Abstract

Current simulations of material deformation are a balance between computational effort and accuracy of the simulation. To increase the accuracy of the simulated material response, the simulation becomes more computationally intensive with finer meshes and shorter timesteps, increasing the time and resource requirements needed to perform the simulation. One method for improving predictions of brittle failure while minimizing computational overhead is to implement statistical variability for the material properties being simulated. This method has low computational overhead and requires a relatively small increase in resource requirements while significantly increasing the precision of simulation results. Currently, most simulation frameworks inaccurately describe brittle and heterogeneous materials as uniform bodies of equal strength and consistency. This over-simplification underscores the need to implement statistical variability to help better predict material response and failure modes for materials that contain intermittent abnormalities such as changes in hardness, strength, and grain size throughout the specimen. Uintah, the computational framework developed by the University of Utah's C-SAFE program, has a simplistic native Gaussian distribution function that was hard-coded into select material models. The goal of this research is to create an easily duplicable method for enabling dynamic global variability according to a Weibull distribution in constitutive models in Uintah and to implement said ability into the constitutive model Kayenta. The main application of Kayenta is to simulate geological response to penetration and perforation. For the purpose of simulating failure in brittle geological samples, the Weibull distribution produces realistic statistical scatter in constituent properties that correlates well to flaws and irregularities observed in laboratory tests.

1. Introduction – purpose to implement same kind of variability only in a much more complex model.

This paper outlines the purpose and the implementation of statistical perturbation into material model wrappers in Uintah. This paper is also meant to be a reference to be used to reproduce the work that was performed by M. Scot Swan during the Spring 2009 semester at the University of Utah in the Computational Solid Mechanics lab. Specifically, it is an outline of the theory behind and the steps required to statistically perturb material properties in constitutive model wrappers in Uintah.

One of the major shortcomings in simulating deformation of brittle or heterogeneous materials is that the material cannot be described as a uniform body of equal strength and consistency. The ability to perturb material properties was implemented to help simulate more accurately the failure modes for materials that contain intermittent abnormalities such as changes in hardness, strength, and grain size throughout the specimen. Concrete is a good example of a heterogeneous material that cannot be adequately simulated at engineering scales without variability in strength caused by particles and flaws that exist at a scale too small to model explicitly. Also, statistical heterogeneity leads to scale effects, meaning large samples are weaker, on average, than smaller samples.

Modeling brittle materials is a great challenge to the computational physics community. Even with increasingly faster computers and more advanced codes, simulating the failure of brittle materials is still highly elusive. As can be seen in the top row of Figure 1.1, conventional damage models are horribly inaccurate when simulating brittle failure, especially when modeled with a coarse mesh. When aleatory uncertainty is introduced to that same model the results are much more realistic even when modeled with a coarse mesh. Even though the simulations become more accurate with this addition, it does not perfectly model brittle materials. However, it does make the results more realistic without a significant increase in computational resources.

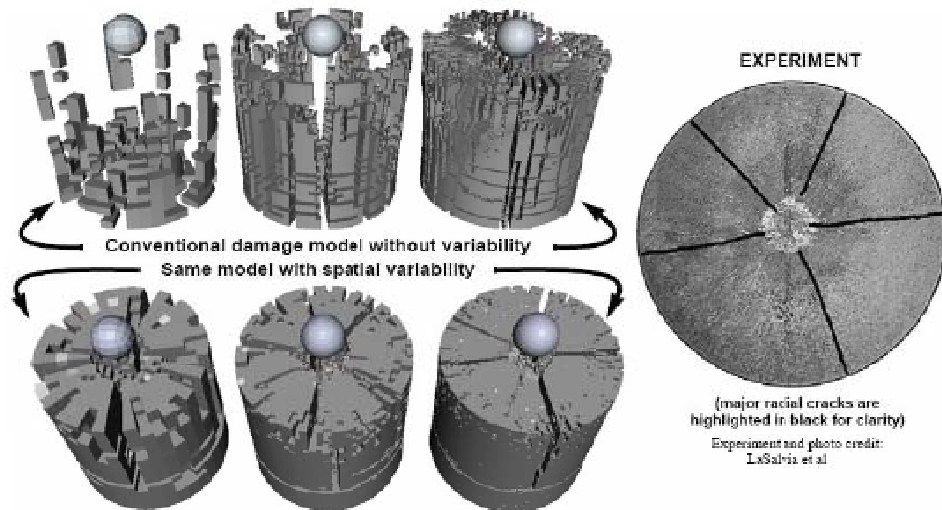

Figure 1.1. This figure (reproduced with permission (1)) shows the shortcomings of conventional simulation models by comparison of results at varying mesh refinements. Without variability and scale effects, the material fails 'everywhere' as shown in the top row. Variability and scale effects allow realistic formation of radial cracks, as shown in the bottom row and compared with the experiment at the right.

Procedures for verifying the mathematical accuracy of a damage model and validating the results against data are not yet well established (2). Because of the highly nonlinear and unstable nature of brittle damage, conventional models using Hooke's law are very inaccurate for these situations. Currently standard modeling procedures that use Eulerian codes introduce error at every mesh remap at every timestep. When many timesteps are needed, the error accumulates and makes the output differ greatly from the actual failure that would be observed. Consequently, a good model could easily yield poor results not because the equations being solved are unrealistic, but because the host code introduces errors in the solutions itself (3). Part of this problem can be solved by using the material point method (MPM) where the host code uses a grid as a computational scratchpad for computing spatial gradients of field variables. The grid is convected with the particles during deformation at every timestep, eliminating the diffusion problems associated with advection on an Eulerian grid. Due to the fact that the code does not perform a mesh remap at the end of every timestep, MPM is a good choice for large deformation problems, particularly those involving complex geometries and contact where typical finite element type methods (Eulerian codes) frequently fail (4).

Uintah, a computational framework developed by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE), was written as an implementation of MPM in order to better simulate complex systems that incorporate chemistry, fluid dynamics, thermodynamics, and solid mechanics. The purpose of this research was to create an easily duplicable method for dynamically perturbing parameters in Uintah constitutive model wrappers.

It has been found that the Weibull distribution can accurately describe a body that cannot be modeled as an isotropic solid due to intermittent flaws and variable strength in the specimen (5). The equation used to generate Weibull realizations is:

$$X = \tilde{X}\left[\frac{\ln R}{\frac{V}{\tilde{V}}\ln(1/2)}\right]^{1/m} \qquad (1)$$

Where X is the realization value, $\tilde{X}$ is the median value for the parameter as found in laboratory tests, *R* is a random number [0,1), *V* is the volume of the element or particle that is being modeled, $\tilde{V}$ is the reference volume used to determine $\tilde{X}$, and m is the Weibull modulus (6). The Weibull modulus is frequently referred to as the 'shape parameter' and that $\frac{V}{\tilde{V}}$ is frequently referred to as the 'scale parameter' and denoted by the letter α.

One of the main benefits of using a Weibull distribution is that it incorporates scale effects into the realization. This ability helps to justify using laboratory data that was collected using a certain size of specimen when each particle being simulated is much smaller than the laboratory specimen. The general concept of scale effects was incorrectly postulated by Leonardo da Vinci in the 16th century. A century later, Edme Mariotte correctly determined the source of scale effects in rope. He found that "a long rope and a short one always support the same weight unless that in a long rope there may happen to be some faulty place in which it will break sooner than in a shorter". The idea of global failure being a product of initial localized failure (as opposed to uniform global failure as seen in Figure 1.1) is key to the rationale for using a Weibull distribution to generate perturbed property values for simulation.

2. Implementation

Initially, there were three phases to introduce dynamic global variability into a material model in Uintah:

1. Create a native Weibull realization generator that can be called by any material model wrapper.
2. Define in the file "constitutive_models.xml" the XML tags that will be used to pass in the required data from the input .ups file.
3. Edit the wrapper code for model in which variability is desired.
   a. Copy the attached Weibull parser code (found in appendix A) into the model wrapper .cc file if the perturbed material parameters will be passed from the .ups file in one XML tag.
   b. In the header of the wrapper.cc file, add "#include <Core/Math/Weibull.h>" so that the Weibull realization generator can be called from inside the wrapper.
   c. Create new Labels (ParameterLabel and ParameterLabel_preReloc) for each perturbed parameter.
   d. Create and 'register' the new particle field array with the DataWarehouse.
   e. Initialize the Weibull realization generator and populate the particle field array for each parameter for which variability is desired.
   f. 'Hijack' the constant parameter that is to be perturbed before it is used. The primary focus being the function ComputeStressTensor().

1) Weibull Realization Generator

The function to produce Weibull realizations was patterned after the Gaussian realization generator. Both of these functions are found in "/installdir/SCIRun/src/Core/Math/". The Weibull realization generator requires several inputs to initialize: the median value, the Weibull modulus, the reference (laboratory test) volume, and the seed for the random number generator. These parameters are required just to initialize the realization generator. When actual realizations are desired, the .rand() function of the Weibull class is called. The .rand() function requires the volume of the particle for which the realization is being generated for. The generator uses equation 1 from the introduction to generate realizations. Also in Weibull.h is a function to determine the probability that a certain value was taken from a certain distribution. The equation is (7):

$$P(x) = \frac{m}{\alpha}\left(\frac{x-\tilde{X}}{\alpha}\right)^{m-1} e^{-\left(\frac{x-\tilde{X}}{\alpha}\right)^{m}}$$

This function is not used, but was present in the Gaussian realization generator, so it was included in the Weibull realization generator as well.

Example 2.1 – taken directly from Kayenta.cc

```
// Initialize Weibull generator
SCIRun::Weibull weibGen(Med, Mod, RefVol, Seed);

// make particle volume array available
constParticleVariable<double> pVolume;
new_dw->get(pVolume, lb->pVolumeLabel, pset);

// allocate storage for the perturbed array in the DataWarehouse
new_dw->allocateAndPut(partArray,partArrayLabel,pset);

// loop through for each particle and create realizations
ParticleSubset::iterator iter = pset->being();
For(;iter != pset->end(); iter++){
  partArray[*iter] = weibGen.rand(pVolume[*iter]);
}
```

2) Define XML Tags

Currently, as of May 2009, Uintah requires an entry in the file "constitutive_models.xml" in order to retrieve user input from the .ups file. This file can be located in the "/installdir/SCIRun/src/StandAlone/inputs/UPS_SPEC/". This file is not compiled when Uintah's makefile is run. Uintah reads this file from the source directory every time that a simulation is run. In fact, the "/installdir/SCIRun/opt/StandAlone/inputs/" directory doesn't even exist. The file is full of specifications for every tag in every constitutive model that is available in Uintah. Editing this file is very straightforward and can easily be deduced by seeing the hundreds of examples in the file. For the purposes of Kayenta, duplicate XML tags were produced for each of the parameters with the new 'Weibull perturbed' inputs having a 'W' before the standard tag (ie <B0></B0> for constant, <WB0></WB0> for Weibull perturbed). Here is an example line from the file:

```
<WPEAKI1I   spec="OPTIONAL STRING">
```

It is important to note that if the Weibull parser is being used then the specification must be set to "OPTIONAL STRING". Otherwise, the specification for each individual parameter is "OPTIONAL DOUBLE".

The function for parsing the .ups file is called by "ps->get(tag,variable)". This .get() function is the reason why the aforementioned constitutive_models.xml file exists. The function ps->get can only be called on tags that are defined in that file. For example:

```
ps->get("WPEAKI1I",  wPeakI1I.WeibDist);
```

The above example will parse the .ups file and put the string (see the following section as to why) that it finds in the "WPEAKI1I" tag into the variable "wPeakI1I.WeibDist".

3)a) Wrapper: add the Weibull parser code, if desired

During development of the ability to add variability, it was expressed that it would be convenient to be able to set variability in the model by using the string of parameters in the XML tag instead of needing a new tag for each of the values required for Weibull perturbation of each parameter. In response, a parser was created to accept strings that contain all of the Weibull parameters. To use the WeibullParser() function, all that is needed is to add the code (a copy of which is found in appendix A) in the wrapper's .cc file and the following line into the wrapper's .h file:

```
Virtual void WeibullParser(WeibParameters &iP);
```

Where the data type WeibParameters defined in the .h file as a structure of:

```
Struct WeibParameters {
        bool Perturb;           // 'True' for perturbed parameter
        double WeibMed;         // Median of distribution OR constant value depending on Perturb
        double WeibMod;         // Weibull modulus
        double WeibRefVol;      // Reference volume (volume of laboratory test specimen)
        int WeibSeed;           // Seed value for the random number generator
        std::string WeibDist;   // Variable to hold the value retrieved from ps->get(…)
                                        // this string populates the rest of the structure
};
```

The use of the structure as written above (assuming no alterations to the WeibullParser() function) is required in order to make the parser work. The parser is written in such a way that if a single numeric value is found in the string it will just save that value to the var.WeibMed variable and keep the "Perturb" flag set to 'False'.

If perturbation is desired then the only thing that must be changed is the content inside the XML tag of the .ups file. The trigger for a Weibull distribution is for the first 4 alphanumeric values to be 'weib' (case insensitive). The parser is incredibly forgiving when dealing with whitespace and stray non-alphanumeric values that might have found their way into the string. The rest of the string must be formatted in the following way:

```
            |--Distribution--|-Median-|-Modulus-|-Reference Vol -|- Seed -|

<WPEAKI1I>      weibull,         45e6,         4,         0.0001,         0      </WPEAKI1I>
```

The following would be equally acceptable and would yield the same values:

```
<WPEAKI1I>Weibull,45e6,4,0.0001,0</WPEAKI1I>
<WPEAKI1I>wEIbSomeReallyLongString,45.0   e \6,   4.0000000,  1e-4?#@!*,  0              </WPEAKI1I>
```

Now an example of the usage of the parser in the actual .cc file:

```
WeibParameters wPeakI1I;
ps->get("WPEAKI1I",wPeakI1I.WeibDist);
WeibullParser(wPeakI1I);
```

This will create the structure, fetch the string from the .ups file, and populate the structure by the contents of the string from the .ups file.

3)b) Wrapper: header – Self explanatory.

3)c) Wrapper: Label creation

Labels need to be created in order for Uintah to correctly hand off particle data as it is transferred from patch to patch. The label creation is usually handled in the function initializeLocalMPMLabels(). The following is an example of the syntax:

```
parameterLabel = VarLabel::create("wpeakI1I",
                        ParticleVariable<double>::getTypeDescription());
parameterLabel_preReloc = VarLabel::create("wpeakI1I+",
                        ParticleVariable<double>::getTypeDescription());
```

For large lists of particle variables that might be perturbed, it might be more efficient to create a vector of strings so that after initializing the strings with the desired labels the actual creation of the labels can be handled in a single for loop (for an example see Kayenta.cc).

3)d/e) Wrapper: Create and register particle arrays / populate particle arrays

After the .ups file has been parsed, the perturbed values have been extracted out of the strings, and the labels have been created for the DataWarehouse to track particles between patches, it is now ready to initialize the particle arrays to hold perturbed values. The particle arrays are initialized in the function initializeCMData(). The code in this function must now determine which variables will require a particle array for perturbed values. The first two lines of the following example initialize the particle arrays and the remainder of the lines initialize the Weibull realization generator and in the final for loop the particle arrays are populated.

```
If (wPeakI1I.Perturb) {
    // create and register the particle variables
    ParticleVariable<double> wpeakI1I;
    new_dw->allocateAndPut(wPeakI1I, parameterLabel, pset);

    //initialize Weibull realization generator
    SCIRun::Weibull weibGen(wPeakI1I.WeibMed,wPeakI1I.Mod,
                        wPeakI1I.WeibRefVol,wPeakI1I.WeibSeed);

    //  Make particle volume available
    constParticleVariable<double> pVolume;
```

```
        new_dw->get(pVolume, lb->pVolumeLabel, pset);

        // Perform the initialization of each particle
        ParticleSubset::iterator iter = pset->begin();
        for(;iter != pset->end();iter++){
                wpeakI1I[*iter] = weibGen.rand(pVolume[*iter]);
        }
}
```

It can be seen that the coding can be simplified for many perturbed parameters by encapsulating this if statement into a for loop for each parameter that might be perturbed.

3)f) Wrapper: 'hijack' the property values before computation

This step was necessary for adding global variability into Kayenta, but should be handled on a case-by-case basis. For most material models that have their code actually in the wrapper (instead of a function call to an outside function) the code can easily be adapted for each variable that might be perturbed. One option, only viable for models that will have relatively few perturbed parameters, is to make arrays that will always be initialized every time the model is called. In this case, the function that populates the array will simply assign the same numeric value to every particle if no distribution is desired. If that is not an acceptable fix, then setting up a system of if statements around the instances where these perturbed parameters are used is the only acceptable method of initialization. The implementation of 'hijacking' parameters was very easily implemented in Kayenta due to the actual model being called from one line and the parameters being stored in an array. The following is the example from Kayenta.cc where the array of particle arrays is called WUI[] and the values in the UI[] array are being replaced right before the call to the material model:

```
// 'Hijack' the UI arrary with perturbed values if desired
// swap values for WUI[i][idx] with UI[i]
for (int i=0;i < d_NUI; i++){
        if (WUI[i].Perturb){
                double tmpvar = UI[i];
                UI[i] = WUI[i][idx];
                WUI[i][idx] = tmpvar;
        }
}

ISOTROPIC_GEOMATERIAL_CALC(nblk, d_NINSV, dt, UI, sigarg,
                                        Darray, svarg, USM);

// Return the values to their correct arrays
for (int i=0;i < d_NUI; i++){
        if (WUI[i].Perturb){
                double tmpvar = UI[idx];
                UI[i] = WUI[i][idx];
                WUI[i][idx] = tmpvar;
```

```
        }
}
```

As can be seen, when the material model is a function that is called from the wrapper and the parameters are contained in an array that mirrors the original variable layout, the 'hijacking' is incredibly simple. Even when it cannot be written in a way as short and simple as the above example, at the beginning of computeStressTensor() the if statements can be placed for each variable that might be perturbed, with the last thing in that function being the if statements that return the original values.

3. Verification of distribution

Verification of the Weibull generator is done by treating the realizations just like laboratory data. Verification was done using a python script and Uintah's built-in particle data extractor, Puda. After a simulation has run, Puda can be used to extract any data that is saved into the .uda files. Some of the input arguments for Puda are: the beginning timestep, ending timestep, material, particle variable, and to ignore extra cells. To verify the Weibull generator, a simulation is run that incorporates statistical perturbation and that saves the perturbed values to the .uda file. Puda is then run to extract the perturbed particle values and saves those values to a text file. That text file is passed to a Python script (attached in Appendix B) that performs the necessary calculations and outputs a summary of how well the data fits the expected analytical distribution with the same parameters.

When the data is extracted, it is sorted from the minimum to the maximum. Each value is assigned a 'probability of failure' with the minimum value being (nearly) zero and the maximum value being (nearly) 1. Below is an example for 20 failure stress Weibull realizations:

N	σ	Probability Failed (Pf)
1	0.3634	(1/20)*(1-1/2)
2	0.4826	(1/20)*(2-1/2)
3	0.6015	(1/20)*(3-1/2)
4	0.6477	(1/20)*(4-1/2)
…	…	…
20	1.4746	(1/20)*(20-1/2)

Figure 3.1. 20 realizations of failure stress according to a Weibull distribution with a median value of 1, a Weibull modulus of 4, and a scale parameter of 1. Probability failed is given by the equation $\frac{1}{T}\left(N-\frac{1}{2}\right)$ where T is the total number of realizations being analyzed.

The plot of failure stress vs probability failed for the 20 realizations looks like this

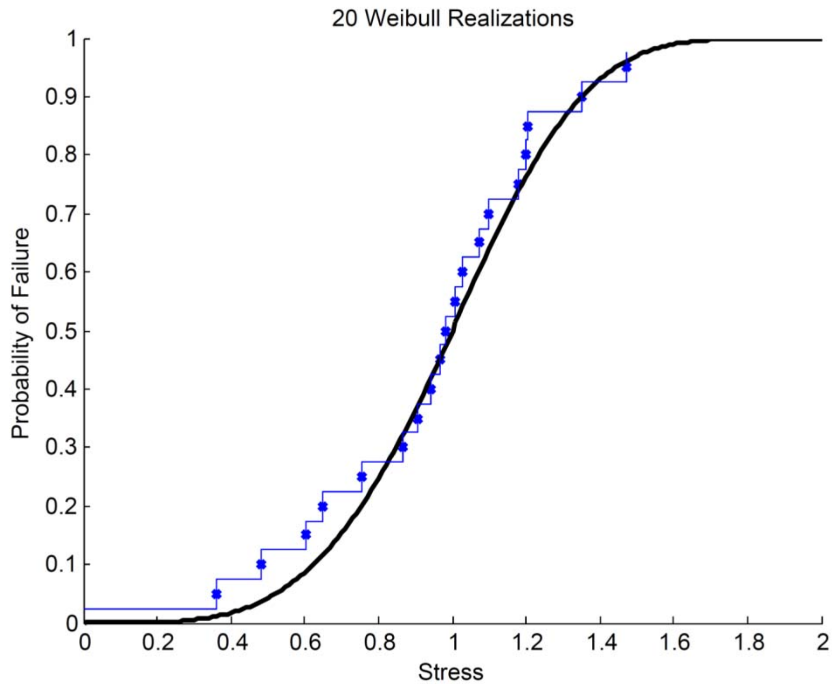

Figure 3.2. Plot of probability of failure vs. failure stress. The blue stair function is a plot of $\frac{N}{T}$. The blue dots are the values of the probability of failure. The black line is the expected analytical solution to the CDF of the probability. The equation for the analytical solution (3) is $P_f = 1 - 2^{-\alpha*\left(\frac{X}{\bar{X}}\right)^m}$.

The rough fit can be visually verified by the type of plot in Figure 3.2, but lacks mathematical rigor. By performing a transformation on the probability of failure and on the failure stress, the plot will turn into a linear function. A linear regression can check the fit and can give a mathematical value to the fit of the line. The slope of the resultant line is the Weibull modulus. Probability of failure is transformed with the following function to yield the new Y values:

$$y = \ln\left(\ln\frac{1}{1-P_f}\right) - \ln(\ln 2)$$

Failure stress is transformed to yield the new χ values:

$$\chi = \ln\frac{X}{\bar{X}}$$

The plot of χ vs. Y for our 20 realization example is:

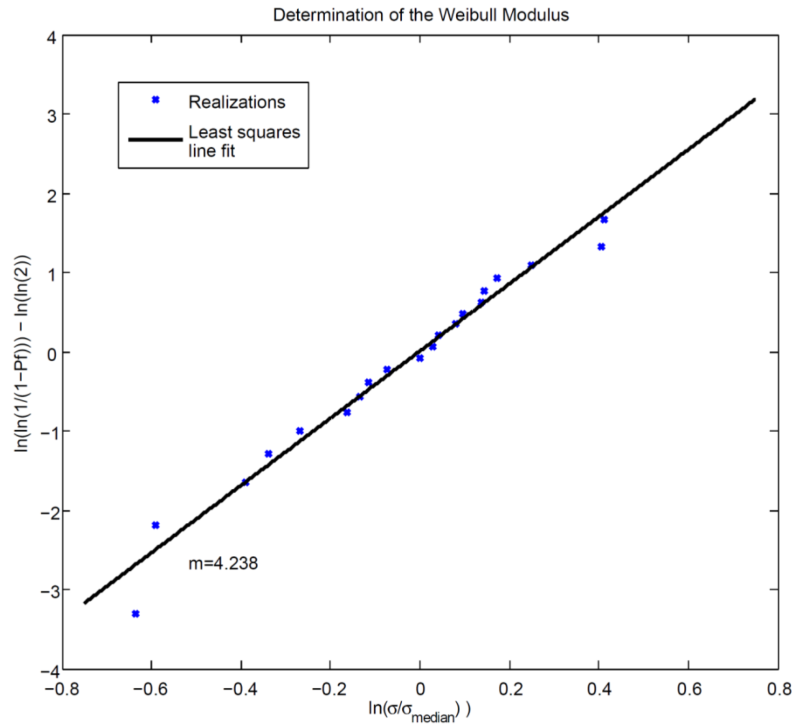

Figure 3.3. logarithmic plot of probability of failure vs failure stress. The blue dots are the realization values and the black line is the expected distribution given by the linear function $y = mx$.

Many simulations have been used to generate values from the Weibull realization generator and all have been verified and all have produced nearly perfect distributions. The output of the Python script analyzer is given for a simulation involving 6456 realizations with the input parameters:

Median value: 200 000 Weibull Modulus: 4 Scale Parameter: 1

START OUTPUT
sample size: 6456
median value: 200470
minimum value: 15550.6
maximum value: 370522
PF median: 0.500077

Weibull modulus: 4.0045
Intercept of modulus: -0.00743831
Error of line fit: 0.999685
END OUTPUT

As can be seen, the realization generator produced values that almost exactly fit the expected line with a slope of the Weibull modulus. The plot of the data used for this analysis is below:

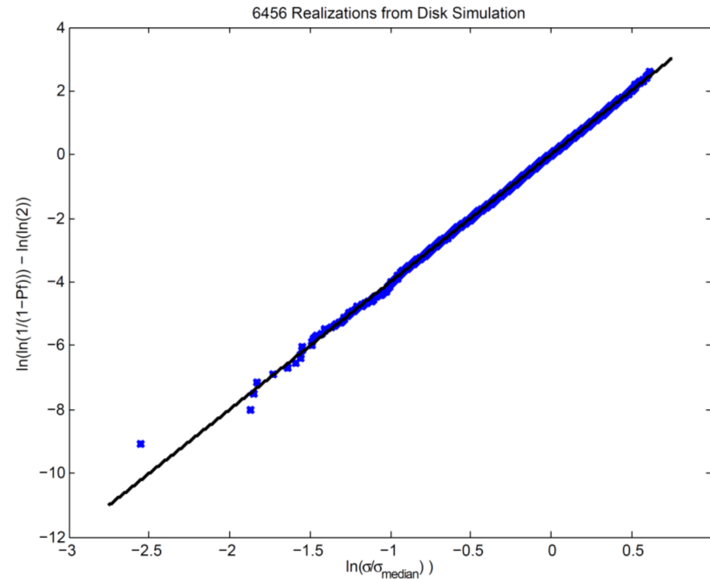

Figure 3.4. Plot of values generated in a simulation that incorporates statistical perturbation of failure stress.

4. Comments and Considerations

Specifically for the implementation of global variability in Kayenta, it is a very powerful tool, but there are few safeguards to keep the user from misusing the ability. The largest concern for the user when using this ability is that some variables are interdependent on other variables as they are passed and possibly recomputed by geochk. Because the code does not detect when a dependent variable is perturbed or not, it is the user's responsibility to perturb all interdependent variables with the same seed in order to get the desired effect. In the words of Erik Strack "We give the user a loaded gun and let them shoot themselves with it".

Even though the WeibullParser() is robust and has been adequately tested, it still might fail in such a way that the developer did not foresee. However, the parser is simple enough that it can be edited for specific purposes and specific input sets.

The seed of the Weibull realization generator is defaulted to zero, meaning that the 'random' numbers will always be given in the same order unless the seed is changed. The ability to pass the seed in as an input argument is useful to run batteries of tests with different realization sets of the same parameters. As mentioned above, it is the responsibility of the user to perturb and to seed interdependent variables together with the same seed (but not necessarily with the same Weibull parameters).

As of writing, May 2009, Kayenta has not been fully verified in Uintah. Due to this, it is difficult to ascertain whether or not the statistical variability is performing as expected. However, it has been verified to produce correct values and to run without crashing the simulation. If there are any problems, it is not in the coding, but in the method of implementation.

APPENDIX A

Weibull parser C++ code

```
// Author: M. Scot Swan
// Date: May 2009
// University of Utah
// Department of Mechanical Engineering
//
// Weibull input parser that accepts a structure of input
// parameters defined as:
//
// bool Perturb        'True' for perturbed parameter
// double WeibMed      Medain distrib. value OR const value
//                     depending on bool Perturb
// double WeibMod      Weibull modulus
// double WeibRefVol   Reference Volume
// int    WeibSeed     Seed for random number generator
// std::string WeibDist  String for Distribution
//
// the string 'WeibDist' accepts strings of the following form
// when a perturbed value is desired:
//
// --Distribution--|-Median-|-Modulus-|-Reference Vol -|- Seed -|
// "    weibull,      45e6,      4,       0.0001,        0"
//
// or simply a number if no perturbed value is desired.

void
Kayenta::WeibullParser(WeibParameters &iP)
{

 // Remove all unneeded characters
 // only remaining are alphanumeric '.' and ','
        for ( int i = iP.WeibDist.length()-1; i >= 0; i--)
        {

                iP.WeibDist[i] = tolower(iP.WeibDist[i]);

                if ( !isalnum(iP.WeibDist[i]) &&
     iP.WeibDist[i] != '.' &&
     iP.WeibDist[i] != ',' &&
     iP.WeibDist[i] != '-' &&
     iP.WeibDist[i] != EOF)
                {
                        iP.WeibDist.erase(i,1);
                }

        } // End for

 if (iP.WeibDist.substr(0,4) == "weib")
 {
  iP.Perturb = true;
 } else {
  iP.Perturb = false;
 }
```

```
// ######
// If perturbation is NOT desired
// ######
if ( !iP.Perturb )
{
  bool escape = false;
  int num_of_e = 0;
  int num_of_periods = 0;
  for ( unsigned int i = 0; i < iP.WeibDist.length(); i++)
  {
    if ( iP.WeibDist[i] != '.'
        && iP.WeibDist[i] != 'e'
        && iP.WeibDist[i] != '-'
        && !isdigit(iP.WeibDist[i]) ) escape = true;

    if ( iP.WeibDist[i] == 'e' ) num_of_e += 1;

    if ( iP.WeibDist[i] == '.' ) num_of_periods += 1;

    if ( num_of_e > 1 || num_of_periods > 1 || escape )
    {
      std::cerr << "\n\nERROR:\nInput value cannot be parsed. Please\n"
                "check your input values.\n" << std::endl;
      exit (1);
    }
  } // end for(int i = 0;....)

if ( escape ) exit (1);

iP.WeibMed  = atof(iP.WeibDist.c_str());
}


// ######
// If perturbation IS desired
// ######
if ( iP.Perturb )
{
                int weibValues[4];
  int weibValuesCounter = 0;

  for ( unsigned int r = 0; r < iP.WeibDist.length(); r++)
  {
    if ( iP.WeibDist[r] == ',' )
    {
      weibValues[weibValuesCounter] = r;
      weibValuesCounter += 1;
    } // end if(iP.WeibDist[r] == ',')
  } // end for(int r = 0; ...... )

  if (weibValuesCounter != 4)
  {
    std::cerr << "\n\nERROR:\nWeibull perturbed input string must contain\n"
              "exactly 4 commas. Verify that your input string is\n"
              "of the form 'weibull, 45e6, 4, 0.001, 1'.\n" << std::endl;
    exit (1);
```

```
    } // end if(weibValuesCounter != 4)

    std::string weibMedian;
    std::string weibModulus;
    std::string weibRefVol;
    std::string weibSeed;

    weibMedian  = iP.WeibDist.substr(weibValues[0]+1,weibValues[1]-weibValues[0]-1);
    weibModulus = iP.WeibDist.substr(weibValues[1]+1,weibValues[2]-weibValues[1]-1);
    weibRefVol  = iP.WeibDist.substr(weibValues[2]+1,weibValues[3]-weibValues[2]-1);
    weibSeed    = iP.WeibDist.substr(weibValues[3]+1);

    iP.WeibMed    = atof(weibMedian.c_str());
    iP.WeibMod    = atof(weibModulus.c_str());
    iP.WeibRefVol = atof(weibRefVol.c_str());
    iP.WeibSeed   = atoi(weibSeed.c_str());

  } // End if (iP.Perturb)

}
```

APPENDIX B

Weibull Verification Python Script

```
#!/usr/bin/python
from math import log
from math import floor
from math import sqrt
import sys

#
# author: M. Scot Swan
# Date:   May 2009
# Mechanical Engineering Deparment
# University of Utah
#
# usage:
# ./weibull.py pudaoutputfile.txt
#
def linreg(X, Y):
    """
    Summary
        Linear regression of y = ax + b
    Usage
        real, real, real = linreg(list, list)
    Returns coefficients to the regression line
                                    "y=ax+b" from x[] and y[], and R^2 Value
    """
    if len(X) != len(Y):  raise ValueError, 'unequal length'
    N = len(X)
    Sx = Sy = Sxx = Syy = Sxy = 0.0
    for x, y in map(None, X, Y):
        Sx = Sx + x
        Sy = Sy + y
        Sxx = Sxx + x*x
        Syy = Syy + y*y
        Sxy = Sxy + x*y
    det = Sxx * N - Sx * Sx
    a, b = (Sxy * N - Sy * Sx)/det, (Sxx * Sy - Sx * Sxy)/det
    meanerror = residual = 0.0
    for x, y in map(None, X, Y):
        meanerror = meanerror + (y - Sy/N)**2
        residual = residual + (y - a * x - b)**2
    RR = 1 - residual/meanerror
    ss = residual / (N-2)
    Var_a, Var_b = ss * N / det, ss * Sxx / det
    return a, b, RR


values = []
counter = 0

# Extract from the Puda output file needed data
for line in open(sys.argv[1]):
                a, b, c, d = line.split()
                values.append(float(d))
                counter = counter +1
```

```
minn = min(values)
maxx = max(values)
pf = []
capY = []
capX = []
values.sort()
median = values[int(floor(counter/2))]

# analyze the data (transform into a linear fit)
for j in range(1,counter):
        pf.append((1.0/counter)*(j-0.5))
        capY.append(log(log(1.0/(1.0-pf[j-1])))-log(log(2.0)))
        capX.append(log(values[j-1]/median))

PFmedian = pf[int(floor(counter/2))]

a, b, rr = linreg(capX,capY)

print "\nInput File:"
print sys.argv[1]

print "\n\n#### START OUTPUT ####\n"
print "sample size:         %d" % counter
print "median value:        %g" % median
print "minimum value:       %g" % min
print "maximum value:       %g" % max
print "PF median:           %g\n" % PFmedian
print "Weibull Modulus:     %g" % a
print "intercept of Modulus: %g" % b
print "Error of line fit:   %g" % rr
print "\n#### END OUTPUT ####\n"
```

Bibliography

1. *Validating theories for brittle damage.* **Brannon, R.M., J.M. Wells, et al.** 2007, Metallurgical and Materials Transactions, pp. 2861-2868.

2. *Proc. 25th Army Science Conf.* **B. Leavy, C. Krauthauser, J. Houskamp, and J. LaSalvia.** 2006.

3. **Strack, R.M. Brannon and O.E.** *unpublished research.* Albuquerque, NM : Sandia National Laboratories, 2005-2006.

4. **M. Steffen, P.C. Wallstedt, J.E. Guilkey, R.M. Kirby, M. Berzins.** Examination and Analysis of Implementation Choices within the Material Point method (MPM). *Computer Modeling in Engineering & Sciences.* 2008, Vol. 31, 2.

5. **Jayatilaka, A. De S.** Fracture of brittle materials in uniaxial compression. *Journal of Material Science.* 1978, 13.

6. **Strack, R.M. Brannon and O.E.** *Theory and Results for Incorporating Aleatory Uncertainty and Scale effects in Damage Models for Failure and Fragmentation.* Albuquerque, NM : Sandia National laboratories, 2006.

7. **ReliaSoft Corporation.** WeibullProbability Density Function. *Weibull.com.* [Online] 2006. http://www.weibull.com/LifeDataWeb/weibull_probability_density_function.htm.