Summary of Work on OpenSees Reliability/Optimization/Sensitivity

Michael H. Scott[1] and Kevin R. Mackie[2]

November 5, 2012

[1]School of Civil and Construction Engineering, Oregon State University, Corvallis, OR 97331
[2]Department of Civil and Environmental Engineering, University of Central Florida, Orlando, FL 32816

Contents

Abstract

As agreed upon by PEER researchers in mid-February 2010 at the Richmond Field Station, the original implementation of reliability modules within the OpenSees finite element software framework is unsustainable. This is due primarily to a rigid software design that made it difficult for developers to implement new or modify existing reliability, optimization, and sensitivity modules, and secondarily due to an inflexible format adopted for OpenSees/Tcl input commands in a reliability analysis. Small improvements in the software design and input commands have been made sporadically over the last few years; however, these changes, along with the addition of modules with a singular focus, have brought the reliability modules to a stand still. Now, wholesale design changes must be made in order to bring the reliability modules to a working state and to make the framework flexible for future extensions in probabilistic finite element analysis with OpenSees.

Item 1

Introduction and Sanity Check

1.1 Introduction

There exists a significant amount of information in the literature on how to perform reliability, sensitivity, and optimization analyses for a variety of problems. There are also a plethora of software tools to enable computation of numerical solutions for complex problems. However, there are still very few software platforms that are enabled with multiple functionalities. Specifically, it is of interest in performance-based earthquake engineering (PBEE) to perform nonlinear finite element analysis in the presence of uncertainties. Therefore, it is desirable to have a finite element analysis platform that has the ability to compute response sensitivities directly, solve reliability problems with specified uncertain quantities in the models, and solve optimization problems that consider response quantities in the objective.

The OpenSees framework is the only tool with these capabilities; however, it contains several legacy implementations that limit the ability to extend the framework to truly be an object-oriented code for future users and developers. The original finite element reliability analysis code was added by Haukaas and Der Kiureghian []. This effort was the first time past work on CalRel, FERUM, FEAP, etc. were combined with the new object-oriented framework in OpenSees. However, there were several major issues preventing the maintenance and extensibility of the code. They are summarized briefly here.

- Over-reliance on Tcl. The Tcl interpreter was used as a vessel for communication between all of the reliability modules, often through the use of `TclEval` commands that would compile even after implementations had changed. Additionally, the limit state functions, gradients, and several of the analysis routines were directly interwoven with Tcl commands that prevented any future changes to either the Tcl interpreter or a shift to a different interpreter (such as Python or Matlab).

- Code duplication and lack of class hierarchy. While the class structure was established initially following traditional object-oriented software patterns, a majority of the base classes were abstract with only a single concrete class. These concrete classes usually contained only single methods. The result was significant code duplication between

classes, primarily related to the need to parse limit state functions during evaluation and for the purposes of gradient computations. Additional efforts at extending the reliability modules (such as TELM, optimization modules, etc.) unfortunately duplicated the existing code, often resulting in two similar but conflicting implementations of the same base class or derived classes.

- Sequential "black box" analysis codes that prevent user interaction during and after analysis. The original code base largely followed a sequential script or function approach whereby the properties of the analysis were established, the analysis was run, and the output dumped to a file. This limitation is counter to the scripting and control that the user has over traditional OpenSees models and analyses.

- Occasional errors in code or compiler-specific compilation issues. There were several bugs in the code that prohibited only select options from running (did not effect the overall reliability modules), such as some of the random variable parameter definitions, SORM, etc. In addition, as changes were made to OpenSees, numerous portions of the reliability modules were broken due to some of the issues mentioned above.

1.2 Changes in Tcl Interpreter Commands and Core Classes

The following table provides a summary of the changes made to the class structure. Note that a majority of the original class hierarchy was retained. The majority of changes were made to the function evaluator classes, gradient classes, and performance function classes.

Table 1.1: Changes to reliability class structure.

Old Name	New Name	Notes
GFunEvaluator (BasicGFunEvaluator, OpenSeesGFunEvaluator, TclGFunEvaluator, MatlabGFunEvaluator)	FunctionEvaluator	See Chapter 4
GradGEvaluator (OpenSeesGradGEvaluator, FiniteDifferenceGradGEvaluator)	GradientEvaluator	See Chapter 5
ProbabilityTransformation (NatafProbability-Transformation)	ProbabilityTransformation	–
FindDesignPointAlgorithm (SearchWithStepSize-AndStepDirection)	FindDesignPointAlgorithm	–
SearchDirection (GradientProjectionSearchDirection, HLRFSearchDirection, PolakHeSearchDirectionAndMeritFunction, SQPSearchDirectionAnd-MeritFunction)	SearchDirection	See Chapter 9
StepSizeRule (FixedStepSizeRule, ArmijoStep-SizeRule)	StepSizeRule	–

MeritFunctionCheck (AdkZhangMeritFunctionCheck, CriteriaReductionMeritFunctionCheck, PolakHeSearchDirectionAndMeritFunction, SQPSearchDirectionAndMeritFunction)	MeritFunctionCheck	See Chapter 9
RandomNumberGenerator (CStdLibRandGenerator)	RandomNumberGenerator	–
ReliabilityConvergenceCheck (StandardReliabilityConvergenceCheck, OptimalityConditionReliabilityConvergenceCheck)	ReliabilityConvergenceCheck	–
RootFindingAlgorithm (ModifiedNewtonRootFindingAlgorithm, SecantRootFindingAlgorithm)	–	See Chapter 9
FindCurvaturesAlgorithm (FirstPrincipalCurvature)	FindCurvatures	See Chapter 6
RandomVariable (Normal, Lognormal, etc.)	RandomVariable	–
CorrelationCoefficient	CorrelationCoefficient	–
PerformanceFunction	PerformanceFunction	See Chapter 3
RandomVariablePositioner	–	Deprecated, see Chapter 2
ParameterPositioner	–	Deprecated, see Chapter 2
Filter (StandardOscillatorFilter)	Filter	–
ModulatingFunction (ConstantModulatingFunction, GammaModulatingFunction, TrapezoidalModulatingFunction)	ModulatingFunction	–
Spectrum (NarrowBandSpectrum, PointsSpectrum, JonswapSpectrum)	Spectrum	–
TimeSeries (DiscretizedRandomProcessSeries, SimulatedRandomProcessSeries)	TimeSeries	–

The following table provides a summary of the changes made to the Tcl interpreter commands. As with the class hierarchy, the changes to the interpreter commands are minimal to allow old input files to be used after making some key changes.

Table 1.2: Changes to reliability Tcl commands.

Old Command	New Command	Notes
randomVariable	randomVariable	Arguments have changed
rvReduction	–	Deprecated
correlate, correlateGroup, correlationStructure	correlate, correlatedGroup	–
randomVariablePositioner	parameter	No more positioners, see Chapter 2
parameterPositioner	parameter	No more positioners, see Chapter 2
DiscretizedRandomProcess, SimulatedRandomProcess	DiscretizedRandomProcess, SimulatedRandomProcess	–

modulatingFunction	modulatingFunction	–
filter	filter	–
spectrum	spectrum	–
performanceFunction	performanceFunction	See Chapter 3
probabilityTransformation	probabilityTransformation	–
gFunEvaluator	functionEvaluator	See Chapter 4
gradGEvaluator	gradientEvaluator	See Chapter 5
searchDirection	searchDirection	–
stepSizeRule	stepSizeRule	–
rootFinding	rootFinding	See Chapter 9
meritFunctionCheck	meritFunctionCheck	–
reliabilityConvergenceCheck	reliabilityConvergenceCheck	–
startPoint	startPoint	–
findDesignPoint	findDesignPoint	–
randomNumberGenerator	randomNumberGenerator	–
findCurvatures	findCurvatures	See Chapter 6

1.3 Linear-Elastic Frame Reliability Analysis

After significant time away from OpenSees development, my first task was to become familiar again with uncertainty modeling and sensitivity and reliability analysis in OpenSees. In the spring quarter of 2010 I taught CE 588, Probability-Based Analysis and Design, in which I provided students with a simple script for FORM analysis. The design point is found by the Hasofer-Lind algorithm with the Nataf transformation between **x**- and **u**-space. During the final week of the quarter, we discussed finite element reliability analysis, which was demonstrated by linking this MATLAB-based FORM analysis with a linear-elastic frame analysis program written by those students who took CE 585, Matrix Structural Analysis, in a previous fall quarter.

1.3.1 Model Definition

An in-class example I used is shown in Figure 1.1. All members have $E = 30000$ ksi. Column members 1 and 3 have $A = 29$ in^2 and $I = 2000$ in^4, while girder member 2 has $A = 25$ in^2 and $I = 1500$ in^4. To simulate uncertain response of the frame, the following random variables are mapped to the frame model parameters described therein:

X_1 – elastic modulus of both column members (elements 1 and 3)

$\mu = 30000$ ksi, $\sigma = 3000$ ksi, Lognormal PDF

X_2 – lateral load at node 2

$\mu = 25$ kip, $\sigma = 5$ kip, Normal PDF

X_3 – X-coordinate of node 1

$\mu = 0$ in (relative to origin), $\sigma = 1$ in, Normal PDF

X_4 – distributed gravity load on girder (element 2)

$\mu = 0.1$ kip/in, $\sigma = 0.02$ kip/in, Normal PDF

All random variables are uncorrelated.

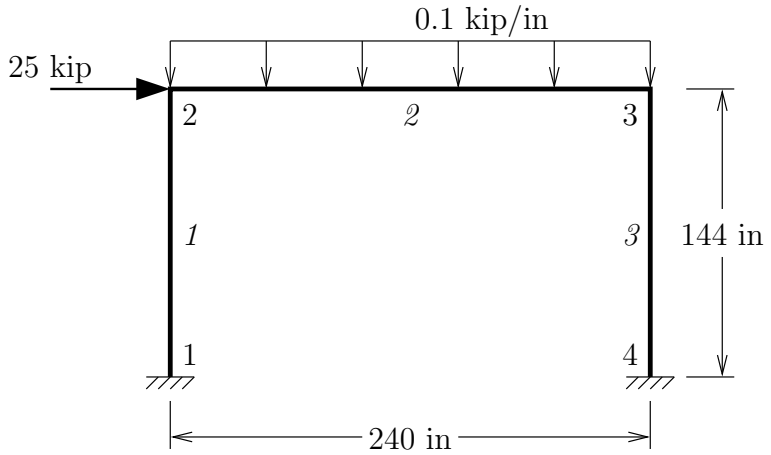

Figure 1.1: Portal frame example for finite element reliability analysis.

After a straightforward OpenSees model definition of the frame shown in Figure 1.1, the following Tcl script commands are issued in order to model uncertainty. First, parameters are created and mapped to the properties listed above, as shown in Figure 1.2. Note that non-sequential parameter tags are used to emphasize the departure from requiring all reliability objects be numbered with sequential tags starting at one.

```
parameter 12 element 1 E
addToParameter 12 element 3 E
parameter 25 loadPattern 1 loadAtNode 2 1
parameter  3 node 1 coord 1
parameter 45 loadPattern 1 elementLoad 2 wy
```

Figure 1.2: Creation of parameter objects in OpenSees/Tcl script.

Next, random variable objects are created using the mean values, standard deviations, and probability distributions previously listed. The relevant script commands are shown in Figure 1.3 with non-sequential tags assigned to the random variables. The Tcl variables `E`, `P`, and `w` are previously defined in the script with their respective mean values. Note that the uniform girder load, `w`, is defined with a negative value so that its direction of application is correct within the local coordinate system of element 2. In addition, a Tcl array, `param` is defined in order to establish to which parameter tag each random variable is associated.

The final step of the mapping between reliability and finite element domains is the creation of random variable positioners. As shown in Figure 1.4, the creation of random variable

```
randomVariable 62 lognormal $E [expr 0.1*$E]; set param(62) 12
randomVariable 32 normal $P [expr 0.2*$P]; set param(32) 25
randomVariable 89 normal 0 1; set param(89) 3
randomVariable 41 normal $w [expr abs(0.2*$w)]; set param(41) 45
```

Figure 1.3: Creation of random variable objects in OpenSees/Tcl script.

position objects is accomplished via a Tcl `foreach` loop over the names of the `param` array created in Figure 1.3. The tag assigned to each random variable position is inconsequential, so the loop induction variable `tag` is used for this purpose.

```
foreach tag [array names param] {
    randomVariablePositioner $tag -rvNum $tag -parameter $param($tag)
}
```

Figure 1.4: Positioning of random variables in FE domain.

The looping structure shown in Figure 1.4 may seem awkward; however, it is necessary to maintain the theme of allowing non-sequential tag assignments to objects of the reliability domain.

1.3.2 Reliability Analysis Results

Two performance functions are defined for this example. The first places a 0.15 in limit on the lateral displacement of node 2, while the second places a 1500 kip-in limit on the moment reaction at node 1. In OpenSees, the following reliability commands are issued

```
performanceFunction 76 ‘‘0.15-\[nodeDisp 2 1\]’’
performanceFunction 23 ‘‘1500.0+\[sectionForce 1 1 2\]’’
```

A positive sign is used in the second performance function to account for the bending moment sign convention within the local coordinate system of the element (each member is represented by a force-based beam-column element with three-point Lobatto integration and elastic sections). Note that the `nodeDisp` and `sectionForce` commands are used in the performance functions, which is a departure from the original performance function syntax of `u_2_1` and `rec_element_1_section_1_force_2` which required additional, unnecessary parsing and recorder creation within the reliability core of OpenSees. The use of commands such as `nodeDisp` and `sectionForce` lets Tcl do the parsing of performance functions (rather than developers) and allows far more flexibility in performance function definitions. This issue will be addressed extensively in subsequent items of this report. Again, non-sequential tags are used in defining the performance functions.

The following reliability analysis options were declared.

```
randomNumberGenerator         CStdLib
probabilityTransformation     Nataf            -print 3
reliabilityConvergenceCheck   Standard         -e1 1.0e-2    -e2 1.0e-2  -print 1
gFunEvaluator                 OpenSees         -analyze 1
gradGEvaluator                FiniteDifference
searchDirection               iHLRF
meritFunctionCheck            AdkZhang         -multi 2.0    -add 50    -factor 0.5
stepSizeRule                  Armijo           -maxNum 10   -base 0.5   -initial 0.3 5
startPoint                    Mean
findDesignPoint               StepSearch       -maxNumIter 30
```

Note in particular that the finite difference option is employed to find the gradient of the g-function. In subsequent report items, the direct differentiation method will be used to this end.

The first order reliability analyses converged to the following reliability indices and sensitivity vectors:

```
Performance Function 23
beta = 2.2094816
         alpha(32) = 0.9814942, gamma(32) = 0.9814942
         alpha(41) = 0.1593380, gamma(41) = 0.1593380
         alpha(62) = 0.1062089, gamma(62) = 0.1062089
         alpha(89) = 0.0003440, gamma(89) = 0.0003440
Performance Function 76
beta = 2.4495696
         alpha(32) = 0.9011882, gamma(32) = 0.9011882
         alpha(41) = -0.0054455, gamma(41) = -0.0054455
         alpha(62) = -0.4333630, gamma(62) = -0.4333630
         alpha(89) = 0.0051653, gamma(89) = 0.0051653
```

Figure 1.5: OpenSees FERA results for linear-elastic frame example.

This output was obtained using the nested loops shown in Figure 1.6, where the use of Tcl commands is emphasized in obtaining analysis results. The commands `getLSFTags` and `getRVTags` return Tcl lists of performance (limit state) function tags and random variable tags, respectively. FORM analysis results are obtained via the `betaFORM` and `alphaFORM` commands which each take a performance function tag as the first argument. The `alphaFORM` command takes an additional argument for the random variable tag, which is a safer implementation than having Tcl return the entire vector because there is no guarantee that the returned vector would have the expected ordering. An analogous command, `gammaFORM`, is defined in order to return the scaled version of the sensitivity vector proposed by Haukaas and ADK. Also note that the order of performance functions returned by the `getLSFTags` command does not reflect the order in which the functions were defined in the script.

```
foreach perf [getLSFTags] {
    puts "Performance Function $perf"
    puts "beta = [betaFORM $perf]"
    foreach rv [getRVTags] {
       puts " alpha($rv) = [alphaFORM $perf $rv]"
    }
}
```

Figure 1.6: Tcl loops for printing FORM analysis results.

The use of script-level commands such as those shown in Figure 1.6 gives the user much more flexibility in creating custom output than the fixed-format text file that is currently generated. These issues will be emphasized later in the report.

For sanity's sake, the OpenSees FERA results shown in Figure 1.5 are compared with those obtained in Figure 1.7 via the aforementioned MATLAB scripts developed for CE 588.

```
beta = 2.443225

pf = 0.007278

x*(1) = 26785.499590;   alpha(1) = -0.438564
x*(2) = 35.978264;      alpha(2) = 0.898669
x*(3) = 0.012543;       alpha(3) = 0.005134
x*(4) = -0.100265;      alpha(4) = -0.005421

beta = 2.204121

pf = 0.013758

x*(1) = 30712.408350;  alpha(1) = 0.107739
x*(2) = 35.814889;  alpha(2) = 0.981334
x*(3) = 0.000667;  alpha(3) = 0.000302
x*(4) = -0.092978;  alpha(4) = 0.159300
```

Figure 1.7: Home-brewed MATLAB FERA output for linear-elastic frame example.

There is only a slight difference in the OpenSees and MATLAB results, most likely due to discrepancies in the transformation of lognormal RV 1, the elastic modulus of members 1 and 3.[1] For the case that all four random variables are normally distributed, identical results are obtained.

[1] I am certain the OpenSees results are more "reliable" than the MATLAB code I threw together.

1.4 Changes in OpenSees C++ Code

In this section, as will be the case for all items in this report, implementation changes and additions to the OpenSees FE and reliability core are briefly described. The following changes were made in order to perform or facilitate the linear-elastic frame reliability analysis.

1.4.1 Node Parameters

In attempting to map RV 3 to the X-coordinate of node 1, I found that there was an empty if-statement in TclParameterCommands.cpp for the case of node parameters. The following code was added:

```
else if (strstr(argv[2],"node") != 0) {
   if (argc < 4) {
      opserr << "WARNING parameter -- insufficient number of arguments
         for parameter with tag " << paramTag << '\n';
      return TCL_ERROR;
   }

   int nodeTag;
   if (Tcl_GetInt(interp, argv[3], &nodeTag) != TCL_OK) {
      opserr << "WARNING parameter -- invalid node tag\n";
      return TCL_ERROR;
   }

   // Retrieve element from domain
   theObject = (DomainComponent *) theTclDomain->getNode(nodeTag);

   argStart = 4;
 }
```

1.4.2 Uniform Member Loads

When mapping RV 4 to the uniformly distributed member load on element 2, I found that only member point loads were "parameterizable." This was easily fixed by adding an "or" clause for "elementLoad" (shown below) in the setParameter method of the LoadPattern class

```
else if (strstr(argv[0],"elementPointLoad") != 0 ||
         strstr(argv[0],"elementLoad") != 0) {

   if (argc < 3)
      return -1;
```

```
   RVisRandomProcessDiscretizer = false;

   int eleNumber = atoi(argv[1]);
   ElementalLoad *theEleLoad = 0;
   ElementalLoadIter &theEleLoadIter = this->getElementalLoads();
   while ((theEleLoad = theEleLoadIter()) != 0) {
      int eleTag = theEleLoad->getElementTag();
      if (eleNumber == eleTag) {
         return theEleLoad->setParameter(&argv[2], argc-2, param);
      }
   }

   return -1;
}
```

The "elementPointLoad" clause was left in the code for legacy despite the fact the all element loads are mapped in the same fashion.

Item 2

Parameterization Framework

Parameterization framework. Deleting old positioners, etc.

Item 3

Pure Virtual Performance Function Class

The previous design of the GFunEvaluator and LimitStateFunction classes has many calls to the Tcl API in the base class when there are subclasses that have nothing to do with Tcl, e.g., the existing MATLAB g-function evaluator and possible new evaluator based on other scripting languages such as Python. Additional methods specific to TELM analysis have appeared in the interface, not all of which are applicable to every type of probabilistic finite element analysis. In addition, the current implementation of performance functions (previously only g-functions) is based entirely on character-by-character parsing that affects GFunEvaluator, LimitStateFunction, and GradGEvaluator. To separate the interface from the implementation, and to allow for future development of the reliability, sensitivity, and optimization modules, an abstract base class is proposed for all performance functions. The current LimitStateFunction functionality was then moved to a concrete subclass and all of the Tcl-dependency was removed. In addition, two concrete subclasses are created to handle constraint and objective functions for optimization. All of the performance function classes are now container classes for the function strings themselves and do not operate on them directly or indirectly. The following chapter describes to new abstract class for function evaluation. These core changes will be transparent to a user of OpenSees reliability.

Item 4

Pure Virtual Function Evaluator Class

As mentioned in the previous chapter, the current design of the GFunEvaluator class has many calls to the Tcl API in the base class when there are subclasses that have nothing to do with Tcl, e.g., the existing MATLAB g-function evaluator and possible new evaluator based on other scripting languages such as Python. The specific parsing of performance functions and over-reliance on Tcl for making calls to other reliability and OpenSees domain commands has caused some issues with maintenance and extensibility of the code base. Therefore, an abstract base class is proposed for function evaluation and all of the existing calls to the Tcl API are shipped to a concrete subclass. The implementation depends solely on Tcl API calls and does not require parsing for either function or gradient evaluation. In addition, concrete subclass templates were added for future interpreter languages such as Matlab and Python. These core changes will be transparent to a user of OpenSees reliability.

Need to talk about new "storage" methods now part of FunctionEvaluator, and potentially the new Storage classes themselves.

4.1 Over-Reliance on Tcl

With an eventual goal of having OpenSees reliability analysis tied to scripting languages such as Python and MATLAB just as well as it currently is to Tcl, it does not make software engineering sense to have Tcl-related method names and formal arguments in what was intended to be a generic base class, GFunEvaluator. For example, the base class constructor takes a pointer to a Tcl interpreter object

```
GFunEvaluator(Tcl_Interp *theTclInterp,
              ReliabilityDomain *theReliabilityDomain,
              Domain *theOpenSeesDomain);
```

This implies that every subclass that is not Tcl-related will have to pass a null pointer for the first argument to the constructor. In addition, there is a base class method to set current values of all random variables as variables in the Tcl namespace

```
int setTclRandomVariables(const Vector &x);
```

Although this is a very useful method, it should be private to each subclass where such an operation is needed. To drive the point home further, there are also methods to parse reliability syntax for translation to Tcl syntax

```
int nodeTclVariable(int nodeNumber, int direction,
                    char* dispOrWhat, char* varName, char* arrName);
int elementTclVariable(int eleNumber, char* varName, char* restString);
```

As alluded to in Item 7, another goal of this project is to remove parsing of reliability syntax from the OpenSees reliability core and instead use the more robust functions provided by the chosen parsing language, e.g., Tcl. Finally, the only pure virtual method in the GFunEvaluator class takes Tcl-based arguments

```
virtual int tokenizeSpecials(TCL_Char *theExpression, Tcl_Obj *paramList) = 0;
```

It could be impossible for non-Tcl subclasses to provide a meaningful implementation of this method because of the second formal argument.

4.2 Current Class Functionality

The current class structure of GFunEvaluator and its subclasses reflect the desired software design; however, the base class methods mentioned in the previous section nullify these intentions. As shown in Figure 4.1, there are currently four subclasses of GFunEvaluator

- **BasicGFunEvaluator** – This class basically does nothing as it simply returns 0 for the *tokenizeSpecials()* method and does not overwrite any of the base class implementations for g-function evaluation.

- **TclGFunEvaluator** – This class is intended for evaluation of a g-function contained in a user-specified Tcl script. It implements the *tokenizeSpecials()* method by parsing the OpenSees reliability syntax for node displacements (`u_2_1`), element forces (`rec_element_1_section_1_force_2`), etc. then calling the base class methods described in the previous section in order to set variables in the Tcl namespace. This class also overrides the *runGFunAnalysis()* method by invoking the `reset` command on the OpenSees domain, updating all random variables with their current realizations, then calling the *Tcl_EvalFile* function with the filename of the user's Tcl script.

- **OpenSeesGFunEvaluator** – This class is intended for evaluation of the g-function via a simple call to the `analyze` OpenSees command. Its implementation of *tokenizeSpecials* is identical to that of the TclGFunEvaluator class, as is the implementation of *runGFunAnalysis()* with the exception that the `analyze` command is invoked in OpenSees using the *Tcl_Eval* function with the user-specified number of steps and time step. *The overloaded constructor that takes a filename seems to make the TclGFunEvaluator class obsolete??*

- **AnalyzerGFunEvaluator** – This class was introduced for TELM analysis, and in the process, several methods were added to the base class GFunEvaluator interface. These methods are specific only to TELM analysis and should be removed from the base class interface.
- **TclMatlabGFunEvaluator** – This class is deprecated; however, its intention was to use MATLAB in combination with OpenSees in order to evaluate a g-function.

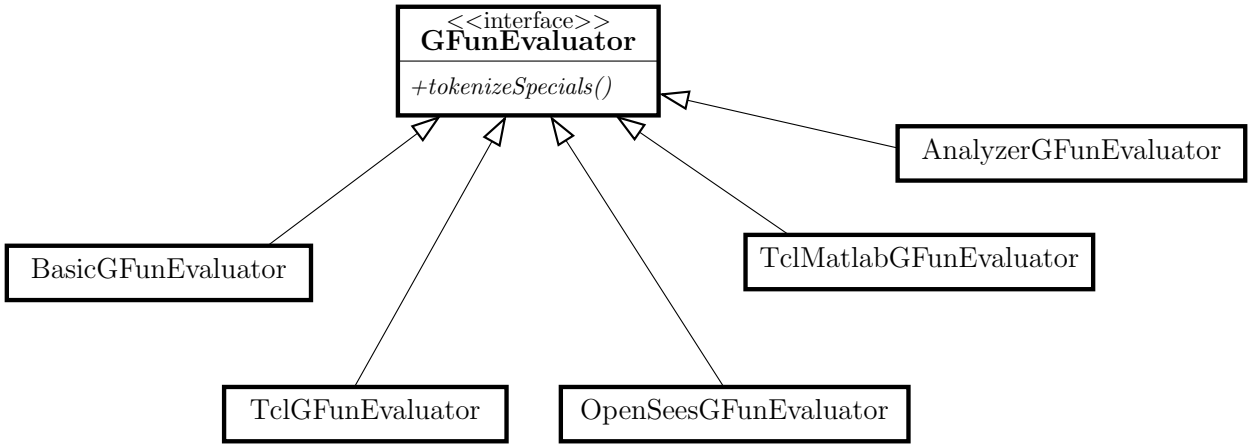

Figure 4.1: Current class structure of GFunEvaluator.

4.2.1 Difference Between runGFunAnalysis and evaluateG

This is more or less a note to self, but the *runGFunAnalysis()* method invokes operations on the finite element domain (`reset` and `analyze`) in order to prepare for evaluation of the limit state function, which is accomplished by calling the *evaluateG()* method. *If we move toward more script-level control of reliability analysis, would the* runGFunAnalysis() *method become obsolete*?

4.3 Proposed Changes

From inspection of the current design, the following changes to the GFunEvaluator class hierarchy are proposed in order to make the design more flexible and extensible for current and future scripting languages:

1. Remove the pointers to Tcl_Interp and ReliabilityDomain objects from the formal argument list of the GFunEvaluator class.
2. For legacy's sake, move the parsing of expressions like `u_2_1` from the base class in to the currently empty BasicGFunEvaluator class.

3. Since their implementations are so similar, combine the TclGFunEvaluator and OpenSeesGFunEvaluator classes in to a single Tcl-based class. In doing so, we should take full advantage of the Tcl API for evaluating g-function expressions.

4. Remove the TELM methods from the base class interface and let them reside only in the AnalyzerGFunEvaluator subclass.

5. Provide a skeletal subclass for Python-based g-function evaluation.

4.3.1 Virtual Methods

The first step to accomplishing the proposed changes is to determine which methods should be (pure) virtual in the GFunEvaluator abstract base class. Instead of the *tokenizeSpecials()* method being pure virtual, it may be more intuitive to make the *runGFunAnalysis()* and *evaluateG()* methods pure virtual. The *tokenizeSpecials()* method can be virtual with a default implementation of do-nothing. The signatures of these methods are shown in Figure 4.2.

```
virtual int runGFunAnalysis(const Vector &x) = 0;
virtual double evaluateG(const Vector &x) = 0;

virtual int tokenizeSpecials(const char *expression) {return 0;}
```

Figure 4.2: Virtual methods for the proposed changes to the GFunEvaluator class.

Item 5

Consistent Gradient and Hessian Class Implementations

The key change was to completely separate the gradient and hessian classes from the probability transformation, design point algorithm, and any Tcl calls. Therefore, the current gradient and hessian classes operate exactly as they should, computing the partial derivatives of any performanceFunction without assumptions on whether the variables are random variables, limit state function parameters, etc. The consequence of this generality is that the size of the gradient vector is potentially different than the size of the gradient vector, for example, required by the design point algorithm for converging to the design point (by definition defined by the size of random variable space). However, the gradient vector is now rapidly resized by querying the parameter class.

The gradient and hessian classes do not perform any parsing of the performanceFunctions and do not, by themselves, evaluate the performanceFunction. Calls are made directly to the functionEvaluator to set variable values in the interpreter of choice, followed by an analysis to compute the results necessary for evaluating the performanceFunction. It is important to note that due to the parameterization framework, it is also not necessary to pass around vectors and matrices containing realizations of variables, gradients, etc. This approach eliminates a considerable amount of overhead from previous versions of the reliability code base. Another important note is that the gradient computations can be omitted entirely if the user specifies an analytic gradient expression (see Chapter 7).

5.1 Gradient

There are currently two concrete subclasses of the Gradient base class, FiniteDifferenceGradient and ImplicitGradient. These two classes perform similar activities to previous implementations, but with less code duplication. The FiniteDifferenceGradient perturbs individual parameters to form the gradient vector. However, the perturbations now come directly from the Parameter class and are therefore specific to the type of parameter being considered. The ImplicitGradient class now enables previous functionality associated with

the direct differentiation method (DDM), but also enables future types of analysis because it is completely general in it's implementation. This is accomplished through the calls to the getSensitivity method of the Parameter classes. The different derived classes of parameters may perform different computations to achieve this sensitivity calculation (for example, the parameter could point to an external function that can compute it's own sensitivities). Currently this is directly associated with current hierarchy of reliability calls already established in the previous version of the code. For example, sensitivity of nodal displacement would be obtained by calls from the Parameter class to getDispSensitivity().

As with previous implementations, the implicit gradient is actually performing a finite difference scheme on the performanceFunction to uncover the $\frac{\partial g}{\partial \mathbf{u}}$ in $\frac{\partial g}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \theta}$. The analytic gradient expression can be used to circumvent the need for this finite difference computation. In addition, this scheme implies you cannot have an explicit parameter appear in the same performanceFunction as an implicit parameter. For example, if there are two parameters: θ_1 is modulus E and θ_2 is a nodal displacement, then $g(\theta_1, \theta_2) = \theta_1 + \theta_2$ is not a viable performanceFunction. The implicit computation would need to consider $\frac{\partial g}{\partial \theta_1} + \frac{\partial g}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \theta_1}$. Such a problem can be solved using FiniteDifferenceGradient.

5.2 Hessian

There is currently only one concrete subclass of the Hessian base class, FiniteDifferenceHessian. As with FiniteDifferenceGradient, the hessian obtains the perturbations directly from the parameters. The diagonal terms of the hessian matrix are obtained using a central difference approximation whereas the off-diagonal terms are obtained from a forward difference approximation.

5.3 Transformations

An important consequence of the implementation is that the partial derivatives are with respect to the original performanceFunction parameters. Therefore, whenever the gradient or hessian is required in transformed space (i.e., in standard normal space for the design point algorithm), vector/matrix requires a transformation. Remaining consistent with the separation of base classes in the new reliability code, any such transformations would be performed only the subclasses requiring specific functionality. The transformation of the gradient to standard normal space is well understood using the Jacobian.

$$\nabla_{\mathbf{x}} g_i = [\frac{\partial g_i}{\partial x_1}\frac{\partial g_i}{\partial x_2}...\frac{\partial g_i}{\partial x_n}] \tag{5.1}$$

$$\nabla_{\mathbf{u}} G_i = \nabla_{\mathbf{x}} g_i \mathbf{J}_{\mathbf{x},\mathbf{u}} \tag{5.2}$$

where g_i is the ith limit state function in random variable space $\mathbf{x}$ and G_i is the ith limit state function in standard normal space $\mathbf{u}$.

However, the transformation of the hessian to standard normal space is not a simple task when the transformation is not linear (which is the case for most of the random variables and probability transformations). Transformation of the hessian can be accomplished by combination of terms:

$$\mathbf{H_u}G_i = \mathbf{J}^T_{\mathbf{x,u}}\mathbf{H_x}g_i\mathbf{J_{x,u}} + \nabla_{\mathbf{x}}g_i\nabla_{\mathbf{u}}\mathbf{J_{x,u}} \tag{5.3}$$

The transformation of the gradient to standard normal space is performed by the design point algorithm and other analysis modules as necessary (e.g., FOSM). The abbreviated code to accomplish this is illustrated below.

```
result = theGradientEvaluator->computeGradient(gFunctionValue);
Vector temp_grad = theGradientEvaluator->getGradient();

// map gradient from all parameters to just RVs
for (int j = 0; j < numberOfRandomVariables; j++) {
    int param_indx = theReliabilityDomain->getParameterIndexFromRandomVariableIndex(
    (*gradientInOriginalSpace)(j) = temp_grad(param_indx);
}

// Get Jacobian x-space to u-space
result = theProbabilityTransformation->getJacobian_x_to_u(*Jxu);

// Gradient in standard normal space
gradientInStandardNormalSpace->addMatrixTransposeVector(0.0, *Jxu, *gradientInOrigin
```

The transformation of the hessian to standard normal space occurs only in the CurvatureFitting subclass of the FindCurvatures base class. The abbreviated procedure is illustrated below.

```
// compute Hessian
int result = theHessianEvaluator->computeHessian();

// transform Hessian
Matrix hessU(nrv,nrv);
Matrix temp_hess(numberOfParameters,numberOfParameters);
temp_hess = theHessianEvaluator->getHessian();

// map hessian from all parameters to just RVs
Matrix hessX(nrv,nrv);
for (int j = 0; j < nrv; j++) {
    int param_indx_j = theReliabilityDomain->getParameterIndexFromRandomVariableInde
    for (int k = 0; k <= j; k++) {
        int param_indx_k = theReliabilityDomain->getParameterIndexFromRandomVariable
        hessX(j,k) = temp_hess(param_indx_j,param_indx_k);
```

```
            hessX(k,j) = hessX(j,k);
        }
    }

    // Get Jacobian x-space to u-space
    result = theProbabilityTransformation->getJacobian_x_to_u(Jxu);

    // Hessian in standard normal space (if the transformation is linear)
    hessU.addMatrixTripleProduct(0.0,Jxu,hessX,1.0);

    // Now add the nonlinear term
    hessU += hessNL;
```

The nonlinear term in the hessian transformation is currently computed using finite differences, and therefore will suffer from known accuracy problems and computational demands.

Item 6

Reliability Analyses and System Reliability

Flow control of reliability analyses, SORM, system analysis.... need to complete.

6.1 Reliability analyses

Can change the analysis options in-between reliability analyses, investigate response using getAlpha, etc. There are now defaults provided if you don't select all of the reliability analysis options, performed through previous `inputCheck` function in the TclReliabilityBuilder. Also, the following reliability analysis options were revised to operate with the new base class structure: FOSM, FORM, ImportanceSampling, SORM, and System. Need to complete.

6.2 Sensitivity to Random Variable Distribution Parameters

Standard FORM analysis returns information on not only the design point, but also the importance (unit) vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\gamma}$. These two vectors show the relative importance of the random variables in $\mathbf{u}$ and $\mathbf{x}$ spaces, respectively (actually the latter is in an equivalent normal space). Previous versions of the reliability code also allowed the computation of the so-called $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ sensitivity vectors for normal, lognormal, and uniform random variables. These two vectors are not usually normalized and show the relative importance of each random variable with respect the mean and standard deviation, respectively. Computation of these two vectors has now been formalized in the code revisions to allow computation of the $\boldsymbol{\delta}$ and $\boldsymbol{\eta}$ for any random variable. In addition, the formulation allows future computation of sensitivities with respect to the random variable distribution parameters.

The entries of the two vectors are defined as $\delta_i = \sigma_i \frac{\partial \beta}{\partial \mathbf{M}}$ and $\delta_i = \sigma_i \frac{\partial \beta}{\partial \mathbf{D}}$, where $\mathbf{M}$ is the mean vector and $\mathbf{D}$ is the standard deviation vector. The partial derivatives can be obtained from the dot product of $\boldsymbol{\alpha}$ with either $\frac{\partial \mathbf{u}^*}{\partial \mathbf{M}}$ or $\frac{\partial \mathbf{u}^*}{\partial \mathbf{D}}$, respectively. The derivatives with re-

spect to the mean and standard deviation, evaluated at the design point in standard normal space are obtained from the probability transformation and depend directly on the type of random variable transformation employed. Only the Nataf and statistically independent transformations are operable in OpenSees, and both work with the new mean and standard deviation sensitivity vectors. In the statistically independent (AllIndependentTransformation) case, the derivative is passed directly to the random variables themselves which return $\frac{\partial F}{\partial \mu}\frac{1}{\phi(u_i)}$ and $\frac{\partial F}{\partial \sigma}\frac{1}{\phi(u_i)}$, respectively. In the Nataf case, the derivative with respect to the lower Cholesky transformation matrix $\mathbf{L}'$ is also required. Currently the transformation computes this derivative using finite differences. However, the statistically independent case is exact.

The RandomVariable base class returns the sensitivity of the CDF with respect to the mean and standard deviation by assembling two derivatives from the derived classes (and returning the dot product). These two derivatives are $\frac{\partial F}{\partial \boldsymbol{\theta}}$ and $\frac{\partial \boldsymbol{\theta}}{\partial \mu}$ (for mean sensitivity), where $\boldsymbol{\theta}$ is the vector of random variable distribution parameters (changes size with each random variable). For example, to obtain the mean sensitivity, calls are made to the getCDFparameterSensitivity() and getParameterMeanSensitivity() methods. The dot product is performing the following computation:

$$\frac{\partial \beta}{\partial \mu} = \frac{\partial \beta}{\partial \theta_1}\frac{\partial \theta_1}{\partial \mu} + \frac{\partial \beta}{\partial \theta_2}\frac{\partial \theta_2}{\partial \mu} + \cdots \tag{6.1}$$

For all of the random variable distributions in OpenSees, the closed form sensitivities were calculated and implemented. The complete set of sensitivities is contained in Appendix B. It should be noted that the sensitivities involving the derivative of the gamma or incomplete gamma function were not coded, as this method is not readily available in OpenSees. The following derivatives were therefore computed with finite differences: $\frac{\partial F}{\partial \nu}$ in ChiSquare distribution, $\frac{\partial F}{\partial k}$ in Gamma distribution, and $\frac{\partial u}{\partial \mu}$ and $\frac{\partial k}{\partial \mu}$ in Weibull distribution.

While not yet implemented, it is now a simple extension to compute sensitivities with respect to random variable distribution parameters because of the methods discussed above. The random variables all return sensitivities with respect to the random variable distribution parameters directly. For example, the sensitivity of the reliability index to one random variable distribution parameter can be obtained from the previous results as:

$$\frac{\partial \beta}{\partial \theta_i} = \frac{\partial \beta}{\partial \mu}\frac{\partial \mu}{\partial \theta_i} + \frac{\partial \beta}{\partial \sigma}\frac{\partial \sigma}{\partial \theta_i} \tag{6.2}$$

6.3 Second Order Reliability Method

While the other reliability analysis options were merely modified to work with the new code base (and in some cases simplified drastically), additional functionality was added to SORM and FindCurvatures. CurvatureFitting was added as a FindCurvatures algorithm. Curvature fitting SORM requires the computation of the hessian matrix (see Chapter 5), and the computation of the eigenvalues of a matrix with reduced dimensions (number of random variables minus one). The hessian is computed for all parameters and then transformed to standard normal space as described in Chapter 5. Additionally, the orthonormal matrix

$\mathbf{P}$ is derived based on Gram-Schmidt orthogonalization using the $\boldsymbol{\alpha}$ vector as the last row. The curvatures are obtained from the eigenvalues of the matrix $\mathbf{A} \in \Re^{nrv-1,nrv-1}$. The eigenvectors define the principal axes.

$$\mathbf{A} = \frac{\mathbf{P}^T(\mathbf{H_u}G)\mathbf{P}}{\|\nabla_{\mathbf{u}}G\|} \tag{6.3}$$

The norm of the gradient is obtained from a previous FORM analysis, as are the $\boldsymbol{\alpha}$ vector and the design point. The eigenvalue computation is performed using LAPACK dgeev.

Preliminary work was performed to merge the previous FirstPrincipalCurvature and CurvaturesBySearchAlgorithm classes, because they both currently perform the same calculations. CurvaturesBySearchAlgorithm was never completed in the original implementation.

6.4 System Analysis

SystemAnalysis existed in the previous code base with the option only for systems defined by components all in series or all in parallel. This functionality was extended to allow generalized system reliability analysis to be performed. The Tcl syntax for executing a system reliability analysis is as follows:

```
runSystemAnalysis fileName? analysisMethod? (allInParallel | allInSeries | cutsets) <-Nm
```

where `fileName` is the file name for the output, and `analysisMethod` is one of PCM, IPCM, MVN, or SCIS. The analysis methods operate on individual cutsets to evaluate the multivariate function $\Phi_n(\mathbf{B}, \mathbf{R})$, where $\mathbf{B}$ is the vector of reliability indices corresponding to the limit state functions comprising the cutset, and $\mathbf{R}$ is the correlation matrix. Two of the analysis methods are variants based on the work of Pandey and utilize conditional marginal distributions. The original algorithm is the product of conditional marginals (PCM) that was originally intended for parallel systems. A second improved product of conditional marginals (IPCM) was subsequently introduced for series systems. The other two methods are simulation based. The first is the multivariate normal method from Genz [] and the second is the sequential conditioned importance sampling method from Ambartzumian []. The two simulation-based methods take two additional arguments that dictate how many samples to complete before returning the system probability estimates. The maximum number of simulations is specified using `Nmax` and the tolerance on the probability of failure estimate is specified using `tol`. These two arguments default to 2e5 and 1e-6, respectively.

If system analysis is preceded by FORM analysis, the method will compute the $\mathbf{B}$ vector and $\mathbf{R}$ matrix from the design point of the specified limit state function. If the user desires to perform system analysis independent of a component FORM analysis, the $\mathbf{B}$ vector and $\mathbf{R}$ matrix can be specified directly using the input file switches shown in the command line syntax shown above. The type of system analysis to be performed is one of either allInParallel, allInSeries, or cutsets. Generalized system reliability analysis is possible using the cutsets option. Cutsets contain components in parallel, and the cutsets are arranged in

series. Before the system analysis may be performed, the cutsets need to be created in the interpreter. The command syntax is as follows:

```
cutset tag? lsf1? ... <lsfm?>
```

where the kth cutset may contain 1 or more limit state functions. The limit state functions are specified using the tags that follow the cutset tag. A negative limit state function tag may be used to indicate that the user would like the complement of the event (survival) rather than the failure event. The generalized system reliability analysis will degenerate into a series system if all the cutsets contain only one component. Similarly the system generates into a parallel system if only a single cutset is used with multiple components.

In addition to the aforementioned analysis methods for computing the system probability of failure, OpenSees will also compute several system bounds depending on the analysis type. For parallel systems (`allInParallel`), the uni-component bounds of Boole are computed, and therefore may be wider than desired. For series systems (`allInSeries`), the bi-component bounds of Kounias, Hohenbichler, and Ditlevsen (so called KHD bounds) are computed. Both the upper and lower KHD bi-component bounds require checking all permutations of the components to obtain the tightest bounds. Technically this requires $(n+1)!$ permutations, where n is the number of components. This can be computationally challenging; therefore, OpenSees accomplishes this by randomly ordering the first $500n$ permutation sets. The results are accurate only for smaller numbers of components; however, the random ordering may improve the bounds for large numbers of components as well.

Generalized system reliability analysis is accomplished using the inclusion-exclusion rule. It is necessary to consider the addition (or subtraction) of all n choose k combination probabilities of the components. Therefore, the accuracy of the reported system probability of failure will degenerate as the number of components becomes significantly larger. The speed of the system reliability algorithm will also decrease significantly due to number of individual parallel systems that need to be evaluated in such a context.

Item 7

Analytic Expressions for g-Function Gradients

A useful feature that I have pondered in my research of girder live load reliability analysis is user-specified analytic expressions for the gradient of the i^{th} performance function, g_i, with respect to the j^{th} random variable, x_j. Using the chain rule, we have

$$\frac{\partial g_i}{\partial x_j} = \frac{\partial g_i}{\partial \mathbf{U}_f}\frac{\partial \mathbf{U}_f}{\partial x_j} + \frac{\partial g_i}{\partial \mathbf{P}}\frac{\partial \mathbf{P}}{\partial x_j} + \dots \tag{7.1}$$

where $\mathbf{U}_f$ is nodal response and $\mathbf{P}$ represents any derived response quantity recovered from the nodal displacements, such as member forces. The two derivatives in each product on the right-hand side are computed separately. Although OpenSees has DDM capabilities for $\partial \mathbf{U}_f/\partial x_j$ and $\partial \mathbf{P}/\partial x_j$,[1] there still is a reliance on finite differences to get $\partial g_i/\partial \mathbf{U}_f$ and $\partial g_i/\partial \mathbf{P}$.

In this part of the report, a simple method to allow the user to input $\partial g_i/\partial x_j$ directly is implemented. The implementation is likely to change later in the report after analysis results are removed from LimitStateFunction class.

7.1 Desired Functionality

As demonstrated in Item 1, we would like to move away from unnecessary parsing, within the OpenSees reliability core, of a performance function. To this end, Tcl commands are used for response quantities when declaring performance functions

```
performanceFunction 76 "0.15-\[nodeDisp 2 1\]"
performanceFunction 23 "1500.0+\[sectionForce 1 1 2\]"
```

[1] Actually it's the derivative with respect to an FE parameter combined with a Boolean mapping of the parameter to a random variable.

Upon inspection of the OpenSeesGradGEvaluation.cpp file, which is the "DDM" implementation, it is noted that the performance (limit-state) function is evaluated twice with an original and a perturbed value to get $\partial g_i/\partial \mathbf{U}_f$ via a finite difference calculation. Then this approximation is multiplied by the "DDM" gradient of the nodal response, $\partial \mathbf{U}_f/\partial x_j$ in order to get $\partial g_i/\partial x_j$. An identical approach is taken for derivatives with respect to derived response quantities.

```
gradPerformanceFunction $tag_gi $tag_xj ‘‘analytic expression’’
```

Figure 7.1: Proposed Tcl command for analytic gradient of the g-function.

For the user to input the "exact" derivative of a g-function, there would have to be OpenSees/Tcl commands along the lines of that shown in Figure 7.1, where `tag_gi` is the tag of the i^{th} performance function and `tag_xj` is the tag of the j^{th} random variable. While this appears to necessitate a large number of commands be issued when there are several parameters, the script-level looping commands described in Item 1 can be used to make the object creation compact, as shown in Figure 7.2. Such a loop can be written for each performance function defined for a reliability analysis. Note that the `sensNodeDisp` Tcl command returns the "DDM" gradient of a nodal displacement using the same inputs as the `nodeDisp` command plus an additional input indicating the parameter with respect to which the gradient is taken (idential approach for the `sensSectionForce` command).

```
foreach {rvTag paramTag} [getRVPositioners] {
    gradPerformanceFunction 76 $rvTag "-\[sensNodeDisp 2 1 $paramTag\]"
    gradPerformanceFunction 23 $rvTag "\[sensSectionForce 1 1 2 $paramTag\]"
}
```

Figure 7.2: Tcl loop to create proposed analytic gradient expressions.

When creating objects for the gradient of a performance function, iteration takes place over the random variable positioners so that each random variable is mapped to its associated parameter of the finite element model, as shown in Figure 7.2. The `getRVPositioners` command returns a list of random variable and parameter tag pairs. Also, it is generally not possible to iterate over the performance functions in order to accomplish the gradient definitions shown in Figure 7.2 because the analytic expressions are unique to each performance function and must be input explicitly by the user.

7.2 Proposed Implementation

Barring unforeseen circumstances, it is not necessary to introduce a new container class to the reliability domain in order to store expressions for the gradient of a performance

function. Instead, private data and public methods can be added to the LimitStateFunction class which is a container class for the performance function.

An STL map is used to store integer keys and associated strings that contain the gradient expressions. The map is declared in the private data section of LimitStateFunction.h:

```
private:
   map<int, string> mapOfGradientExpressions;
```

To retrieve and manipulate this information, the following public methods are declared for the LimitStateFunction class

```
public:
   int addGradientExpression(const char *expression, int rvTag);
   int removeGradientExpression(int rvTag);
   const char* getGradientExpression(int rvTag);
```

The objective of each method is fairly self-explanatory and the underlying implementations of these methods call the appropriate STL map functions.

7.3 Tcl Reliability Model Builder

To make the desired functionality available in an OpenSees/Tcl script, a `gradPerformanceFunction` command is added to the list of commands in TclReliabilityBuilder.cpp. The `addGradLimitState` function is defined in this file and its implementation follows that of `addLimitState`.

7.4 Linking with the Gradient Evaluator

Now that gradient expressions can be created in an OpenSees/Tcl script and stored in the reliability domain (in the LimitStateFunction class), using the expressions during a reliability analysis is the next step. To simply test out the idea, regardless of good software engineering practice at this point, the section of code shown in Figure 7.3 is inserted in the `computeGradG` method of the OpenSeesGradGEvaluation class with an early return.

As seen in Figure 7.3, a short loop is added after the active limit state function is "downloaded" from the reliability domain. The loop iterates over all random variables, in which the RV tag is obtained and used to get its associated gradient expression for the current limit state function. Then, the gradient expression is evaluated using the `Tcl_ExprDouble` function and the result of this expression is inserted in the `grad_g` vector.

Initially, the implementation appeared not to work, i.e., significantly different results were obtained for the linear-elastic frame example of Item 1 when using finite difference versus analytic evaluation of the g-function gradient. After some investigation, it was discovered that the problem lies in the DDM implementation of member loads. As shown in Figure 7.4, similar results were obtained between the proposed analytic gradient expressions and the finite difference approximations after removing the uniformly distributed load on element 2 from the list of random variables.

```
// "Download" limit-state function from reliability domain
int lsf = theReliabilityDomain->getTagOfActiveLimitStateFunction();
LimitStateFunction *theLimitStateFunction =
  theReliabilityDomain->getLimitStateFunctionPtr(lsf);

for (int i = 0; i < nrv; i++) {
  RandomVariable *theRV = theReliabilityDomain->getRandomVariablePtrFromIndex(i);
  int tag = theRV->getTag();
  const char *gradExpression = theLimitStateFunction->getGradientExpression(tag);
  double result = 0;
  if (Tcl_ExprDouble( theTclInterp, gradExpression, &result) == TCL_ERROR) {
    opserr << "ERROR OpenSeesGradGEvaluator -- error in Tcl_ExprDouble
        for the analytic gradient command" << endln;
    return -1;
  }
  (*grad_g)(i) += result;
}

return 0;
```

Figure 7.3: Code to use analytic expressions in the OpenSees gradient g-function class.

7.5 Changes in OpenSees C++ Code

The following changes were made in order to implement analytic expressions for the gradient of a limit state function.

7.5.1 The LimitStateFunction Class

The aforementioned public methods and private data for the string container of analytic gradient expressions were added to LimitStateFunction.h and LimitStateFunction.cpp.

7.5.2 Tcl Command

The associated Tcl command, `gradPerformanceFunction` was checked in as part of the TclReliabilityBuilder.cpp file.

7.5.3 RandomVariablePositioner

To enable the `getRVPositioners` Tcl command, methods to get the random variable and parameters tags associated with a random variable positioner object were added.

The `getRVPositioners` Tcl command was added to TclReliabilityBuilder.cpp.

```
Analytic Gradients
Performance Function 23
beta = 2.2380832
         alpha(32) = 0.9941831, gamma(32) = 0.9941831
         alpha(62) = 0.1077010, gamma(62) = 0.1077010
         alpha(89) = -0.0007124, gamma(89) = -0.0007124
Performance Function 76
beta = 2.4496044
         alpha(32) = 0.9011834, gamma(32) = 0.9011834
         alpha(62) = -0.4334066, gamma(62) = -0.4334066
         alpha(89) = 0.0052057, gamma(89) = 0.0052057

Finite Difference Gradients
Performance Function 23
beta = 2.2380833
         alpha(32) = 0.9941833, gamma(32) = 0.9941833
         alpha(62) = 0.1076992, gamma(62) = 0.1076992
         alpha(89) = -0.0007124, gamma(89) = -0.0007124
Performance Function 76
beta = 2.4496059
         alpha(32) = 0.9011994, gamma(32) = 0.9011994
         alpha(62) = -0.4333735, gamma(62) = -0.4333735
         alpha(89) = 0.0052046, gamma(89) = 0.0052046
```

Figure 7.4: OpenSees FERA results for linear-elastic frame example using user-defined analytic expressions and finite differences for the gradient of each g-function.

Item 8

Control of FEA during Reliability Analysis

With the current implementation of the g-function evaluator, it is necessary to tell this object which finite element analysis commands should be invoked prior to running a reliability analysis

```
gFunEvaluator OpenSees      -analyze 1
```

This hands control of the FEA over to the reliability domain, which can make it cumbersome for a user to run repeated reliability analyses inside an analysis loop or to run a reliability analysis based on a condition attained during an analysis. In addition, the command structure is counter-intuitive to the script-level control offered by the commands added to Tcl for FEA in OpenSees.

Item 9

Conclusions and Future Work

9.1 Conclusions

There exists a significant amount of information in the literature on how to perform reliability, sensitivity, and optimization analyses for a variety of problems. There are also a plethora of software tools to enable computation of numerical solutions for complex problems. However, there are still very few software platforms that are enabled with multiple functionalities. Specifically, it is of interest in performance-based earthquake engineering (PBEE) to perform nonlinear finite element analysis in the presence of uncertainties. Therefore, it is desirable to have a finite element analysis platform that has the ability to compute response sensitivities directly, solve reliability problems with specified uncertain quantities in the models, and solve optimization problems that consider response quantities in the objective.

9.2 Future Work

The changes made as part of this project are by no means comprehensive, and additional work is necessary in the future to allow truly coupled finite element, reliability, sensitivity, and optimization analyses. Below is a list of issues that were discovered during our work that should be addressed in future work.

- `TclReliabilityBuilder` does not use base class static pointers for analysis objects.
- While the Type 1 Largest (T1L) distribution is equivalent to the Gumbel distribution, it is implemented at the moment as a direct copy of the Gumbel file. This can be better implemented internally in the future so only a single random variable distribution derived class is needed.
- Search directions that are not HLRF-based (e.g., PolakHe and SQP) have not been tested or evaluated for functionality. Unraveling this is not a trivial task due to the fact that several of the available search directions utilize multiple inheritance. They are derived from several other base classes, such as meritFunction, HessianApproximation,

etc. and vary based on the search direction implemented. Authors suggest multiple inheritance should be realigned with common practice to use only single inheritance.

- The optimization portion of the code (and interface with SNOPT) needs to be brought in line with the new base classes introduced and the functionality now provided by the higher-level reliability classes. This should significantly reduce code duplication and allow more seamless extensions in the future.

- While significant additions were made to random variable distribution parameter sensitivities as part of this work, not all of the distributions were implemented. The remaining distributions are Type 3 Smallest, Laplace, and Pareto.

- The conversion of `NatafProbabilityTransformation` from FERUM to OpenSees resulted in a few problems that are rarely encountered. For example, for determining the Nataf correlation factors for some distributions (i.e., Beta distribution), the wrong random variable distribution parameters are used; therefore, the Nataf factors are not correct. Also, the code to determine the correlation structure was written as a sequential if-then-else construct (with more than 128 nested statements). Therefore, on Windows, not all of the random variable pairs are available.

- The tail-equivalent linearization method (TELM) code introduced by Fujimura and Der Kiureghian contained duplication of many of the original reliability code issues, plus some additional ones. It is not functional at the moment, and will require some effort to bring it in line with both the new reliability higher-level functionality, as well to make it more object-oriented in nature.

- The analysis method `GFunVisualizationAnalysis` is no longer functional.

- The collection of files pertaining to `rootFinding` will need some thought and subsequent revisions. The current use of `SecantRootFinding` in some of the nested statements in `ArmijoStepSizeRule` is not functional with the current changes. In addition, there appear to be several base classes that are not related (`ZeroFindingAlgorithm` and `RootFinding`).

- The `AllIndependentTransformation` is just a copy of `NatafProbabilityTransformation` but with the correlation structure set to the identity matrix. This needs to be properly coded in the future for performance of the code, so that it does not do dense matrix-matrix and matrix-vector multiplies, decompositions to allow the transformation from x to z, z to u, etc.

- Some of the domain components still employ sequential tagging and do not have associated iterators (`filter`, `modulatingFunction`, and `spectrum`).

- `PrincipalPlane` does not appear to be a domain component (tagged object), but an analysis result method. However, appears in `components`.

- `UserDefined` random variable distribution needs to be properly coded so that analysis not performed in `TclReliabilityBuilder`, and has consistent methods with the new version of the other random variables.

Appendices

Appendix A

Random Variable Distributions

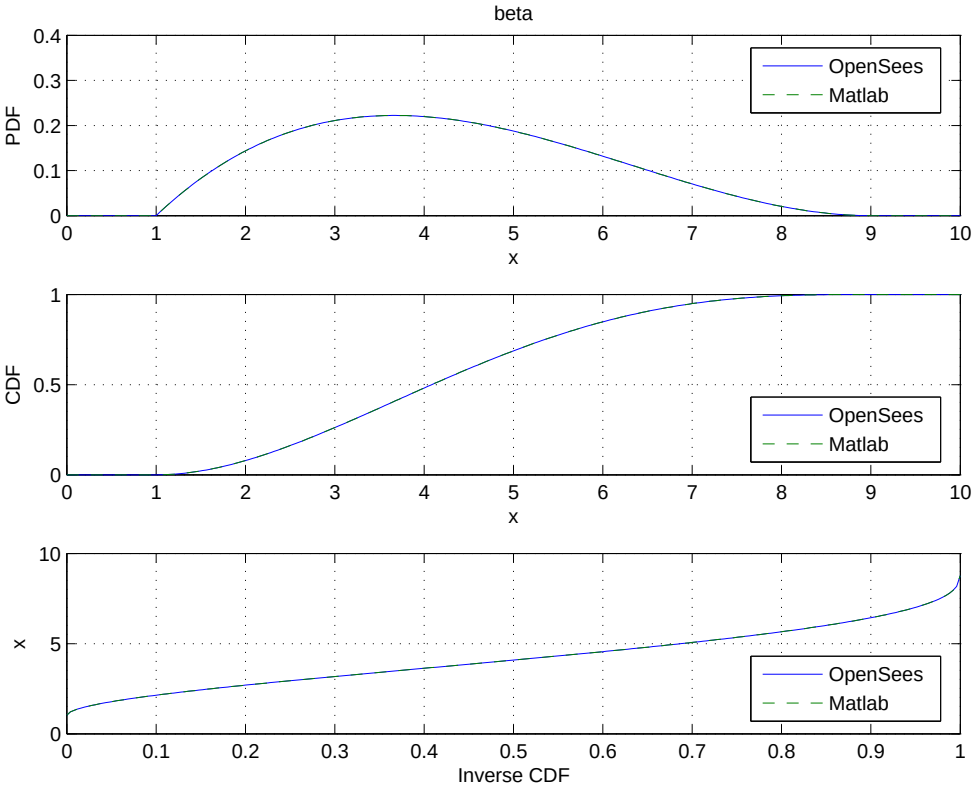

Figure A.1: Beta distribution with parameters $\mathrm{Bet}(a, b, q, r)$.

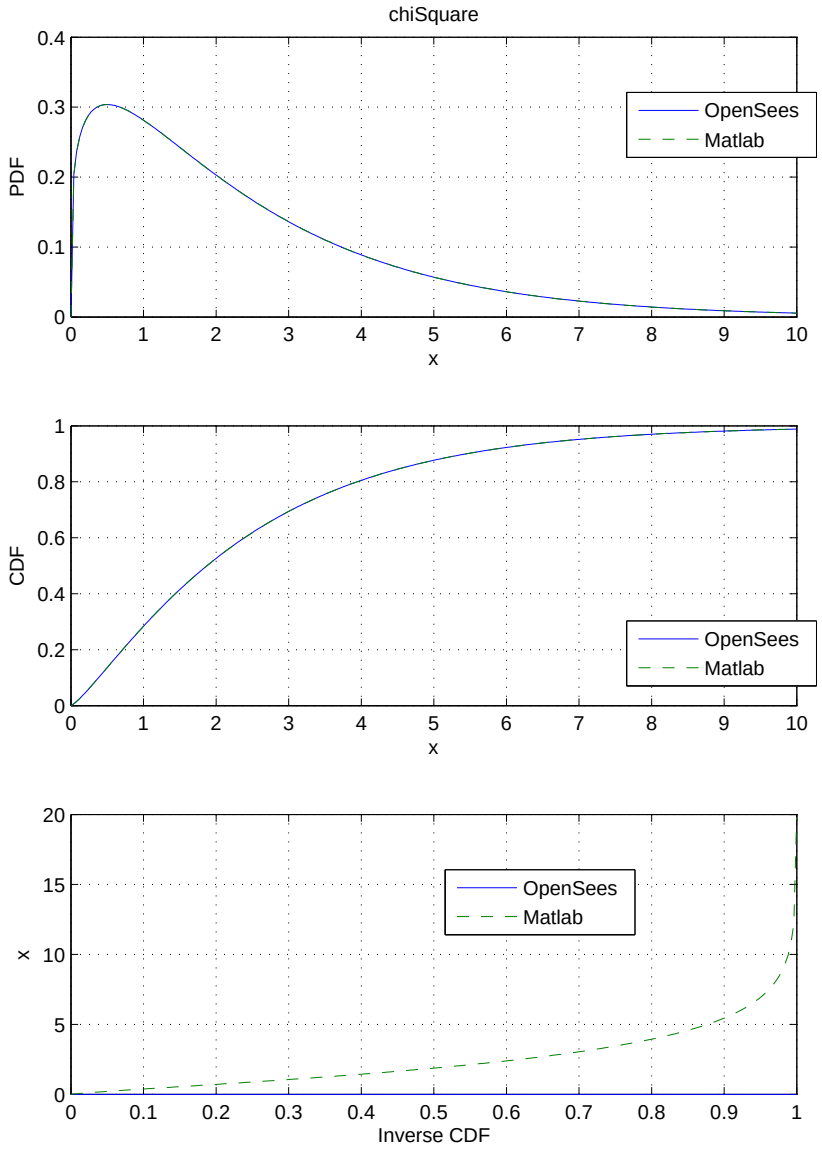

Figure A.2: Chi-square distribution with parameters $\chi^2(\nu)$.

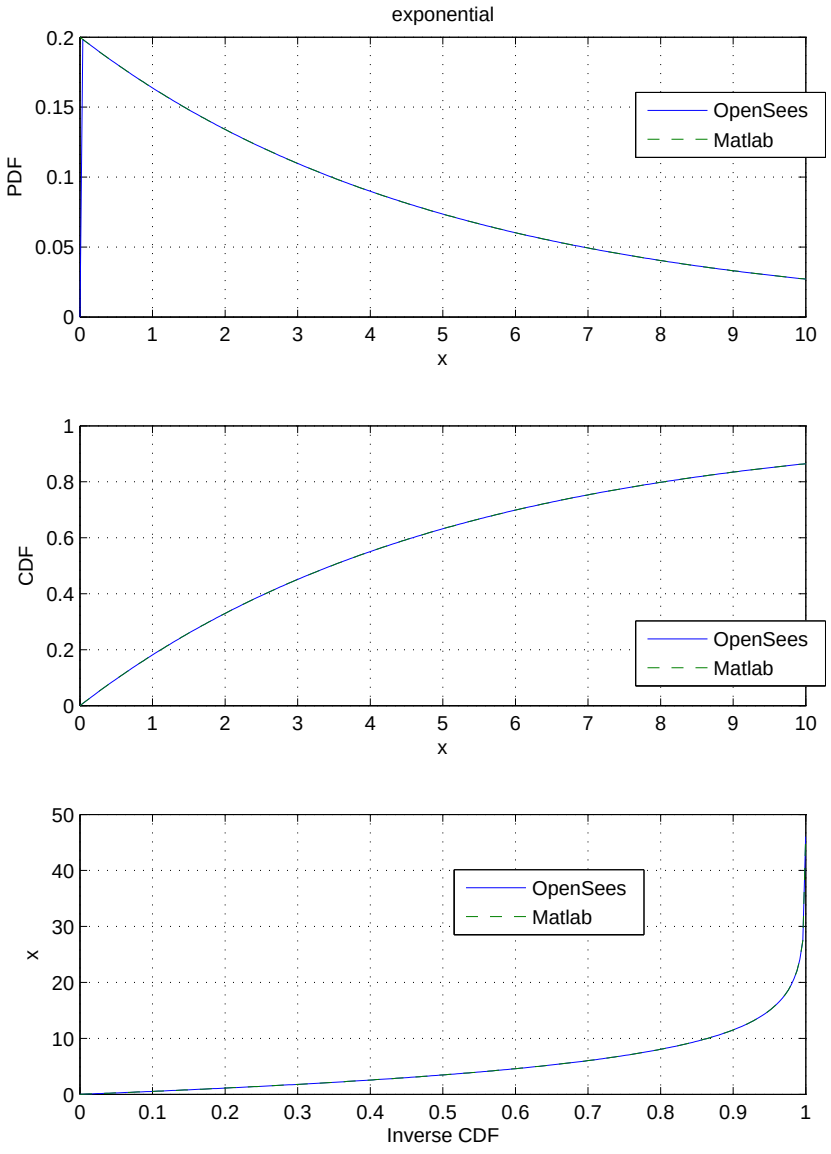

Figure A.3: Exponential distribution with parameters Exp(λ).

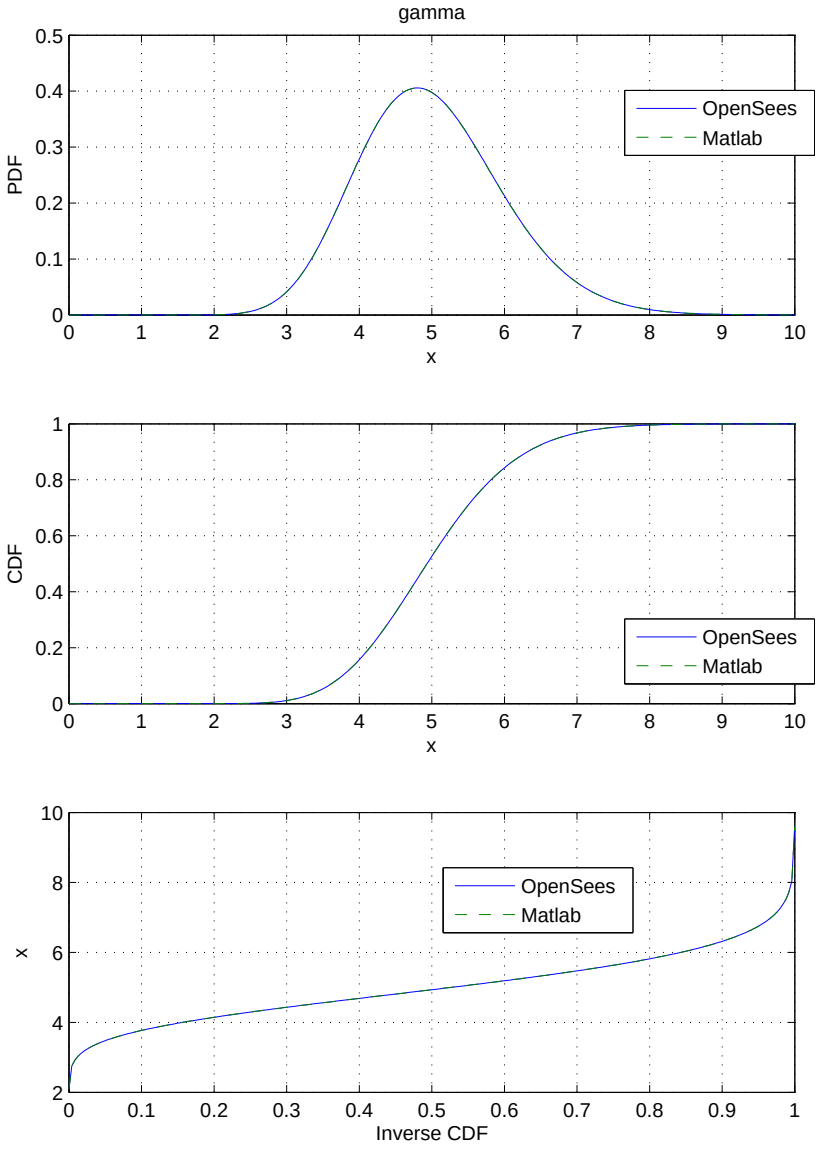

Figure A.4: Gamma distribution with parameters Gam(k, λ).

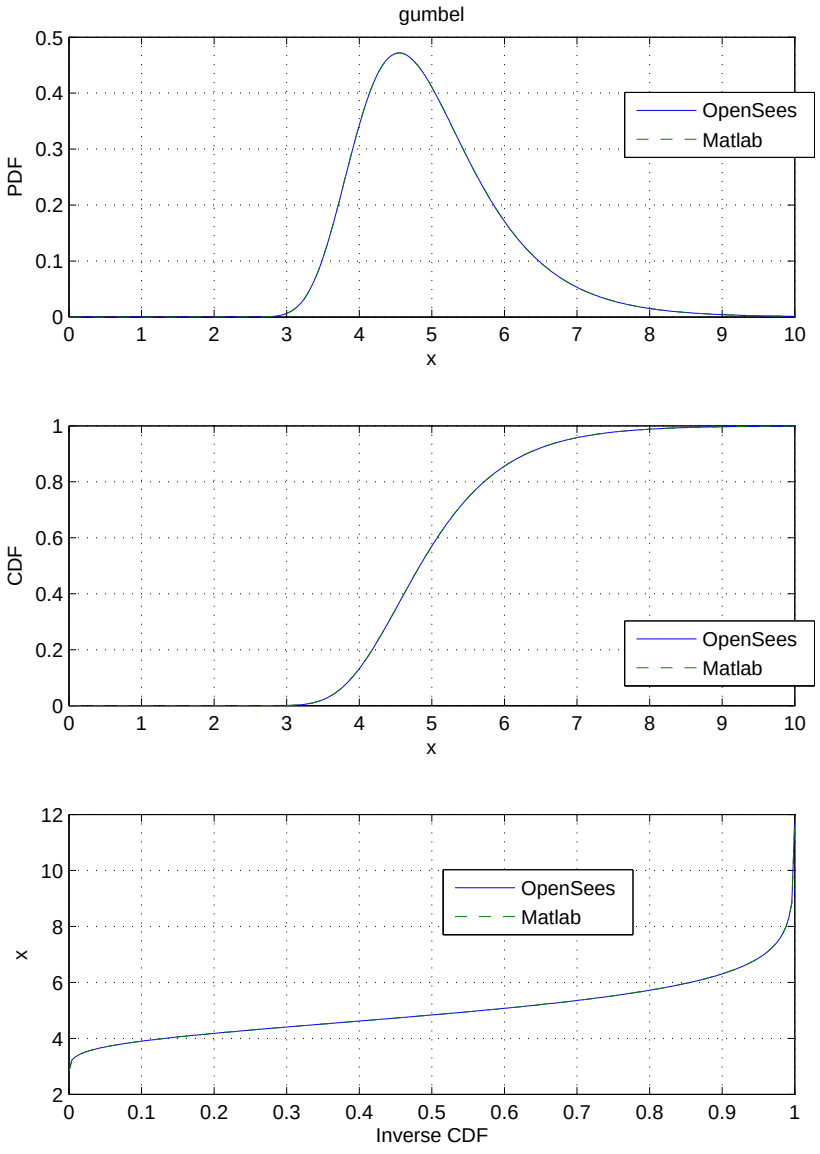

Figure A.5: Gumbel distribution with parameters Gmb(u, α).

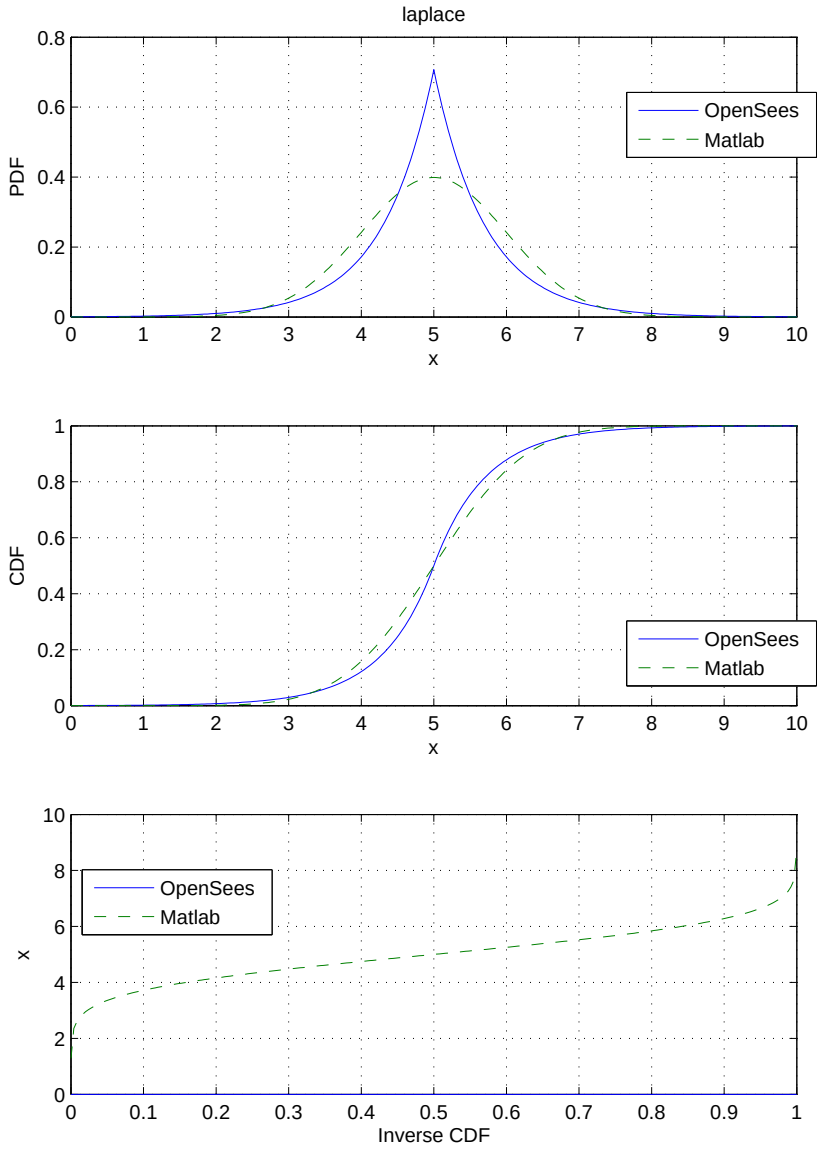

Figure A.6: Laplace distribution with parameters Lap(α, β).

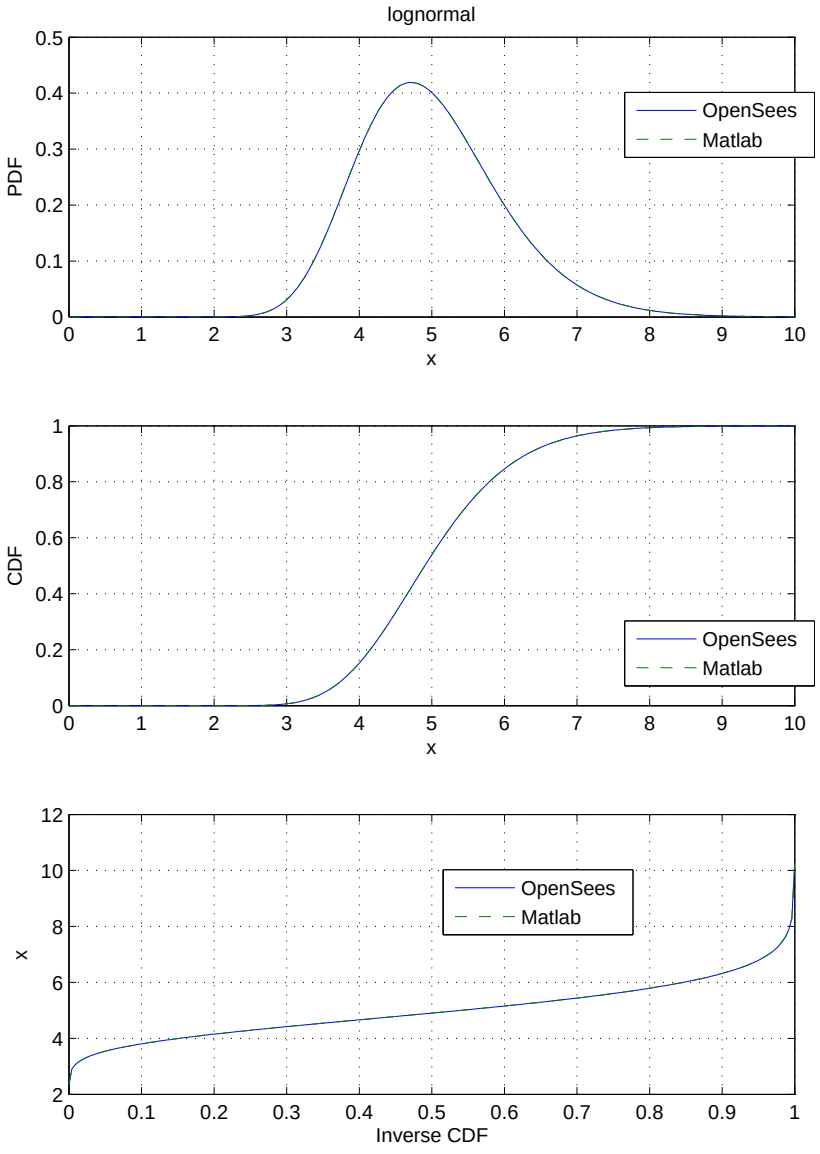

Figure A.7: Lognormal distribution with parameters LN(λ, ζ).

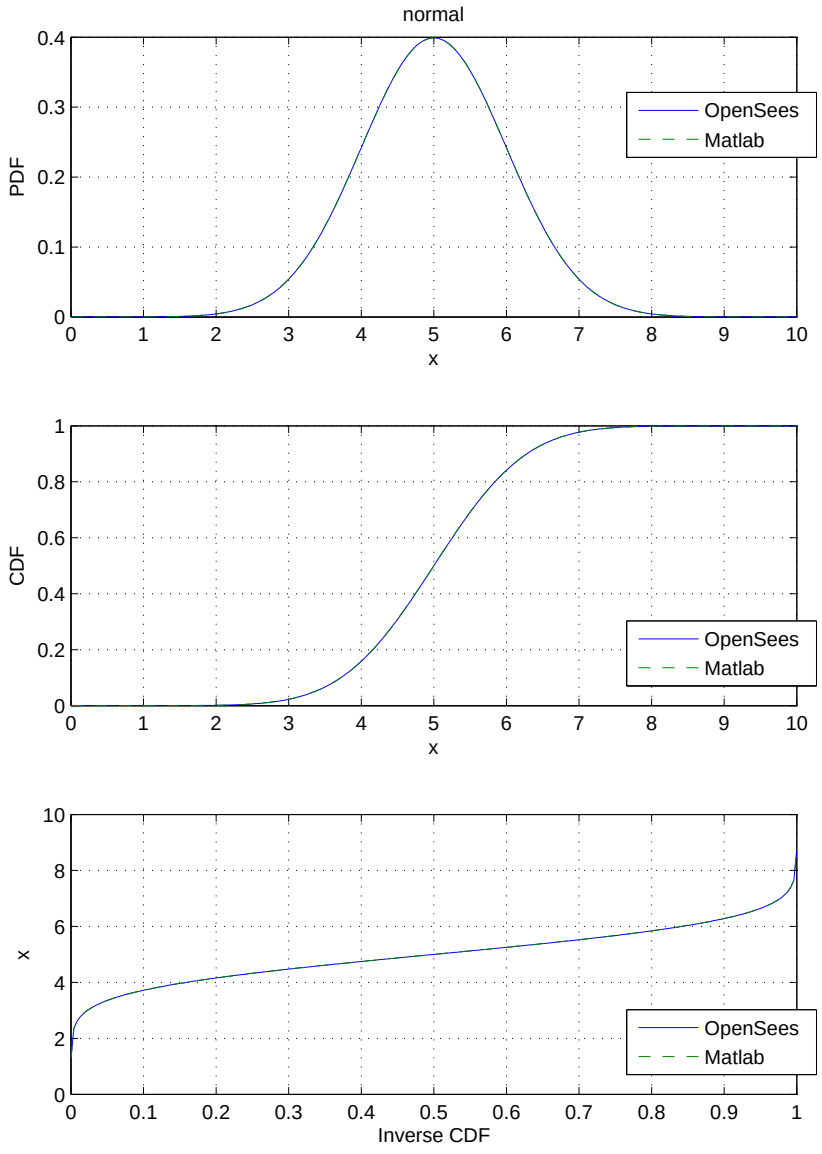

Figure A.8: Normal distribution with parameters $\mathrm{N}(\mu, \sigma)$.

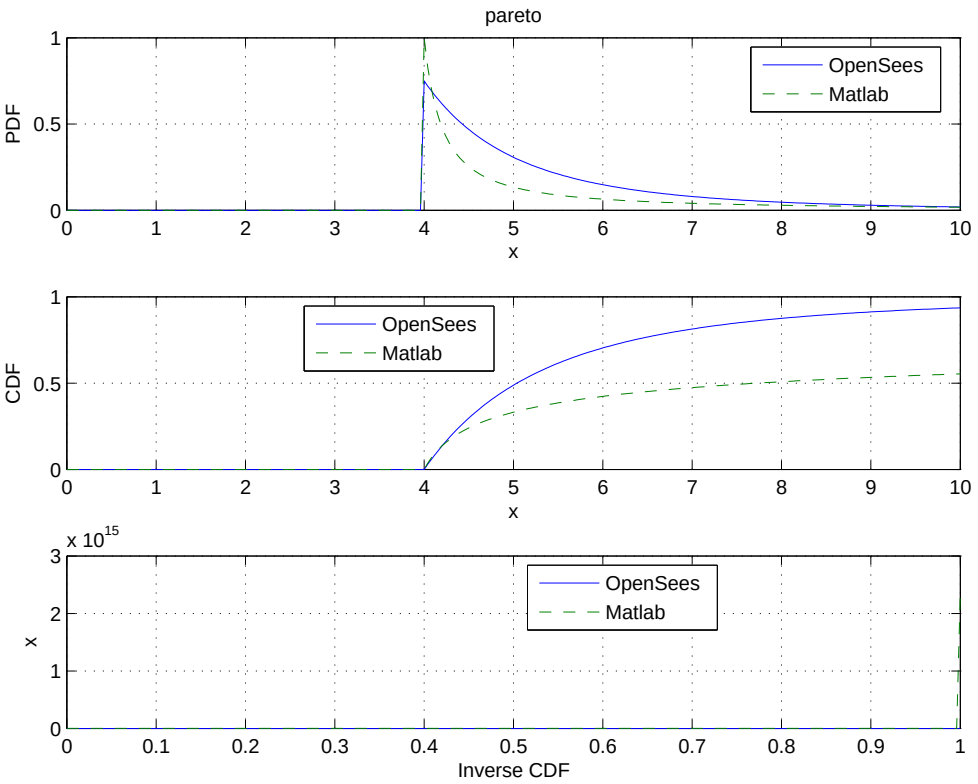

Figure A.9: Pareto distribution with parameters Par(k, u).

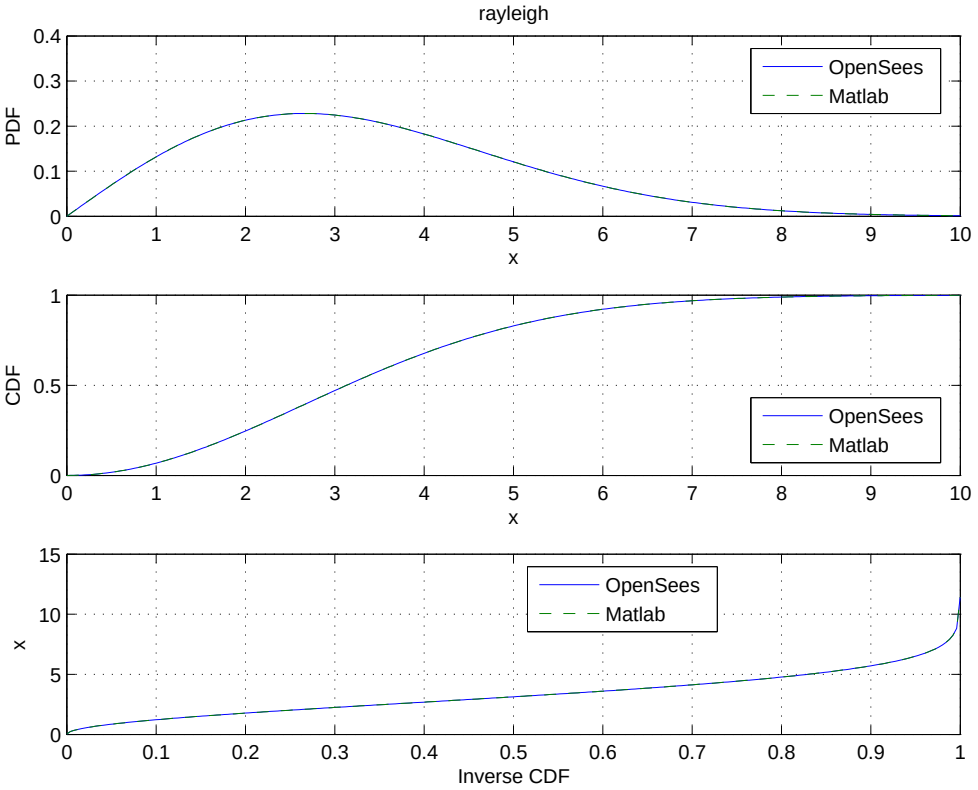

Figure A.10: Rayleigh distribution with parameter Ray(u).

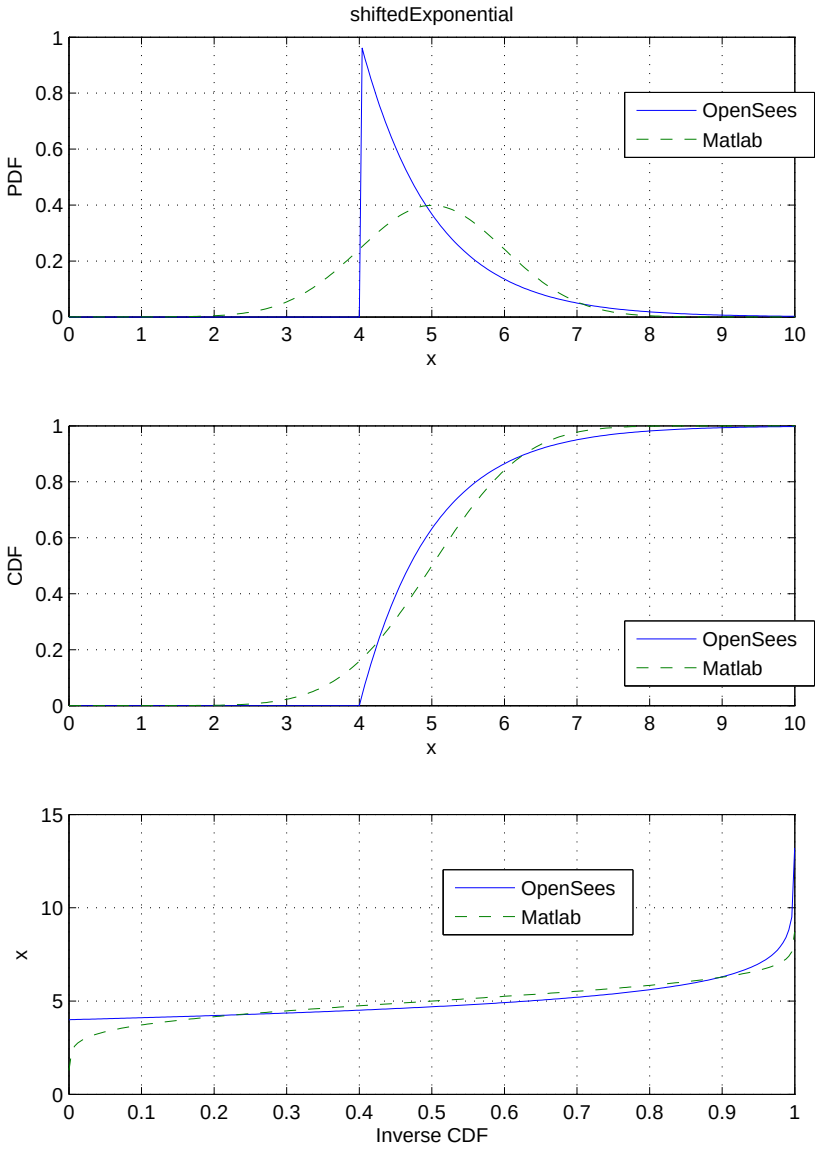

Figure A.11: Shifted Exponential distribution with parameters SE(fill in).

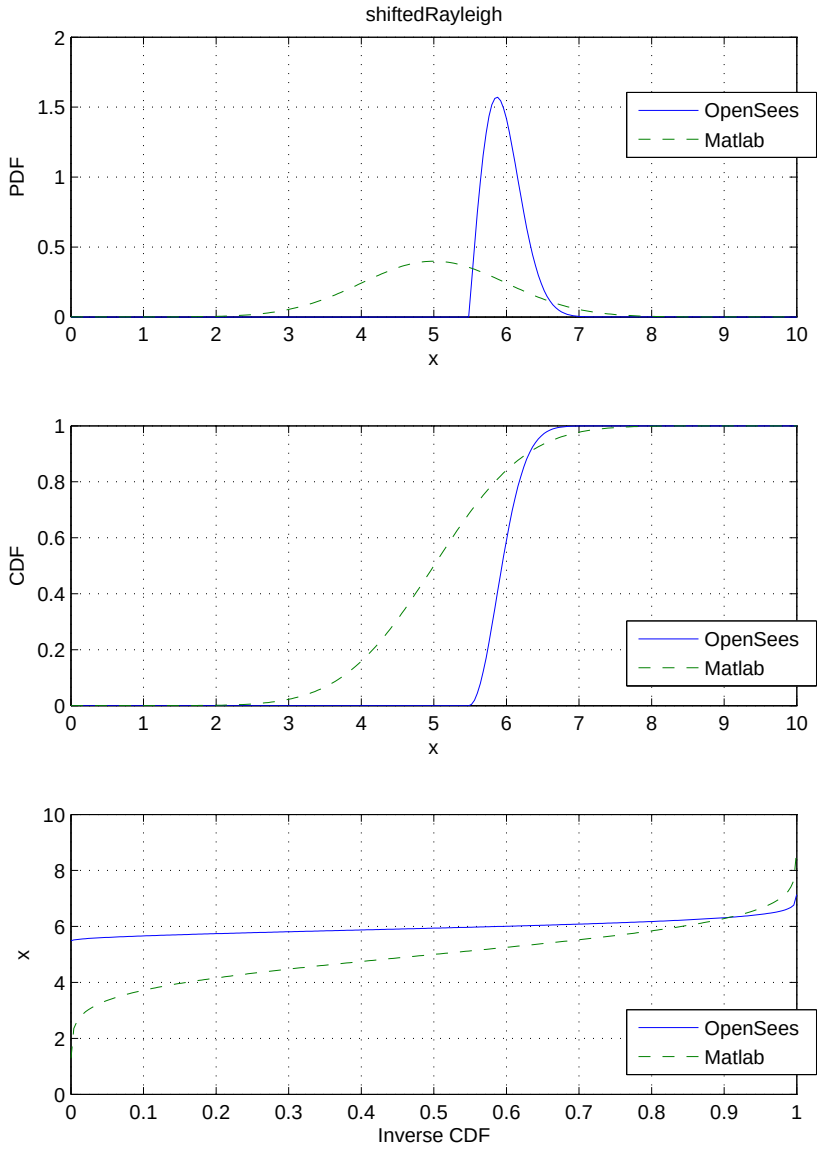

Figure A.12: Shifted Rayleigh distribution with parameters SR(fill in).

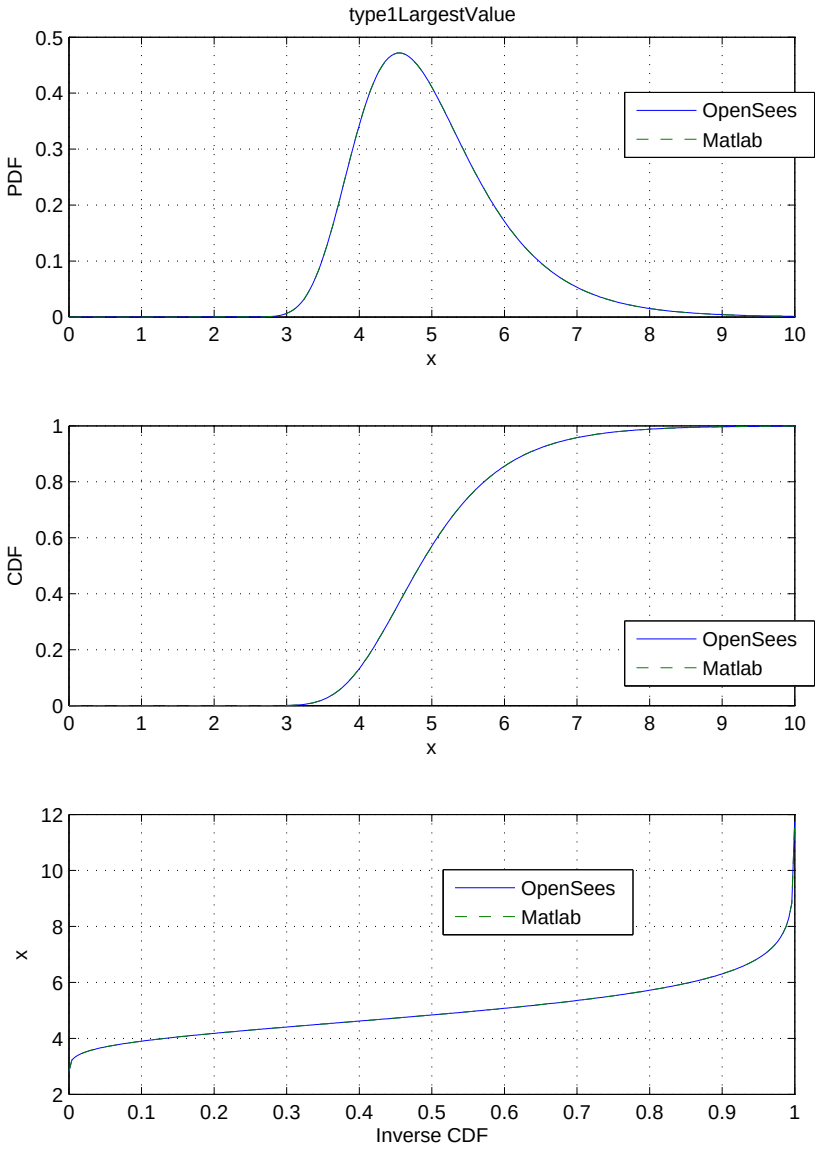

Figure A.13: Type 1 Largest distribution with parameters T1L(u, α).

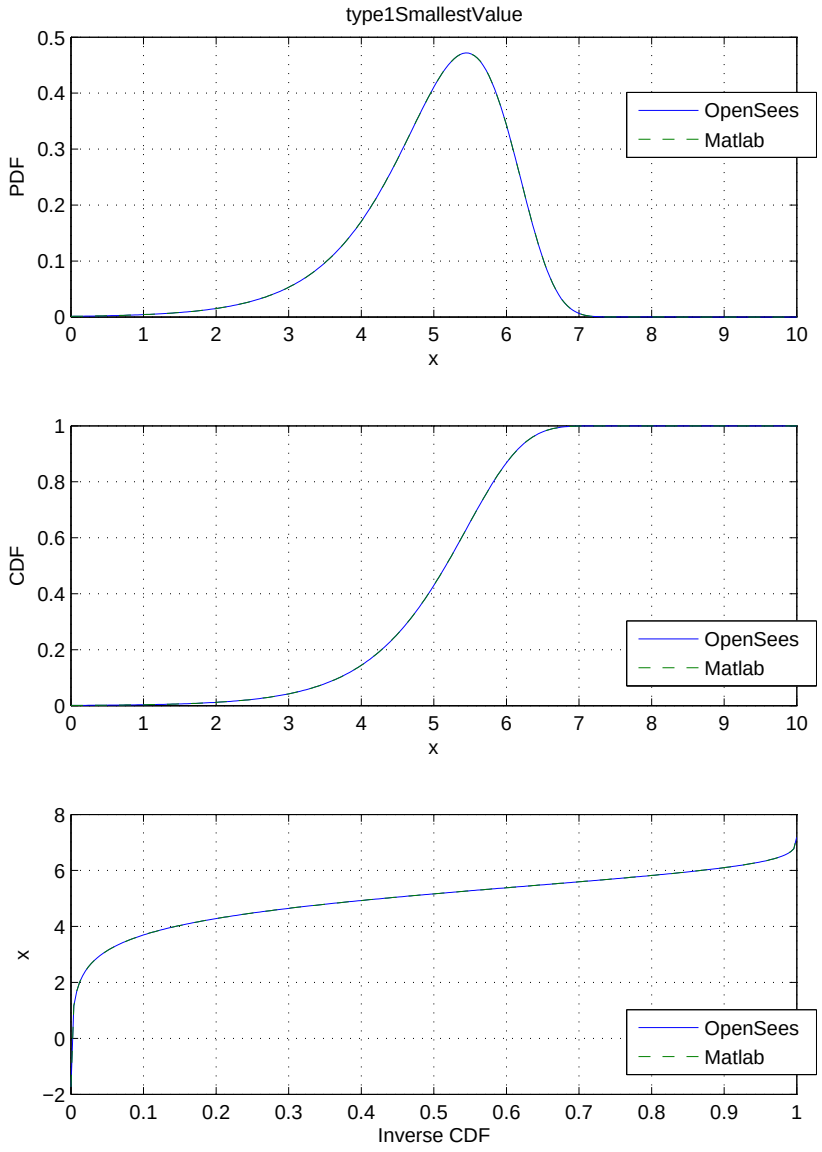

Figure A.14: Type 1 Smallest distribution with parameters T1S(u, α).

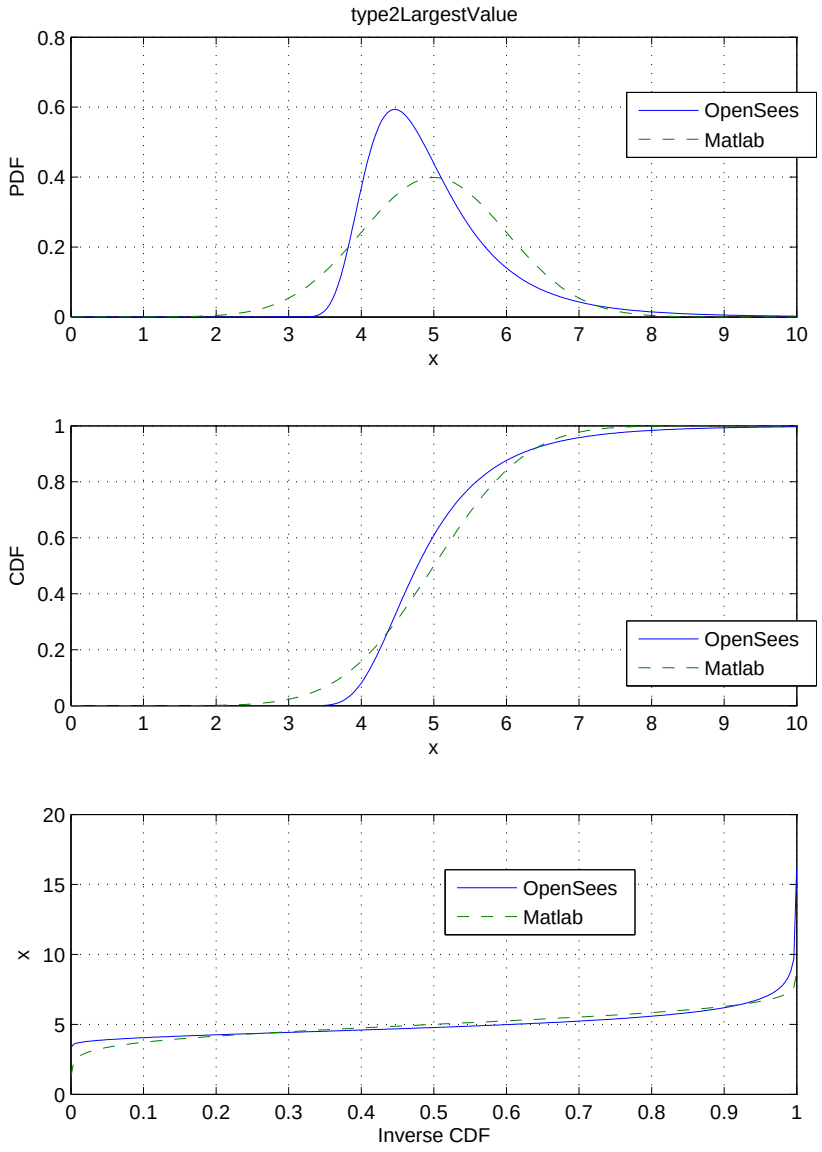

Figure A.15: Type 2 Largest distribution with parameters T2L(u, k).

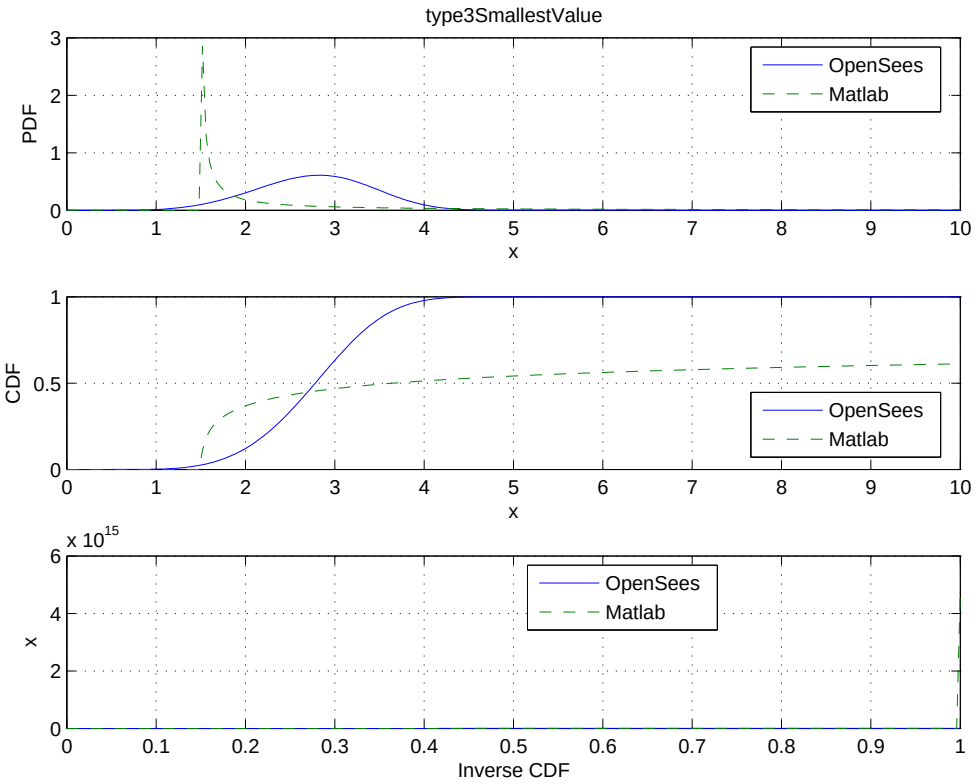

Figure A.16: Type 3 Smallest distribution with parameters T3S(ϵ, u, k).

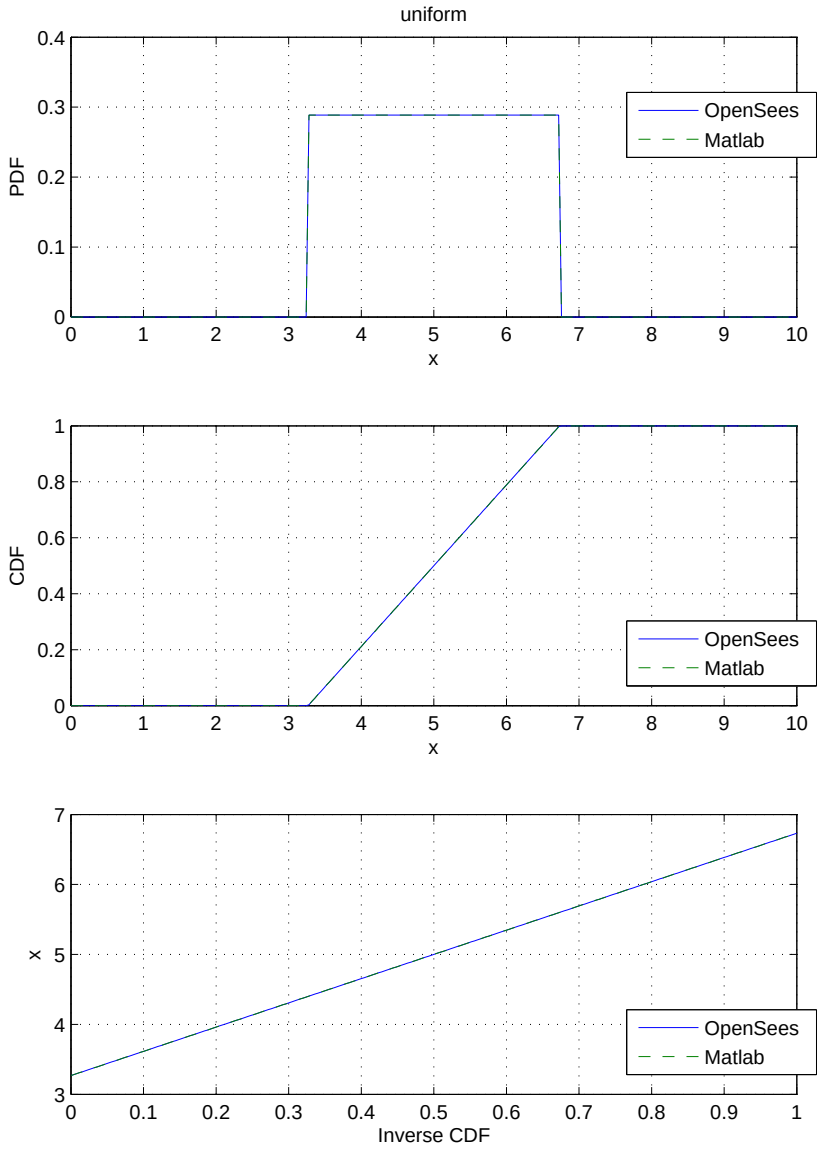

Figure A.17: Uniform distribution with parameters U(a, b).

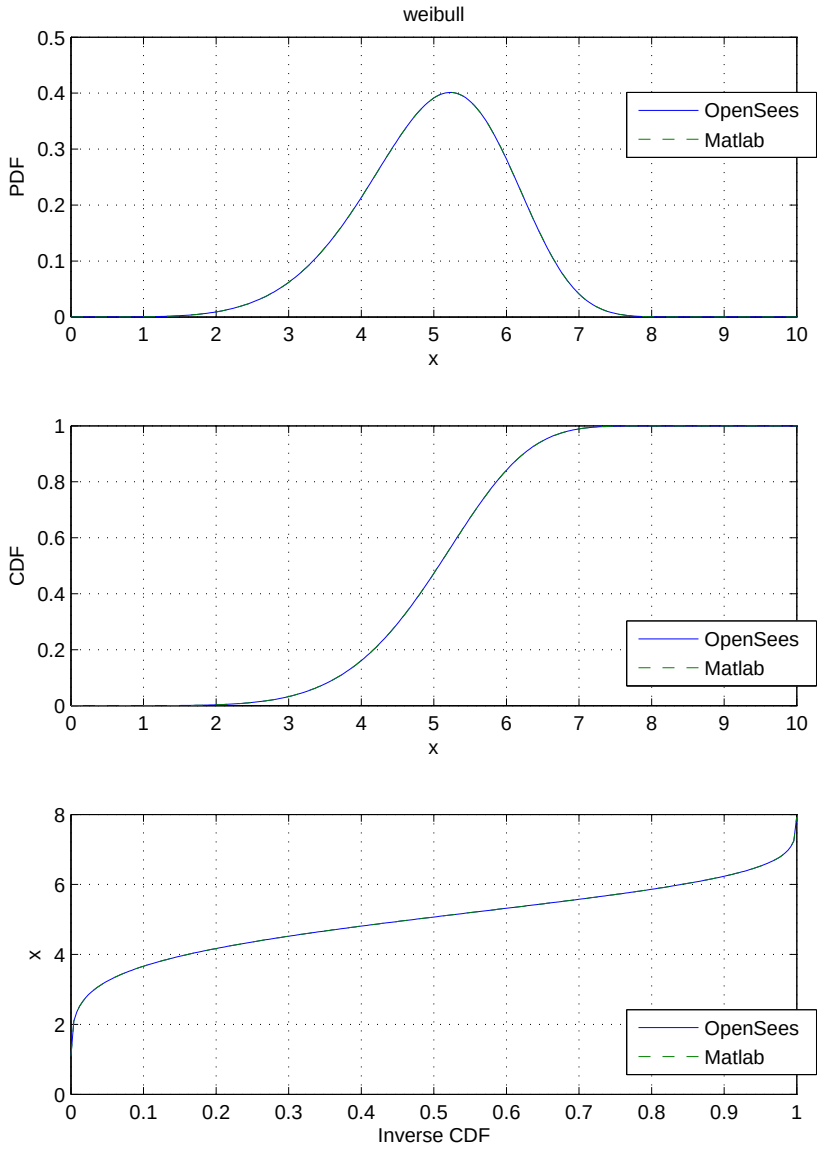

Figure A.18: Weibull distribution with parameters Wbl(u, k).

Appendix B

Random Variable Parameter Sensitivities

B.1 Gumbel Distribution

$$\frac{\partial F}{\partial u} = -f(x) \tag{B.1}$$

$$\frac{\partial F}{\partial \alpha} = -\frac{u-x}{\alpha} f(x) \tag{B.2}$$

$$\frac{\partial u}{\partial \mu} = 1 \tag{B.3}$$

$$\frac{\partial \alpha}{\partial \mu} = 0 \tag{B.4}$$

$$\frac{\partial u}{\partial \sigma} = -\frac{\sqrt{6}\gamma}{\pi} \tag{B.5}$$

$$\frac{\partial \alpha}{\partial \sigma} = -\frac{\pi}{\sqrt{6}\sigma^2} \tag{B.6}$$

B.2 Lognormal Distribution

$$\frac{\partial F}{\partial \lambda} = -xf(x) \tag{B.7}$$

$$\frac{\partial F}{\partial \zeta} = -\frac{(\ln x - \lambda)}{\zeta} xf(x) \tag{B.8}$$

$$\frac{\partial \lambda}{\partial \mu} = \frac{\mu^2 + 2\sigma^2}{\mu(\mu^2 + \sigma^2)} \tag{B.9}$$

$$\frac{\partial \zeta}{\partial \mu} = -\frac{\sigma^2}{\mu\zeta(\mu^2+\sigma^2)} \tag{B.10}$$

$$\frac{\partial \lambda}{\partial \sigma} = -\frac{\sigma}{\mu^2+\sigma^2} \tag{B.11}$$

$$\frac{\partial \zeta}{\partial \sigma} = \frac{\sigma}{\zeta(\mu^2+\sigma^2)} \tag{B.12}$$

B.3 Normal Distribution

$$\frac{\partial F}{\partial \mu} = -f(x) \tag{B.13}$$

$$\frac{\partial F}{\partial \sigma} = -\frac{x-\mu}{\sigma} f(x) \tag{B.14}$$

$$\frac{\partial \mu}{\partial \mu} = 1 \tag{B.15}$$

$$\frac{\partial \sigma}{\partial \mu} = 0 \tag{B.16}$$

$$\frac{\partial \mu}{\partial \sigma} = 0 \tag{B.17}$$

$$\frac{\partial \sigma}{\partial \sigma} = 1 \tag{B.18}$$

B.4 Uniform Distribution

$$\frac{\partial F}{\partial a} = -\frac{1}{b-a} + \frac{x-a}{(b-a)^2} \tag{B.19}$$

$$\frac{\partial F}{\partial b} = -\frac{x-a}{(b-a)^2} \tag{B.20}$$

$$\frac{\partial a}{\partial \mu} = 1 \tag{B.21}$$

$$\frac{\partial b}{\partial \mu} = 1 \tag{B.22}$$

$$\frac{\partial a}{\partial \sigma} = -\sqrt{3} \tag{B.23}$$

$$\frac{\partial b}{\partial \sigma} = \sqrt{3} \tag{B.24}$$