

```
unit SafeWaterUncertBucket;
```

```
//Note: To use this for output vars in SafeWater, we must extend the trial concept  
to a weighted trial - not to bad
```

```
interface
```

```
uses SysUtils, Classes, Variants,  
    Math, Contnrs, LCRGlobals, SafewaterCalcDist;
```

```
const
```

```
    MaxAttempts    = 100;    {Max tries to get lower<trial value<upper}  
    MinusInfinity  = -1e200;  
    Infinity       = 1e200;
```

```
type
```

```
// pextended = ^extended;  
// pvariant = ^variant;
```

```
TAllPercentiles = array[0..100] of single;
```

```
TDescriptiveStatistics = record
```

```
    N                      : Longint;  
    Mean, StdDev, Skewness, Kurtosis, P1, P99 : single;  
    Quantiles              : TAllPercentiles;
```

```
end;
```

```
TDistributed = class
```

```
    fValue    : double;  
    DistType  : word;    {Type of distribution (see above)}  
    IsExtended: boolean; {Determines whether variable loc is extended or
```

```
variant}
```

```
    OwnVar    : boolean; {determines whether the variable is accessed here in  
object}
```

```
    Value     : pDouble;  {pointer to distributed variable}
```

```
    VarValue  : pvariant; {pointer to distributed variable if stored as a  
variant}
```

```
    PTiles    : ppercentiles;
```

```
    CVM       : double;    {goodness-of-fit stat}
```

```
    //Dists    : TDistFits; {results of all distribution fit tests}
```

```
    OrigValue,           {User entered default (mean) value}
```

```
    Lower,               {Lower bound of selection}
```

```
    Upper,               {Upper bound of selection}
```

```
    Censor,              {Value at which to censor distribution}
```

```
    Parm1, Parm2, Parm3  {Distribution specific Parameters}
```

```
    : double;
```

```
    CustomPoints : TVec2; {Points for custome distribution}
```

```
    Points       : TObjectList; {Data points - input or output}
```

```
    Stats        : TDescriptiveStatistics; {Record containing Desc. Statistics}
```

```
    Name         : string[100]; {User supplied variable name}
```

```
    VCFile       : string[200]; {file containing sets of percentils for
```

VariableCustom}

```
    constructor Create(nV : pDouble; nV : pvariant; nType : word;
nOrig,nMin,nMax,nC,nP1,nP2,nP3 : double; S : string; CustomList : TStringList;
VUFile : string=''; pP : PPercentiles=nil);
    destructor destroy; override;
    procedure Toss;
    procedure StorePoint;
    procedure GenerateStats;
```

```
    procedure LoadFromStream(AStream : TStream);
    procedure SaveToStream(AStream : TStream);
end;
```

```
TTrial=class(TObject)
    R : double;
    constructor Create(nR : double);
end;
```

```
TUncertaintyStudy=class(TObject)
    InVars,           {Collection of input variables,distributions and points}
    OutVars      : TObjectList; {Collection of output variables,distributions, and
points}
    RandomStream : TObjectList; {single random number generated for each trial}
    NumTrials    : integer;
    TotalTrials : integer;
    StoreInputs : boolean;
    constructor Create(TT : integer);
    destructor destroy; override;
    function AddInVar(nV : pDouble; nV : pVariant; nType : integer;
nOrig,nMin,nMax,nC,nP1,nP2,nP3 : double; S : string; CustomList : TStringList;
VUFile : string=''; pP : PPercentiles=nil) : TDistributed;
    function AddOutVar(nV : pDouble; S : string) : TDistributed;
    procedure ClearAll;
    procedure ResetValues;
    procedure NewTrial;
    procedure StoreInputTrial;
    procedure StoreOutputTrial;
    procedure StoreTrial;
    //procedure GenerateInputStream(Latin : boolean);
    procedure GenerateRandomStream;
    procedure ExpandOutputStream;
    //procedure FetchInputTrial(Trial : integer);
    function GetRandomFromStream(Trial : integer) : double;
    procedure SumSingleOutputTrial(T : TDistributed; Trial : integer);
    procedure SumOutputTrial(Trial : integer);
    procedure GenerateStats;
    procedure DumbPrint;
    procedure DumbPrintToStrings(T : TStrings);
```

```

procedure DumbPrintToCSVStrings(T : TStrings);
procedure ResetRun;
procedure FindOutputDist(FindDist : Boolean);
function FindInVar(Name : string; var ix : integer) : boolean;
function FindInVarPointer(Name : string) : pDouble;

procedure ReportToStrings(T : TStrings);
//function ReportInputsToHTML(Extra : string) : string;

procedure LoadFromStream(AStream : TStream);
procedure SaveToStream(AStream : TStream);
end;

```

```

function icdf(dtype:integer;y,parm1,parm2,parm3:double):double;
function cdf(dtype:integer;x,parm1,parm2,parm3:double):double;

```

```

var CountExceptions : word;

```

```

implementation

```

```

function RandComp(Data1, Data2 : pointer) : integer;
begin
  //Radomizer...
  Result:=-1+Random(3);
end;

```

```

//to replace MtxVec that returns NON when A=B;
function icdfUniform(Prob,A,B : double) : double;
begin
  icdfUniform:=Prob*(b-a)+a;
end;

```

```

function icdfTriangular(y,A,B,ML:double): single;
var height : double;
    leftarea, rightarea: double;
begin
  icdftriangular:=0;

  // following line added 8/30/17
  if b=a then icdftriangular := A;

  if b=a then exit;
  height:=2/(B-A);
  leftarea:=(ML-A)*height/2;
  rightarea:=(B-ML)*height/2;
  if y<(leftarea/(rightarea+leftarea)) then {x<ml}
    icdftriangular := a+sqrt(2*y/(height/(ML-A))) else
    icdftriangular := b-sqrt((2*y-2)/(height/(ML-B)));

```

```

end;

function cdfTriangular(X,A,B,ML : double) : single;
var height : double;
begin
    cdftriangular:=0;
    if b=a then exit;
    if x<A then exit;
    if x>B then begin cdftriangular:=1; exit; end;

    height:=2/(B-A);
    if x<ML then cdfTriangular:=0.5*(X-A)*(Height/(ML-A))*(X-A)
        else cdfTriangular:=1-0.5*(B-X)*(Height/(ML-B))*(X-B);
end;

function ICDFWeibull( Prob, A, B: Single ): Single;
begin
    Result := A * Power( -Ln(1-Prob), 1/B );
end;

function rTriangular(Minimum, Maximum, MostLikely : double) : double;
var temp, base, part: double;
begin
    base := Maximum - Minimum;
    part := MostLikely - Minimum;
    Temp:=Random;
    if (Temp < (part/base)) then
        Temp := Minimum + Sqrt(base*part*Temp)
    else
        Temp := Maximum - Sqrt(base*(Maximum-Mostlikely)*(1-Temp));
    rTriangular := Temp;
end;

function icdf(dtype:integer;y,parm1,parm2,parm3:double):double;
var t : double;
begin
    case Dtype of
        //TODO Needs Testing      dGamma      : icdf:=parm1+GammaCDFInv(y,Parm2,parm3);
        dBeta      : icdf:=BetaCDFInv(y,parm1,Parm2);
        dNormal     : icdf:=NormalCDFInv(y,parm1,Parm2);
        dLogNormal  : icdf:=LogNormalCDFInv(y,parm1,Parm2);
        //Triangular must be take from old CalcDist or somewewhere
        dTriangular : icdf:=icdfTriangular(y,Parm1,Parm2,Parm3);
        //TODO Needs Testing      dPoisson     : icdf:=PoissonCDFInv(y,parm1);
        dBinomial    : icdf:=BinomCDFInv(y,round(parm2),Parm1);
        //OLD MtVec      dUniform      : icdf:=UniformCDFInv(y,parm1,Parm2);
        dUniform     : icdf:=icdfUniform(y,parm1,Parm2);
        //TODO Needs Testing      dExponential : icdf:=ExpCDFInv(y,parm1);
        //TODO Needs Testing      dGeometric   : icdf:=GeometricCDFInv(y,parm1);
        dWeibull     : icdf:=ICDFWeibull(y,parm1,Parm2);
    end;
end;

```

```

//Logistic must be take from old CalcDist or somewhere
//      dLogistic      : icdf:=LogisticCDFInv(y,parm1,Parm2);
//TODO Needs Testing      dCauchy      : icdf:=CauchyCDFInv(y,parm1,Parm2);
//TODO Needs Testing      dPareto      : icdf:=ParetoCDFInv(y,parm1,Parm2);
//HML must be take from old CalcDist or somewhere
//      dHML           : icdf:=HMLCDFInv(y,parm1,Parm2, Parm3);
//      dStudentT       : begin
//                          t:=StudentCDFInv(y,trunc(parm3));
//                          //icdf:=(t*(parm2/sqrt(parm3)))+parm1;
//                          icdf:=(t*parm2)+parm1;
//                      end;
//      dnone           : icdf:=parm1;
//      end; {case}
end; {ICDF}

```

```

function cdf(dtype:integer;x,parm1,parm2,parm3:double):double;
var t : double;
begin
    Case Dtype of
//TODO Needs Testing      dGamma      : cdf:=GammaCDF(x,parm1,parm2,parm3);
//      dBeta          : cdf:=BetaCDF(x,parm1,Parm2);
//      dNormal        : cdf:=NormalCDF(x,parm1,Parm2);
//      dLogNormal     : cdf:=LogNormalCDF(x,parm1,Parm2);
//Triangular must be take from old CalcDist or somewhere
//      dTriangular    : cdf:=cdfTriangular(x,parm1,Parm2,Parm3);
//TODO Needs Testing      dPoisson     : cdf:=PoissonCDF(x,parm1);
//      dBinomial       : cdf:=BinomCDF(round(x),round(Parm2),Parm1);
//      dUniform        : cdf:=UniformCDF(x,parm1,Parm2);
//TODO Needs Testing      dExponential : cdf:=ExpCDF(x,parm1);
//TODO Needs Testing      dGeometric   : cdf:=GeometricCDF(x,parm1);
//      dWeibull        : cdf:=WeibullCDF(x,parm1,Parm2);
//Logisitic must be take from old CalcDist or somewhere
//      dLogistic      : cdf:=LogisticCDF(x,parm1,Parm2);
//TODO Needs Testing      dCauchy      : cdf:=CauchyCDF(x,parm1,Parm2);
//TODO Needs Testing      dPareto      : cdf:=ParetoCDF(x,parm1,Parm2);
//HML must be take from old CalcDist or somewhere
//      dHML           : cdf:=HMLcdf(x,Parm1,Parm2,Parm3);
//      dStudentT       : begin
//                          t:=(x-parm1)/(parm2/sqrt(parm3));
//                          cdf:=StudentCDF(t,trunc(parm3));
//                      end;
//
//      dnone           : cdf:=parm1;
//      end; {case}
end; {cdf}

```

```

function RandD(dtype:integer;parm1,parm2,parm3:double):double;
var y : double;
begin
    y:=random;

```

```

    Result:=icdf(dtype,y,parm1,parm2,parm3);
end;

constructor TTrial.Create(nR : double);
begin
    R:=nR;
end;

constructor TDistributed.Create(nV : pDouble; nVV : pVariant; nType : word;
nOrig,nMin,nMax,nC,nP1,nP2,nP3 : double; S : string; CustomList : TStringList;
VUFile : string=''; pP : PPercentiles=nil);
var i : integer;
begin
    inherited create;
    DistType:=nType;
    Parm1:=nP1;      Parm2:=nP2;      Parm3:=nP3;
    OrigValue:=nOrig; Lower:=nMin;  Upper:=nMax;
    Value:=nV;
    VarValue:=nVV;

    if DistType=dVariableCustom then begin
        VCFile:=VUFile;
        PTiles:=pP;
    end else
        //if both addresses are nil, use the local var (extended only)
        if (Value=nil) and (VarValue=nil) then
            Value:=@fValue;

    Name:=s;
    Censor:=nC;
    if (Lower=Upper) then begin
        Lower:=MinusInfinity;
        Upper:=Infinity;
    end;
    //fillchar(Dists,SizeOf(Dists),0);
    Points:=TObjectList.create(true);
    {Points.Capacity:=1000;}
    IsExtended:=not (Value=nil);

    //this is set so icdf functions works for no dist;
    if DistType=dNone then
        Parm1:=OrigValue;

    if DistType=dCustom then begin
        raise exception.Create('No custom dist allowed - refactor UncertBucket');
        (*
        CustomPoints:=TVec.Create;
        if CustomList=nil then begin
            CustomPoints.Size(1);
            CustomPoints[0]:=nOrig;

```

```

end else begin
    if CustomList.Count=0 then begin
        CustomPoints.Size(1);
        CustomPoints[0]:=nOrig;
    end else begin
        CustomPoints.Size(CustomList.Count);
        for i:=0 to CustomList.Count-1 do begin
            CustomPoints[i]:=0;
            try
                CustomPoints[i]:=strtofloat(CustomList[i]);
            except
                //TODO add exception.  Values are being set to zero.
            end;
        end;
    end;
end;
CustomPoints.SortAscend();
*)
end;

if DistType=dICustom then begin
    raise exception.Create('No Icustom dist allowed - refactor UncertBucket');
    (*
    CustomPoints:=TVec.Create;
    if CustomList=nil then begin
        CustomPoints.Size(1);
        CustomPoints[0]:=1;
    end else begin
        if CustomList.Count=0 then begin
            CustomPoints.Size(1);
            CustomPoints[0]:=1;
        end else begin
            CustomPoints.Size(CustomList.Count);
            for i:=0 to CustomList.Count-1 do begin
                CustomPoints[i]:=0;
                try
                    CustomPoints[i]:=strtofloat(CustomList[i]);
                except
                    //TODO add exception.  Values are being set to zero.
                end;
            end;
        end;
    end;
    *)
end;

end;

procedure TDistributed.Toss;
var T : double;

```

```

N : integer;

function GotIt : boolean;
var bot,top,randnum: double;
    idx : integer;
begin
    inc(N);

    if DistType=dCustom then begin
        raise exception.Create('No custom dist allowed - refactor UncertBucket');
        (*
        idx:=Random(CustomPoints.Length+1);
        T:=CustomPoints.Values[idx];
        GotIt:=True;
        exit;
        *)
    end;

    if T<Censor then begin
        T:=Censor;
        GotIt:=True;
    end else begin
        if N>MaxAttempts then begin
            top:=cdf(disttype,upper,param1,param2,param3);
            bot:=cdf(disttype,lower,param1,param2,param3);
            randnum:=bot+((top-bot)*random);
            T:=icdf(disttype,randnum,param1,param2,param3);
            GotIt:=True;
        end else
            Gotit:=(Lower<T) and (T<Upper);
        end;
    end;

begin
    N:=0;
    try
        if DistType=dVariableCustom then begin
            TSafeWaterPercUncFile.GetRandom(VCFile,PTiles^);
        end else
            repeat
                T:=RandD(DistType,Param1,Param2,Param3);
            until GotIt;
    except on EMathError do begin
        CountExceptions:=CountExceptions+1;
        N:=MaxAttempts+1;
    end;
end;

// if N<MaxAttempts then move(T,Value^,sizeof(Value^));
if IsExtended then

```



```

    if N<MaxAttempts then Value^:=T
  else
    if N<MaxAttempts then VarValue^:=T;
end;

procedure TDistributed.StorePoint;
var P : TTrial;
    v : double;
begin
  if IsExtended then
    v:=Value^
  else
    try
      v:=StrToFloat(VarToStr(VarValue^));
    except
      //TODO there must be a better way to do this...
    end;

    P:=TTrial.Create(v);
    Points.Add(P);
end;
destructor TDistributed.destroy;
begin
  Points.Free;
  inherited destroy;
end;
procedure TDistributed.GenerateStats;
var T : TVec;
    i : integer;
    interval : double;
begin
  createit(T);
  T.Size(Points.Count);
  for i:=0 to Points.Count-1 do begin
    T.Values[i]:=TTrial(Points.Items[i]).R;
  end;
  T.SortAscend;
  Stats.N:=T.Length;
  Stats.Mean:=T.Mean;
  Stats.StdDev:=T.StdDev(Stats.Mean);
  Stats.Skewness:=T.Skewness(Stats.Mean,Stats.StdDev);
  Stats.Kurtosis:=T.Kurtosis(Stats.Mean,Stats.StdDev);
  Stats.P1:=Percentile(T,0.01);
  Stats.P99:=Percentile(T,0.99);
  interval:=(100/(high(Stats.Quantiles)-low(Stats.Quantiles)+1))/100;
  for i:=low(Stats.Quantiles) to high(Stats.Quantiles) do begin
    Stats.Quantiles[i]:=Percentile(T,i*interval+interval/2);
  end;
  freeit(T);
end;
end;

```

```

procedure TDistributed.LoadFromStream(AStream: TStream);
begin

end;

procedure TDistributed.SaveToStream(AStream: TStream);
begin
end;

constructor TUncertaintyStudy.Create;
begin
    InVars :=TObjectList.Create(true);
    OutVars:=TObjectList.Create(true);
    RandomStream:=TObjectList.Create(true);
    NumTrials:=0;
    TotalTrials:=TT;
    StoreInputs:=False;
end;
procedure TUncertaintyStudy.ClearAll;
begin
    InVars.Free;
    OutVars.Free;
    RandomStream.Free;
end;
destructor TUncertaintyStudy.destroy;
begin
    ClearAll;
    inherited destroy;
end;

function TUncertaintyStudy.FindInVar(Name : string; var ix : integer) : boolean;
var i : integer;
begin
    ix:=-1;
    Result:=False;
    for i:=0 to InVars.Count-1 do begin
        if CompareText(TDistributed(Invars.Items[i]).Name,Name)=0 then begin
            ix:=i;
            Result:=True;
            break;
        end;
    end;
end;

function TUncertaintyStudy.AddInVar(nV : pDouble; nVV : pVariant; nType : integer;
nOrig,nMin,nMax,nC,nP1,nP2,nP3 : double; S : string; CustomList : TStringList;
VUFile : string=''; pP : PPercentiles=nil) : TDistributed;
var P : TDistributed;

```

```

    i : integer;
begin
    if FindInVar(S,i) then begin
        P:=TDistributed(InVars.Items[i]);
    end else begin
        P:=TDistributed.Create(nV,nVV, nType,nOrig,NMin,NMax,nC,nP1,nP2,nP3,S,
CustomList, VUFile,pP);
        InVars.Add(P);
    end;
    Result:=P;
end;
function TUncertaintyStudy.AddOutVar(nV : pDouble; S : string) : TDistributed;
var P : TDistributed;
begin
    NV^:=0;
    P:=TDistributed.Create(nV,nil,dNone,0,0,0,0,0,0,0,S,nil);
    OutVars.Add(P);
    Result:=P;
end;

procedure TUncertaintyStudy.ResetValues;
var i: integer;
    T: TDistributed;
begin
    for i:=0 to InVars.count-1 do begin
        T:=TDistributed(InVars.items[i]);
        if T.DistType=dVariableCustom then begin
            T.Value^:=TSafeWaterPercUncFile.GetMean(T.VCFile,T.PTiles^);
        end else
            if T.IsExtended then
                T.Value^:=T.OrigValue
            else
                T.VarValue^:=T.OrigValue;
        end;
    end;
end;

procedure TUncertaintyStudy.NewTrial;
var i: integer;
begin
    for i:=0 to InVars.count-1 do TDistributed(InVars.items[i]).Toss;
    NumTrials:=NumTrials+1;
end;

procedure TUncertaintyStudy.StoreInputTrial;
var i: integer;
begin
    for i:=0 to InVars.Count-1 do TDistributed(InVars.items[i]).StorePoint;
end;
procedure TUncertaintyStudy.StoreOutputTrial;
var i: integer;

```

```

begin
  for i:=0 to OutVars.Count-1 do TDistributed(OutVars.items[i]).StorePoint;
end;
procedure TUncertaintyStudy.StoreTrial;
begin
  if StoreInputs then StoreInputTrial;
  StoreOutputTrial;
end;
{
procedure TUncertaintyStudy.GenerateInputStream(Latin : boolean);
var i,j,n: integer;
    pct : double;
    TL   : TTrial;
    T     : TDistributed;
    LatinSeries : TObjectList;

    v,PctStart,PctEnd : double;

function GotIt : boolean;
begin
  inc(n);
  if n=100000 then
    raise exception.Create('cannot generate random value in
GenerateInputStream');

    if TL.R<T.Censor then begin
      TL.R:=T.Censor;
      Result:=True;
      exit;
    end;
    Result:=(T.Lower<=TL.R) and (TL.R<=T.Upper);
  end;

begin
  CountExceptions:=0;
  if TotalTrials=0 then exit;
  if not Latin then
    for j:=1 to TotalTrials do begin
      for i:=0 to InVars.count-1 do TDistributed(InVars.items[i]).Toss;
      StoreInputTrial;
    end
  else begin
    pct:=1/TotalTrials;
    LatinSeries:=TObjectList.Create(TRUE);
    for i:=0 to InVars.count-1 do begin
      T:=TDistributed(InVars.items[i]);
      //handled specially...
      if T.DistType=dVariableCustom then continue;
      for j:=1 to TotalTrials do begin

```

```

    TL:=TTrial.Create(0);
    if T.DistType=dCustom then begin
        TL.R:=Percentile(T.CustomPoints,(J*Pct)-Pct/2);
    end else begin
        //find if distribution is truncated
        v:=cdf(T.DistType,T.Lower,T.Parm1,T.Parm2,T.Parm3);
        if v>0 then PctStart:=v else PctStart:=0;
        v:=cdf(T.DistType,T.Upper,T.Parm1,T.Parm2,T.Parm3);
        if v>0 then PctEnd:=v else PctEnd:=1;

        n:=0;
        TL.R:=icdf(T.DistType,PctStart + (((J*Pct)-Pct/2) *
(PctEnd-PctStart)),T.Parm1,T.Parm2,T.Parm3);
        if not GotIt then
            raise exception.Create('cannot generate random value in
GenerateInputStream - Check Upper and Lower bounds for '+T.Name);

        end;
        LatinSeries.Add(TL);
    end;
    LatinSeries.Sort(RandComp);
    for j:=1 to TotalTrials do begin
        TL:=TTrial(LatinSeries.Items[j-1]);
        if T.IsExtended then
            T.Value^:=TL.R
        else
            T.VarValue^:=TL.R;
        T.StorePoint;
    end;
    LatinSeries.Clear;
end;
TObject(LatinSeries).Free;
end;
NumTrials:=TotalTrials;
end;
}

```

```

procedure TUncertaintyStudy.GenerateRandomStream;
var i: integer;
    T:TTrial;
    R : double;
begin
    CountExceptions:=0;
    for i:=0 to TotalTrials do begin
        R:=Random;
        T:=TTrial.Create(R);
        RandomStream.Add(T);
    end;
end;

```

```

procedure TUncertaintyStudy.ExpandOutputStream;
var j: integer;
begin
    for j:=1 to TotalTrials do StoreOutputTrial;
end;

{
procedure TUncertaintyStudy.FetchInputTrial;
var i: integer;
    T: TDistributed;
begin
    for i:=0 to Invars.count-1 do begin
        T:=TDistributed(InVars.items[i]);
        if T.DistType=dVariableCustom then begin
            T.Value^:=TSafeWaterPercUncFile.GetIX(T.VCFile,Trial,T.PTiles^);
        end else
            if T.IsExtended then
                T.Value^:=TTrial(T.Points.Items[Trial-1]).R
            else
                T.VarValue^:=TTrial(T.Points.Items[Trial-1]).R;
                //move(TTrial(T.Points[Trial-1]).R,T.Value^,sizeof(T.Value^))
        end;
    end;
end;
}

function TUncertaintyStudy.GetRandomFromStream(Trial : integer) : double;
begin
    GetRandomFromStream:=TTrial(RandomStream.Items[Trial]).R;
end;

procedure TUncertaintyStudy.SumSingleOutputTrial(T : TDistributed; Trial : integer);
begin
    TTrial(T.Points[Trial]).R:=TTrial(T.Points[Trial]).R+T.Value^;
end;

procedure TUncertaintyStudy.SumOutputTrial;
var i: integer;
    T: TDistributed;
begin
    for i:=0 to Outvars.count-1 do begin
        T:=TDistributed(OutVars.items[i]);
        SumSingleOutputTrial(T,Trial);
    end;
end;

procedure TUncertaintyStudy.DumbPrint;
var
    J : Word;
    T : TEXT;
    procedure ShowVars(P:tdistributed) ; far;

```

```

begin
  With P do
    WriteLn(j:8, Value^:10:3, Lower:10:3, Upper:10:3,
      OrigValue:10:3, Parm1:10:3, Parm2:10:3, Parm3:10:3);
    j:=j+1;
  end;
  procedure ShowRes(P:TTrial) ; far;
  begin
    Write(T,P.R:10:3);
  end;
begin end;
(*begin
  j:=0;
  {$IFDEF Windows}
  ScreenSize.Y := 100;
  {$ENDIF}
  assign(T,'d:\caa812\test.prn');
  rewrite(T);
  WriteLn('Number':8, 'Value':10, 'Min':10, 'Max':10, 'Orig':10,
'Parm1':10, 'Parm2':10, 'Parm3':10);
  InVars^.ForEach(@ShowVars);
  for j:=0 to NumTrials-1 do begin
    for i:=0 to InVars^.Count-1 do begin
      p:=PDistributed(InVars^.At(i));
    {
      write(PTrial(P^.Points^.At(j))^R:10:3);}
      write(T,PTrial(P^.Points^.At(j))^R:10:3);
    end;
    for i:=0 to OutVars^.Count-1 do begin
      p:=PDistributed(OutVars^.At(i));
    {
      write(PTrial(P^.Points^.At(j))^R:10:3);}
      write(T,PTrial(P^.Points^.At(j))^R:10:3);
    end;
    writeln(T);
  {
    writeln;
  }
  end;
  close(T);
end;*)

procedure TUncertaintyStudy.ResetRun;
begin
  NumTrials:=0;
  Invars.destroy;
  OutVars.destroy;
end;

procedure TUncertaintyStudy.FindOutputDist(FindDist : Boolean);
begin
end;

procedure TUncertaintyStudy.DumbPrintToStrings(T : TStrings);

```

```

var i,j : integer;
    D : TDistributed;
begin
    for i:=0 to Invars.Count-1 do begin
        D:=TDistributed(Invars[i]);
        D.GenerateStats;
        T.Add(D.Name);
        T.Add('Mean: '+FloatToStr(D.Stats.Mean));
        T.Add('StdDev: '+FloatToStr(D.Stats.StdDev));
        for j:=low(D.Stats.Quantiles) to high(D.Stats.Quantiles) do begin
            T.Add(inttostr(j)+' : '+FloatToStr(D.Stats.Quantiles[j]));
        end;
    end;
end;

procedure TUncertaintyStudy.DumbPrintToCSVStrings(T : TStrings);
var i,j : integer;
    D : TDistributed;
    s : string;
begin
    T.Add('name,mean,stddev,P1,p5,p50,p95,p99');
    for i:=0 to Invars.Count-1 do begin
        D:=TDistributed(Invars[i]);
        D.GenerateStats;
        S:=D.Name+', '+FloatToStr(D.Stats.Mean)+', '+FloatToStr(D.Stats.StdDev)+', '+
            FloatToStr(D.Stats.Quantiles[1])+', '+FloatToStr(D.Stats.Quantiles[5])+', '+
            FloatToStr(D.Stats.Quantiles[50])+', '+FloatToStr(D.Stats.Quantiles[95])+', '+
            FloatToStr(D.Stats.Quantiles[99]);
        T.Add(S);
    end;
end;

procedure TUncertaintyStudy.ReportToStrings(T : TStrings);
var i,j : integer;
    D : TDistributed;
begin
    T.Add('***Uncertainty Object***');
    T.Add('Input Variable Stats');
    for i:=0 to Invars.Count-1 do begin
        D:=TDistributed(Invars[i]);
        D.GenerateStats;
        T.Add(D.Name);
        T.Add('Mean: '+FloatToStr(D.Stats.Mean));
        T.Add('StdDev: '+FloatToStr(D.Stats.StdDev));
        for j:=low(D.Stats.Quantiles) to high(D.Stats.Quantiles) do begin
            T.Add(inttostr(j)+' : '+FloatToStr(D.Stats.Quantiles[j]));
        end;
    end;
end;

```



```

end;

{
function TUncertaintyStudy.ReportInputsToHTML(Extra : string) : string;
var i,j,SFC : integer;
    D : TDistributed;
    S,DT : string;

begin
    SFC:=4;

    Result:='<table '+Extra+'>';
    Result:=Result+'<tr class="TableHeading" align="right"><td
align="left">Name</td>'+
    '<td align="left">Distribution</td>'+
    '<td>Point Value</td><td>Censor Value</td><td>Low Bound</td>'+
    '<td>Up Bound</td><td>Mean</td><td>StdDev</td>'+
    '<td>P1</td><td>P10</td><td>P50</td>'+
    '<td>P90</td><td>P99</td></tr>';

    for i:=0 to Invars.Count-1 do begin
        D:=TDistributed(Invars[i]);
        //special case...
        if D.DistType=dVariableCustom then continue;

        D.GenerateStats;
        DT:=StrDist(D.DistType);

        s:=StrParm(D.DistType,1);
        if S<>' ' then
            DT:=DT+' ('+S+':'+SigFigs(D.Parm1,SFC);
        s:=StrParm(D.DistType,2);
        if S<>' ' then
            DT:=DT+' '+S+':'+SigFigs(D.Parm2,SFC);
        s:=StrParm(D.DistType,3);
        if S<>' ' then
            DT:=DT+' '+S+':'+SigFigs(D.Parm3,3);
        if pos(',',DT)>0 then Dt:=Dt+' ');

        Result:=Result+'<tr align="right"><td align="left">'+D.Name+
            '</td><td align="left">'+DT+
            '</td><td>'+SigFigs(D.OrigValue,SFC)+
            '</td><td>'+SigFigs(D.Censor,SFC)+
            '</td><td>'+SigFigs(D.Lower,SFC)+
            '</td><td>'+SigFigs(D.Upper,SFC)+
            '</td><td>'+SigFigs(D.Stats.Mean,SFC)+
            '</td><td>'+SigFigs(D.Stats.StdDev,SFC)+
            '</td><td>'+SigFigs(D.Stats.Quantiles[1],SFC)+
            '</td><td>'+SigFigs(D.Stats.Quantiles[10],SFC)+

```

```

        '</td><td>'+SigFigs(D.Stats.Quantiles[50],SFC)+
        '</td><td>'+SigFigs(D.Stats.Quantiles[90],SFC)+
        '</td><td>'+SigFigs(D.Stats.Quantiles[99],SFC)+
        '</td></tr>';

    end;
    Result:=Result+'</table><br>';
end;
}

procedure TUncertaintyStudy.GenerateStats;
var i: integer;
    T: TDistributed;
begin
    for i:=0 to OutVars.count-1 do begin
        T:=TDistributed(OutVars.items[i]);
        T.GenerateStats;
    end;
    for i:=0 to InVars.count-1 do begin
        T:=TDistributed(InVars.items[i]);
        T.GenerateStats;
    end;
end;

procedure TUncertaintyStudy.LoadFromStream(AStream : TStream);
begin
end;

procedure TUncertaintyStudy.SaveToStream(AStream : TStream);
begin
end;

function TUncertaintyStudy.FindInVarPointer(Name: string): pDouble;
var i : integer;
begin
    if FindInVar(Name,i) then begin
        if TDistributed(InVars.Items[i]).IsExtended then
            Result:=TDistributed(InVars.Items[i]).Value
        else
            raise exception.Create('Not implemented for variants: '+Name)
        end else
            raise exception.Create('Could not find '+Name+' in FindInVarPointer');
    end;

begin
end.

```