```
IsReal[x_] := (Im[x] == 0);
GetCopulaInfo[Copula_] := Module[{ϕ, ψ, ZDist, CopulaName, θ},
   ZDist = None;
   If[Copula =!= "Product",
    {CopulaName, θ} = Copula;
    Switch[CopulaName,
     "Clayton",
     ϕ[u̇_] = (1 + u̇/θ)^(-θ);
     ψ[u̇_] = θ (u̇^(-1/θ) - 1);
     ZDist = GammaDistribution[θ, 1/θ];
     ,
     "Frank",
     ϕ[u̇_] = 1/Log[θ] Log[1 + Exp[-u̇] (Exp[Log[θ]] - 1)];
     ψ[u̇_] = -Log[ (Exp[Log[θ] u̇] - 1) / (Exp[Log[θ]] - 1)],
     "GumbelHougaard",
     ϕ[u̇_] = Exp[-u̇^(1/θ)];
     ψ[u̇_] = (-Log[u̇])^θ;
     ZDist = StableDistribution[1, 1/θ, 1, 0, Cos[Pi / (2 θ)]^θ],
     _,
     ϕ = None; ψ = None
    ]];
   {ϕ, ψ, ZDist}
  ];

GetPDFsCDFs[Margs_] := Module[{n = Length[Margs], LBs, UBs, PDFs, CDFs},
   (* Extract the simple form for the pdfs/cdfs assuming the input is inside the relevant support. *)
   LBs = Table[Quantile[Margs[[i]], 0], {i, n}]; UBs = Table[Quantile[Margs[[i]], 1], {i, n}];
   PDFs = Table[Refine[PDF[Margs[[i]], ẋ_i], ẋ_i > LBs[[i]] && ẋ_i < UBs[[i]]], {i, n}];
   CDFs = Table[Refine[CDF[Margs[[i]], ẋ_i], ẋ_i > LBs[[i]] && ẋ_i < UBs[[i]]], {i, n}];
   {PDFs, CDFs}
  ];

GetJointPDF[Copula_, Margs_] := Module[{n = Length[Margs]},
   (* Calculate the pdf of X using properties of Archimedean copulas *)
   Block[{ϕ, ψ, ZDist, CPDF, IndPDF},
    {ϕ, ψ, ZDist} = GetCopulaInfo[Copula];
    CPDF = If[IsIndCopula[Copula], 1,
      (D[ϕ[t], {t, n}] /. {t → Sum[ψ[u̇_i], {i, n}]}) * Product[ψ'[u̇_i], {i, n}]
     ];

    IndPDF = Product[pdf, {pdf, First@GetPDFsCDFs[Margs]}];
    IndPDF * CPDF
   ]
  ];

DelayMarginals[Margs_] := (Margs[[1, 0]] === LogNormalDistribution);
IsIndCopula[Copula_] := (Copula === "Product");
HasMOForm[Copula_] := ListQ[Copula] && (Copula[[1]] == "Clayton" || Copula[[1]] == "GumbelHougaard");

GetTestName[Copula_, Margs_] := Module[{n = Length[Margs]},
   If[IsIndCopula[Copula], "Independent", ToString[Copula[[1]]]] <> "_" <> StringTrim[ToString[Margs[[1, 0]]], "Distribution"] <> "_" <> ToString[n] <> "_dim"
  ];

SimulateXs[Copula_, Margs_, R_] := Module[{n = Length[Margs], ϕ, ψ, ZDist, Zs, Vs, RVUnifs},
   {ϕ, ψ, ZDist} = GetCopulaInfo[Copula];

   (* If we have Marshal-Olkin dependence, we simulate with this *)
   If[HasMOForm[Copula],
    Print["Simulating using Marshall-Olkin form..."];

    Zs = RandomVariate[ZDist, R];
    Vs = RandomVariate[ExponentialDistribution[1], {n, R}];

    {Transpose[Table[InverseCDF[Margs[[i]], ϕ[Vs[[i]]/Zs]], {i, n}]], Zs}
    ,
    (* If not it is independent or Frank copula. *)
    Print["Simulating using built-in method"];

    {RandomVariate[ProductDistribution @@ Margs, R], None}
   ]
  ];

GetConditionalProbs[Copula_, Margs_, R_, ss_, RVs_, Zs_, FullPDF_, LeaveOut_] := Module[{n = Length[Margs], ϕ, ψ, ZDist, PDFs, CDFs, Others, Si, InSupport},

   {ϕ, ψ, ZDist} = GetCopulaInfo[Copula];
   {PDFs, CDFs} = GetPDFsCDFs[Margs];

   (* Get the indices which are conditioned upon. *)
   Others = DeleteCases[Range[n], LeaveOut];

   (* Sum over the conditional RVs. *)
   Si = Total /@ RVs[[ ;; , Others]];

   (* Find the support of Margs[LeaveOut] *)
   Block[{LB, UB},
    LB = Quantile[Margs[[LeaveOut]], 0];
    UB = Quantile[Margs[[LeaveOut]], 1];

    InSupport[ẋ_] = If[
      UB < ∞,
      If[LB > -∞, Boole[ẋ > LB && ẋ < UB], Boole[ẋ < UB]],
      If[LB > -∞, Boole[ẋ > LB], 1]
     ];
   ];

   (* If we have the Marshal-Olkin form, compute the extended CondMC estimator *)
   If[HasMOForm[Copula],
    Block[{CondCPDF, fLeaveOut, FLeaveOut, Pos},

     fLeaveOut[x_] = PDFs[[LeaveOut]] /. {ẋ_LeaveOut → x};
     FLeaveOut[x_] = CDFs[[LeaveOut]] /. {ẋ_LeaveOut → x};

     Re@Table[
       Pos = InSupport /@ (s - Si);

       Quiet[(-Zs * ψ'[FLeaveOut[s - Si]] Exp[-Zs ψ[FLeaveOut[s - Si]]] fLeaveOut[s - Si] /. {Indeterminate → 0, ComplexInfinity → 0}) * Pos]
       , {s, ss}]
     ]
    ,
    Block[{CondCPDF, CondIndPDF, CondFullPDF, Pos, CondToEval, Repl},
     (* Find the conditional pdf of X_LeaveOut | X_Others *)
     CondCPDF = If[IsIndCopula[Copula], 1,
       (D[ϕ[t], {t, n - 1}] /.
          {t → Sum[ψ[u̇_i], {i, Others}]}) * Product[ψ'[u̇_i], {i, Others}]
      ];
     If[! DelayMarginals[Margs], CondCPDF = CondCPDF /. Table[u̇_i → CDFs[[i]], {i, n}]];

     CondIndPDF = Product[PDFs[[i]], {i, Others}];
     CondFullPDF = CondIndPDF * CondCPDF;

     Re@Table[
       Pos = InSupport /@ (s - Si);
       CondToEval = FullPDF/CondFullPDF;

       Repl = Flatten@{
          Table[{
            If[! FreeQ[CondToEval, ẋ_i], ẋ_i → RVs[[ ;; , i]], {}],
            If[! FreeQ[CondToEval, u̇_i], u̇_i → CDF[Margs[[i]], RVs[[ ;; , i]]], {}]
           }, {i, Others}],
          ẋ_LeaveOut → s - Si,
          If[! FreeQ[CondToEval, u̇_LeaveOut], u̇_LeaveOut → CDF[Margs[[LeaveOut]], s - Si], {}]
         };
       Quiet[(CondToEval /. Repl /. {Indeterminate → 0, ComplexInfinity → 0}) * Pos]
       , {s, ss}]
     ]
    ]
  ];

CondMCandAKEstimators[Copula_, Margs_, R_, ss_, RVs_, Zs_] := Module[{n = Length[Margs], FullPDF, SingleCondIndex, SingleCondEsts, SingleCondVars, SingleCondTime, CondEsts, CondVars, CondTime, AKEsts, AKVars, AKTime, AllSingleConds, AllConds, AllAKs},

   SingleCondIndex = n;

   AllSingleConds = ConstantArray[0, {Length[ss], R}];
   AllConds = ConstantArray[0, {Length[ss], R}];
   AllAKs = ConstantArray[0, {Length[ss], R}];

   SingleCondTime = 0; CondTime = 0; AKTime = 0;

   FullPDF = GetJointPDF[Copula, Margs];

   (* Do each of the n versions of these estimators *)
   Block[{CondProbs, CondProbTime},
    Do[
     If[R > 10^5, Print["Starting CondMC+AK step ", LeaveOut, " of ", n]];

     CondProbTime = First@Timing[
        CondProbs = GetConditionalProbs[Copula, Margs, R, ss, RVs, Zs, FullPDF, LeaveOut];
       ];
     AllConds += 1/n CondProbs;
     CondTime += CondProbTime;

     (* If this is the special dimension for the 'single' CondMC estimator, store it separately. *)
     If[LeaveOut == SingleCondIndex,
      AllSingleConds = CondProbs;
      SingleCondTime += CondProbTime;
     ];

     (* Compute the Asmussen-Kroese density estimates by reusing the previously calculated conditional densities. *)
     Block[{Others, Mi, Si},
      AKTime += CondProbTime + First@Timing[
          Others = DeleteCases[Range[n], LeaveOut];

          (* Calculate the maxima & sums of the X_Others vectors *)
          {Mi, Si} = ({Max[#], Total[#]} & /@ RVs[[ ;; , Others]])ᵀ;

          AllAKs += Table[CondProbs[[i]] Table[Boole[(Mi[[r]] < ss[[i]] - Si[[r]])], {r, R}], {i, Length[ss]}];
         ];
     ];
     , {LeaveOut, n}];
   ];

   (* Calculate the final/combined estimates and their variances. *)
   SingleCondTime += First@Timing[{SingleCondEsts, SingleCondVars} = ({Mean[#], Variance[#]} & /@ AllSingleConds)ᵀ];
   CondTime += First@Timing[{CondEsts, CondVars} = ({Mean[#], Variance[#]} & /@ AllConds)ᵀ];
   AKTime += First@Timing[{AKEsts, AKVars} = ({Mean[#], Variance[#]} & /@ AllAKs)ᵀ];

   {SingleCondEsts, SingleCondVars, SingleCondTime, CondEsts, CondVars, CondTime, AKEsts, AKVars, AKTime}
  ];

PushoutEstimator[Copula_, Margs_, R_, ss_, RVs_] := Module[{n = Length[Margs], PDFs, CDFs, FullPDF, Grads, OurEsts},

   (* Perform the push-out estimator *)
   Print["Starting push-out estimator"];

   {PDFs, CDFs} = GetPDFsCDFs[Margs];

   (* Sometimes the repeated CDF evaluation is very slow, so we do a trick to avoid repeated calls to this. *)
   FullPDF = GetJointPDF[Copula, Margs] /. If[DelayMarginals[Margs], Table[u̇_i → u̇_i[ẋ_i], {i, n}],
      Table[u̇_i → CDFs[[i]], {i, n}]];

   (* Calculate the gradient of the log density at each replicate point.  *)
   Block[{Derivs, Repl, M},

    Derivs = Grad[Log[FullPDF], Table[ẋ_i, {i, n}]];

    Repl = Flatten@{
       Table[{
         If[! FreeQ[Derivs, u̇_i], {u̇_i[ẋ_i] → CDF[Margs[[i]], RVs[[ ;; , i]]], u̇_i'[ẋ_i] → PDF[Margs[[i]], RVs[[ ;; , i]]]}, {}],
         If[! FreeQ[Derivs, ẋ_i], ẋ_i → RVs[[ ;; , i]], {}]
        }, {i, n}]
      };

    (* This line literally breaks when one marginal is the exponential distribution -- that's because the derivative is constant in that direction,
    so once the replacement "Repl" has occurred then the resulting 'thing' isn't a proper matrix. It's too ugly to fix this now. *)
    M = Transpose[Derivs /. Repl];
    Grads = Table[RVs[[i]].M[[i]], {i, R}] + n;
   ];

   (* Construct the two versions of the estimator, then make the approximately optimal mixture of them. *)
   Block[{Sums, Inds, Lefts, Rights, PilotSize, CC, p},
    Sums = Total /@ RVs;
    OurEsts = Table[
      Inds = Table[Boole[Sr ≤ s], {Sr, Sums}];
      Lefts = Grads * Inds / s;
      Rights = - Grads * (1 - Inds) / s;

      PilotSize = Max[3, Ceiling[0.05 * R]];
      CC = Covariance[Transpose@{Lefts[[ ;; PilotSize]], Rights[[ ;; PilotSize]]}];
      p = (CC[[2, 2]] - CC[[1, 2]]) / (CC[[1, 1]] + CC[[2, 2]] - 2 CC[[1, 2]]);

      p * Lefts[[PilotSize + 1 ;;]] + (1 - p) Rights[[PilotSize + 1 ;;]]
      , {s, ss}];
   ];

   ({Mean[#], Variance[#]} & /@ OurEsts)ᵀ
  ];


(* http://mathematica.stackexchange.com/questions/69771/implement-the-bisection-algorithm-elegantly-and-easily *)
biSection[func_, {a_, b_}, eps_?Positive] /; (a < b && func[a] func[b] < 0) :=
  Module[{c = a, d = b}, With[{test = If[func[a] > 0, Greater, Less]}, While[d - c > eps, With[{e = (c + d) / 2}, If[test[func[e], 0], c = e, d = e]]];
    {c, d}]];

VectorisedNewtonsMethod[fn_, targets_, niter_, xstarts_: None] := Module[{update, xs, diffs},
   xs = If[xstarts === None, 1.0 * targets, xstarts];
   update[x_, t_] = FullSimplify[(fn[x] - t)/D[fn[x], x]];

   Do[
    xs = xs - update[xs, targets] /. {Underflow[] → Indeterminate};

    , {i, niter}];

   xs];

VectorisedHalleysMethod[fn_, targets_, niter_, xstarts_: None] := Module[{loggradf, logupdate, update, xs, diffs},
   xs = If[xstarts === None, 1.0 * targets,
     Table[If[xstarts[[i]] =!= Indeterminate, xstarts[[i]], 1.0 * targets[[i]]], {i, Length[xstarts]}]
    ];

   update[x_, t_] = FullSimplify[(fn[x] - t)/fn'[x] (1 + ((fn[x] - t) fn''[x])/(2 fn'[x]^2))];
   Do[

    xs = xs - update[xs, targets] /. {Underflow[] → Indeterminate};

    , {i, niter}];

   xs];

SimulateConditionallyFromG[Copula_, Margs_, R_, s_, Uniforms_, PreviousInverses_: None, Verbose_: False] := Module[{n = Length[Margs], cdf, Us, Ys, ϕ, ψ, YPartialSums, LRs, Inverses, SumPsis, DϕJMinus1, DϕJMinus1AtSumPsis, maxOutputDesired, Invs, InvTime = 0},

   Us = ConstantArray[None, {R, n}];
   Ys = ConstantArray[None, {R, n}];

   {ϕ, ψ} = GetCopulaInfo[Copula][[1 ;; 2]];
   If[! IsIndCopula[Copula],
    DϕJMinus1[ẋ_] = ϕ[ẋ]
   ];

   Us[[ ;; , 1]] = Uniforms[[ ;; , 1]] * CDF[Margs[[1]], s];
   Ys[[ ;; , 1]] = InverseCDF[Margs[[1]], Us[[ ;; , 1]]] / s;
   YPartialSums = Ys[[ ;; , 1]];

   LRs = ConstantArray[CDF[Margs[[1]], s], R];

   Inverses = {};

   Do[
    If[Verbose, Print["Smallest leftover val is ", 1 - Max[YPartialSums]]];

    maxOutputDesired = CDF[Margs[[j]], s * (1 - YPartialSums)];

    Block[{maxInputRequired},

     maxInputRequired = If[IsIndCopula[Copula],
       maxOutputDesired,

       DϕJMinus1[x_] = D[DϕJMinus1[x], x];

       SumPsis = Total /@ (ψ[Us[[ ;; , 1 ;; j - 1]]]) /. {Overflow[] → Indeterminate};

       DϕJMinus1AtSumPsis = DϕJMinus1[SumPsis];

       DϕJMinus1[SumPsis + ψ[maxOutputDesired]] / DϕJMinus1AtSumPsis
      ];

     LRs = LRs * maxInputRequired;

     Us[[ ;; , j]] = Uniforms[[ ;; , j]] * maxInputRequired;
    ];

    If[! IsIndCopula[Copula],
     InvTime = InvTime + First@AbsoluteTiming[
         Block[{InvArgs, order, prev = 0},
          InvArgs = Us[[ ;; , j]] * DϕJMinus1AtSumPsis;

          Invs = VectorisedHalleysMethod[DϕJMinus1, InvArgs, 10^4,
            If[PreviousInverses =!= None, PreviousInverses[[j - 1]], None]
           ];

          Invs = Invs /. {Overflow[] → Indeterminate, Underflow[] → Indeterminate};

          AppendTo[Inverses, Invs];

          Us[[ ;; , j]] = ϕ[Invs - SumPsis];

          LRs = LRs * Table[If[IsReal[Us[[r, j]]], 1, Indeterminate], {r, R}];

          If[! AllTrue[Us[[ ;; , j]], IsReal],
           Throw["This shouldn't be complex valued " <> ToString[Us[[ ;; , j]]]]
          ];
         ];
        ];
    ];

    Ys[[ ;; , j]] = InverseCDF[Margs[[j]], Us[[ ;; , j]]] / s;
    YPartialSums = YPartialSums + Ys[[ ;; , j]];
    , {j, 2, n}];

   If[Verbose && ! IsIndCopula[Copula], Print["Inversion time: ", InvTime]];

   {Ys, LRs, Inverses}
  ];

PushoutISEstimator[Copula_, Margs_, R_, ss_] := Module[{n = Length[Margs], PDFs, CDFs, FullPDF, Derivs, OurISEsts},

   (* Perform the push-out + IS estimator *)
   Print["Starting push-out + IS estimator"];

   {PDFs, CDFs} = GetPDFsCDFs[Margs];

   (* Sometimes the repeated CDF evaluation is very slow, so we do a trick to avoid repeated calls to this. *)
   FullPDF = GetJointPDF[Copula, Margs] /. If[DelayMarginals[Margs], Table[u̇_i → u̇_i[ẋ_i], {i, n}],
      Table[u̇_i → CDFs[[i]], {i, n}]];

   (* Calculate the gradient of the log density.  *)
   Derivs = Grad[Log[FullPDF], Table[ẋ_i, {i, n}]];

   Block[{RVUnifs, PreviousInverses, Ys, LRs, Xs, Repl, M, Grads},
    (* Get some uniforms using low discrepency sequence. *)
    BlockRandom[
     SeedRandom[1, Method → {"MKL", Method → {"Sobol", "Dimension" → n}}];
     (*SeedRandom[1];*)
     RVUnifs = RandomVariate[UniformDistribution[], {RforIS[R], n} (*,WorkingPrecision→20*)]
    ];

    PreviousInverses = None;
    OurISEsts = Table[
      Print[N[s], " of ", Max[ss]];

      (* Use these uniforms to generate from the new IS density. *)
      {Ys, LRs, PreviousInverses} = SimulateConditionallyFromG[Copula, Margs, RforIS[R], s, RVUnifs, PreviousInverses];

      Xs = Ys * s;

      Repl = Flatten@{
         Table[{
           If[! FreeQ[Derivs, u̇_i], {u̇_i[ẋ_i] → CDF[Margs[[i]], Xs[[ ;; , i]]], u̇_i'[ẋ_i] → PDF[Margs[[i]], Xs[[ ;; , i]]]}, {}],
           If[! FreeQ[Derivs, ẋ_i], ẋ_i → Xs[[ ;; , i]], {}]
          }, {i, n}]
        };

      M = Transpose[Derivs /. Repl];
      Grads = Table[Xs[[i]].M[[i]], {i, RforIS[R]}] + n;

      (Grads / s) * LRs
      , {s, ss}];
   ];

   ({Mean[#], Variance[#]} & /@ DeleteCases[OurISEsts, Indeterminate])ᵀ
  ];

PrintResults[EstNames_, AllEsts_, AllStdDevs_, AllTimes_, AllREs_] := Module[{},
   (* Estimates. *)
   Print["Estimates:"];
   Do[Print["\t* ", EstNames[[e]], ": ", AllEsts[[e]]], {e, Length[AllEsts]}];

   (* Standard deviations. *)
   Print["Standard deviations:"];
   Do[Print["\t* ", EstNames[[e]], ": ", AllStdDevs[[e]]], {e, Length[AllStdDevs]}];

   (* Computation times. *)
   Print["Times:"];
   Do[Print["\t* ", EstNames[[e]], ": ", AllTimes[[e]]], {e, Length[AllTimes]}];

   (* Estimated relative errors. *)
   Print["Estimated Relative errors:"];
   Do[Print["\t* ", EstNames[[e]], ": ", AllREs[[e]]], {e, Length[AllREs]}];
  ];

PlotResults[TestName_, ss_, EstNames_, AllEsts_, AllStdDevs_, AllTimes_, AllREs_] := Module[{WithS, Colours, EstimatesPlotSmall, EstimatesPlot, StdDevsPlotSmall, SqrtWNRVPlotSmall, SqrtWNRVPlot, CombinedPlot},

   WithS[Data_] := Transpose[{ss, Data}];
   Colours = Lookup["DefaultPlotStyle"]@Lookup[Method]@Charting`ResolvePlotTheme[Automatic, Plot];

   EstimatesPlotSmall =
    ListPlot[Map[WithS, AllEsts], Joined → True, ImageSize → {250, 125}, AspectRatio → 1 / 2, PlotStyle → Thickness[0.012], AxesLabel → {"x", "Estimates"}];

   EstimatesPlot =
    ListPlot[Map[WithS, AllEsts], Joined → True, ImageSize → {500, 125}, AspectRatio → 0.21, PlotStyle → Thickness[0.006], AxesLabel → {"x", "Estimates"}];

   StdDevsPlotSmall =
    ListPlot[Map[WithS, AllStdDevs], Joined → True, PlotStyle → Thickness[0.012], AxesLabel → {"x", "Standard Deviation"}, ImageSize → {250, 125}, AspectRatio → 1 / 2];

   SqrtWNRVPlotSmall =
    ListPlot[Map[WithS, AllREs], Joined → True, ImageSize → {250, 125}, AspectRatio → 1 / 2, PlotStyle → Thickness[0.012], AxesLabel → {"x", "Sqrt(WNRV)"}];

   SqrtWNRVPlot =
    ListPlot[Map[WithS, AllREs], Joined → True, ImageSize → {500, 125}, AspectRatio → 0.21, PlotStyle → Thickness[0.006], AxesLabel → {"x", "Sqrt(WNRV)"}];

   (* Large estimates, smaller WNRV and std. devs. *)
   CombinedPlot = Legended[Column[{EstimatesPlot, Row[{SqrtWNRVPlotSmall, StdDevsPlotSmall}]}, Center], Placed[LineLegend[Colours, EstNames, LegendLayout -> "Row"], Below]];

   Print[TestName <> ".pdf"];
   Print[CombinedPlot];
   Export[TestName <> ".pdf", CombinedPlot];
  ];

PrintAndPlotResults[Copula_, Margs_, R_, ss_, PushoutEsts_, PushoutVars_, PushoutTime_, PushoutISEsts_, PushoutISVars_, PushoutISTime_, SingleCondEsts_, SingleCondVars_, SingleCondTime_, CondEsts_, CondVars_, CondTime_,
   AKEsts_, AKVars_, AKTime_, KDEEsts_, KDETime_] := Module[{TestName, WithS, Colours, EstNames, AllEsts, AllStdDevs, AllTimes, AllREs, EstimatesPlotSmall, StdDevsPlotSmall, SqrtWNRVPlot, SqrtWNRVPlotSmall, CombinedPlot},

   TestName = GetTestName[Copula, Margs];

   (* Setup everything with both conditional MC's. *)
   EstNames = {"Sensitivity", "Single CondMC", "CondMC", "AK"};
   AllEsts = {PushoutEsts, SingleCondEsts, CondEsts, AKEsts};
   AllStdDevs = Sqrt@{PushoutVars, SingleCondVars, CondVars, AKVars};
   AllTimes = {PushoutTime, SingleCondTime, CondTime, AKTime};
   AllREs = Quiet[Sqrt@{(PushoutTime * (PushoutVars / R)/PushoutEsts^2), (SingleCondTime * (SingleCondVars / R)/SingleCondEsts^2), (CondTime * (CondVars / R)/CondEsts^2), (AKTime * (AKVars / R)/AKEsts^2)}, {Power::infy, Infinity::indet}];

   PrintResults[EstNames, AllEsts, AllStdDevs, AllTimes, AllREs];

   PlotResults[TestName <> "_both_cond", ss, EstNames, AllEsts, AllStdDevs, AllTimes, AllREs];

   (* Use just the averaged CondMC. *)
   EstNames = {"Sensitivity", "CondMC", "AK"};
   AllEsts = {PushoutEsts, CondEsts, AKEsts};
   AllStdDevs = Sqrt@{PushoutVars, CondVars, AKVars};
   AllTimes = {PushoutTime, CondTime, AKTime};
   AllREs = Quiet[Sqrt@{(PushoutTime * (PushoutVars / R)/PushoutEsts^2), (CondTime * (CondVars / R)/CondEsts^2), (AKTime * (AKVars / R)/AKEsts^2)}, {Power::infy, Infinity::indet}];

   PlotResults[TestName, ss, EstNames, AllEsts, AllStdDevs, AllTimes, AllREs];

   (* Use just the single CondMC. *)
   EstNames = {"Sensitivity", "Single CondMC", "AK"};
   AllEsts = {PushoutEsts, SingleCondEsts, AKEsts};
   AllStdDevs = Sqrt@{PushoutVars, SingleCondVars, AKVars};
   AllTimes = {PushoutTime, SingleCondTime, AKTime};
   AllREs = Quiet[Sqrt@{(PushoutTime * (PushoutVars / R)/PushoutEsts^2), (SingleCondTime * (SingleCondVars / R)/SingleCondEsts^2), (AKTime * (AKVars / R)/AKEsts^2)}, {Power::infy, Infinity::indet}];

   PlotResults[TestName <> "_single_cond", ss, EstNames, AllEsts, AllStdDevs, AllTimes, AllREs];

   Print[TestName, " done"];
  ];

RunTest[Test_, R_, NumPoints_] := Module[{Copula, Margs, ss, quant, SimulationTime, RVs, Zs, SingleCondEsts, SingleCondVars, SingleCondTime, CondEsts, CondVars, CondTime, AKEsts, AKVars, AKTime, PushoutEsts, PushoutVars,
    PushoutTime, PushoutISEsts, PushoutISVars, PushoutISTime, KDEEsts, KDETime},


   Block[{Marg, n, sRange},
    (* Extract relevant copula properties for this test case *)
    (*{Copula,Marg,n,sRange}=Test;*)
    {Copula, Marg, n} = Test;

    (* If given a list of marginals use them, otherwise take marginals to be identically distributed *)
    Margs = If[ListQ[Marg], Marg, ConstantArray[Marg, n]];

    (*ss= Range[sRange[[1]],sRange[[2]],(sRange[[2]]-sRange[[1]])/NumPoints]//DeleteCases[#,0]&;*)
   ];

   Print["Starting test: ", GetTestName[Copula, Margs]];

   SeedRandom[1];
   Print["Time to simulate (trial run to get 95% quantile): ", First@Timing[
      {RVs, Zs} = SimulateXs[Copula, Margs, R];
     ]
   ];

   (* Update the s values chosen to go up to the 95% empirical quantile of the sums. *)
   quant = Quantile[Total /@ RVs, 0.95];
   ss = Range[0, quant, (quant - 0) / NumPoints] // DeleteCases[#, 0] & // DeleteCases[#, 0.0] &;
   Print["ss = ", ss];

   SeedRandom[1337];
   Print["Time to simulate (new seed): ", SimulationTime = First@Timing[
       {RVs, Zs} = SimulateXs[Copula, Margs, R];
      ]
   ];

   Print["Time to do CondMC + CondMO + AK: ", First@Timing[
      {SingleCondEsts, SingleCondVars, SingleCondTime, CondEsts, CondVars, CondTime, AKEsts, AKVars, AKTime} = CondMCandAKEstimators[Copula, Margs, R, ss, RVs, Zs]
     ]
   ];

   Print["Single Cond MC time: ", SingleCondTime];
   Print["Cond MC time: ", CondTime];
   Print["AK time: ", AKTime];

   Print["Push-out took ", PushoutTime = First@Timing[
       {PushoutEsts, PushoutVars} = PushoutEstimator[Copula, Margs, R, ss, RVs];
      ]
   ];

   If[IsIndCopula[Copula],
    Print["Push-out with IS took ", PushoutISTime = First@Timing[
        {PushoutISEsts, PushoutISVars} = PushoutISEstimator[Copula, Margs, R, ss];
       ]
     ];
   ];

   (*Print["KDE took: ",KDETime=First@Timing[
       Block[{KDEDist,Sums=Total/@RVs},
        KDEDist=KernelMixtureDistribution[Sums];
        KDEEsts=PDF[KDEDist,ss]+PDF[KDEDist,-ss];
       ]]
    ];*)

   PrintAndPlotResults[Copula, Margs, R, ss, PushoutEsts, PushoutVars, PushoutTime + SimulationTime, PushoutISEsts, PushoutISVars, PushoutISTime, SingleCondEsts, SingleCondVars, SingleCondTime + SimulationTime, CondEsts,
    CondVars, CondTime + SimulationTime, AKEsts, AKVars, AKTime + SimulationTime, KDEEsts, KDETime + SimulationTime];
  ];

SetDirectory[NotebookDirectory[]];
Tests = {
   {{"Clayton", 1/2}, Table[GammaDistribution[i + 1, 2/(i + 1)], {i, 5}], 5},
   {{"Clayton", 1/5}, WeibullDistribution[3/10, 1], 10},
   {{"GumbelHougaard", 5}, ExponentialDistribution[1], 15},
   {{"Frank", 1/1000}, Table[LogNormalDistribution[i - 10, Sqrt[i]], {i, 10}], 10}
  };

R = 10^5; NumPoints = 50;

Do[RunTest[test, R, NumPoints], {test, Tests}];
```