```
{
 Suplementary Information for

 Drivers of Geographic patterns of North American Language Diversity

 Marco Túlio Pacheco Coelho, Elisa Barreto Pereira,
 Hannah Haynie, Thiago F. Rangel, Patrick Kavanagh,
 Kathryn R. Kirby, Simon J. Greenhill, Claire Bowern,
 Russell D. Gray, Robert K. Colwell, Nicholas Evans,
 Michael C. Gavin

 Marco Túlio Pacheco Coelho
 Email: marcotpcoelho@gmail.com
 Michael C. Gavin
 Email: Michael.Gavin@colostate.edu

 This source code represents the carrying capacity simulation model
 published by Gavin et al. (2017) - Process□based modelling shows
 how climate and demography shape language diversity.
 https://doi.org/10.1111/geb.1256

}



unit LibLPMMain;

interface

uses
  {$IfDef DLLShapes}LibFastShareMem,{$EndIf}
  SysUtils, Classes, Math, Windows, SyncObjs,
  {$IfNDef CONSOLE}
  ExtCtrls,
  {$EndIf}
  LibTypes, LibGIS, LibFiles, LibStats, LibPhyStats,
  LibGeometry, LibMCMC,
  LibMatrix, LibProbDistrib;

Type
   TUpdateProc = Procedure of object;

Type
   TModelFunction = (TMFPower, TMFLogistic, TMFExponential, EstimatedDensity);

Type
 TModelParameters = Record
   OutPutFolder, SimName: String;

   DataPath: String;
   MapShapeFile: String;
   MapShapeSmallResFile: String;
   EnvDataFile: String;
   SubsitModeParamFile: String;
   AreaParamFile: String;
   MaxPopSizeFile: String;
   NumInitLang: Integer;
   InitPopSize: Integer;
   PopPerLanguage: Integer;
   InitCell: Integer;

   ModelFunction: TModelFunction;

   CarryingCapParams: TDblVector;

   r: Double;                          // Intrinsic population growth rate
   NeighborDistance: Double;
   SpeciationTime: Integer;

   RandomSeed: Integer;

   ObsRichMap: TDblVector;
   ObsRSFD: TDblVector;
```

```
 75:     ObsAvgRSMap: TDblVector;
 76:    End;
 77:
 78: Type
 79:  TParComboRepsResults = Record
 80:    SumRichPerCell: TDblVector;
 81:    Sum2RichPerCell: TDblVector;
 82:    SumAvRangeSizePerCell: TDblVector;
 83:    Sum2AvRangeSizePerCell: TDblVector;
 84:    AcumRSFD: TDblVector;
 85:    NumLangs: TDblVector;
 86:    RepCount: Integer;
 87:   End;
 88:
 89:  TModelReScaleResults = Record
 90:    AvRangeSizePerCell: TDblVector;
 91:    RichPerCell: TDblVector;
 92:    RSFD: TDblVector;
 93:   End;
 94:
 95:  TModelFitStatsResults = Record
 96:    nLangs: Double;
 97:    RegResAvgRich: TRegResults;
 98:    RegResAvgRS: TRegResults;
 99:    KS: Double;
100:   End;
101:
102:
103: Type
104:  PLanguage = ^TLanguage;
105:  TPLangVec = Array of PLanguage;
106:
107:  PMapCell = ^TMapCell;
108:  TPMapCellVec = Array of PMapCell;
109:
110:  PLangMapCell = ^TLangMapCell;
111:  TPLangMapCellVec = Array of PLangMapCell;
112:
113:  TColor = -$7FFFFFFF-1..$7FFFFFFF;
114:
115:  TMapCell = Record
116:    IdCell: Integer;
117:    Coord: TPoint2D;
118:
119:    Environment: TDblVector;
120:    Area: Double;
121:    Neighbors: TPMapCellVec;
122:    K: Integer;
123:
124:    Languages: TPLangVec;
125:    TotPopSize: Integer;
126:    DeltaPop: Integer;
127:    PopSaturated: Boolean;
128:
129:    ColonizedOnce: Boolean;
130:   end;
131:
132:  TLangMapCell = Record
133:    IdCell: Integer;
134:    PosVec: Integer;
135:
136:    MapCell: PMapCell;
137:    Language: PLanguage;
138:
139:    K: Integer;
140:    LangPopSizeInCell: Integer;
141:    DeltaLangPop: Integer;
142:
143:    PatchTag: Integer;
144:    PrevPatchTag: Integer;
145:
146:    BorderCell: Boolean;
147:   End;
148:  TLangMapCellVec = Array of TLangMapCell;
```

```
149:
150:  TLangPatch = Record
151:    PatchSeparationTime: TIntVector;   // Temporal distance to other patches
152:    Cells: TPMapCellVec;               // Cells that belong to the patch
153:   End;
154:
155:  TLanguage = Record
156:    IdLang: Integer;
157:
158:    OccupCells: TLangMapCellVec;
159:    SubsistModeProp: TDblVector;
160:    TotPopSize, MaxPopSize: Integer;
161:    DeltaPop: Integer;
162:
163:    ExtantLang: Boolean;
164:    NeverExisted: Boolean;
165:    ExtinctLang: Boolean;
166:    RecentLang: Boolean;
167:    TransformedLang: Boolean;
168:
169:    DaughterOf: Integer;
170:    AncestorOf: TPLangVec;
171:    BornAt: Integer;
172:    DiedAt: Integer;
173:
174:    Perimeter: Cardinal;     // nCells_Border
175:    Area: Cardinal;          // nCells_Total
176:
177:    NumPatches: Integer;
178:    Patches: Array of TLangPatch;
179:
180:    ChangedThisStep: Boolean;
181:    ChangedLastStep: Boolean;
182:    PopSizeAccounted: Boolean;
183:
184:    Color: TColor;
185:   End;
186:
187:  TMaxPopClass = Record
188:    Min, Max: Double;          // Min and Max limits of pop size in each class
189:
190:    ObsCount: Integer;          // Number of empirical languages that fall in the class
191:    ObsFrequency: Double;      // Frequency of languages that fall in the class
192:
193:    SimCount: Integer;          // Number of simulated languages that fall in the class
194:    SimFrequency: Double;      // Frequency of simulated languages that fall in the class
195:
196:    MaxPopVec: TDblVector;     // Vector with observed (empirical) language pop sizes
197:   End;
198:
199:  TMaxPopDistrib = Array of TMaxPopClass;
200:
201:  TSim = Class(TThread)
202:    Private
203:      SimId: Integer;
204:
205:      // Random number generator and sample function is here to assure that results can be
    replicated
206:      LstRandSeed: Integer;           // Random Seed at the begining of the simulation
207:      RandSeed: Integer;              // Current Seed
208:      Procedure Randomize;            // Create a new random seed
209:      Function Random: Extended;      // Generates a random number
210:      Function Sample(List: TIntVector; n: Integer = -1; WithReplacement: Boolean = False)
    : TIntVector;
211:    Public
212:     DebugMode: Boolean;
213:
214:     Halted: Boolean;
215:     StopRepWhenHalt: Boolean;
216:     RunStepsContinuosly: Boolean;
217:     RunRepsContinuosly: Boolean;
218:
219:     ModelParameters: TModelParameters;
220:
```

```
221:     MapAnim: Boolean;
222:     GIFAnim: Boolean;
223:     AVIAnim: Boolean;
224:     MapAnimSampInt: Integer;
225:     MapAnimExpInt: Integer;
226:     GraphAnim: Boolean;
227:     GraphAnimSampInt: Integer;
228:     GraphAnimExpInt: Integer;
229:     AnimDelay: Double;
230:     PlotPhy: Boolean;
231:
232:     PrjName: String;
233:     OutPutFolder: String;
234:     FileOutReplicate: ^TextFile;
235:     FileOutReplicateWrite: ^RTL_CRITICAL_SECTION;
236:
237:     ModelSet: Boolean;
238:     AutoProg: Boolean;
239:
240:     MapShape: TShapeFile;
241:     MapShapeSmallRes: TShapeFile;
242:     MapCells: Array of TMapCell;        // All cells in the map
243:     Langs: Array of TLanguage;          // All langauges
244:
245:     LangSeq: TIntVector;
246:     CellSeq: TIntVector;
247:
248:     WrkCells: TIntVector;
249:     WrkLangs: TIntVector;
250:
251:     MaxPopVec: TDblMatrix;
252:     MaxPopDistrib: TMaxPopDistrib;
253:
254:     InitCell: Integer;
255:
256:     TotNumTimeSteps: Integer;
257:     TotNumReps: Integer;
258:     TotNumCells: Integer;
259:     TotNumLangs: Integer;
260:     TotNumLangsEver: Integer;
261:     TotNumEnvVars: Integer;
262:     TotNumCarryingCapParams: Integer;
263:     TotNumCellsColonizedOnce: Integer;
264:
265:     ActNumReps: Integer;
266:
267:     TotNumPatches: Integer;
268:
269:     TimeStep: Integer;
270:     Rep: Integer;
271:     SimSpeed: Double;
272:
273:     ErrMsg: String;
274:
275:     ReScaledRep: TModelReScaleResults;
276:     FitStatsRep: TModelFitStatsResults;
277:     AvgFitStats: TModelFitStatsResults;
278:     FitIndex: Double;
279:
280:     ParComboResults: TParComboRepsResults;
281:
282:     RunNextStepEvent: TEvent;
283:     RunNextRepEvent: TEvent;
284:
285:     StepFinished: TEvent;
286:     RepFinished: TEvent;
287:     SimFinished: TEvent;
288:
289:     UpdateNextStepExternalProcedure: TUpdateProc;
290:
291:
292:     Constructor Create(CreateSuspended: Boolean;
293:                        pStopRepWhenHalt: Boolean = False;
294:                        pRunStepsContinuosly: Boolean = False;
```

```
295:                         pRunRepsContinuosly: Boolean = False);
296:
297:     Procedure RunNextStep;
298:
299:     Function SetUpModel(ReSetMapData: Boolean = True): Boolean;
300:     Function ReSetModel: Boolean;
301:     Procedure ClearParComboResults;
302:
303:     Function CellChangePop(Lang: PLanguage; MapCell: PMapCell; DeltaNIndiv: Integer): PLa
    ngMapCell; overload;
304:     Function CellChangePop(LangMapCell: PLangMapCell; DeltaNIndiv: Integer): PLangMapCell
    ; overload;
305:
306:     Function CellChangeDeltaPop(Lang: PLanguage; MapCell: PMapCell; DeltaNIndiv: Integer)
    : PLangMapCell; overload;
307:     Function CellChangeDeltaPop(LangMapCell: PLangMapCell; DeltaNIndiv: Integer): PLangMa
    pCell; overload;
308:
309:     Procedure RemoveFromWrkCells(MapCell: TMapCell);
310:     Procedure RemoveFromWrkLangs(Language: PLanguage);
311:
312:     Procedure AddtoWrkCells(MapCell: TMapCell);
313:     Procedure AddToWrkLangs(Language: PLanguage);
314:
315:     Function GetPopSize(Lang: PLanguage; MapCell: PMapCell): Integer;       overload;
316:     Function GetPopSize(LangMapCell: PLangMapCell): Integer;                overload;
317:     Function GetDeltaPopSize(Lang: PLanguage; MapCell: PMapCell): Integer;  overload;
318:     Function GetDeltaPopSize(LangMapCell: PLangMapCell): Integer;           overload;
319:
320:     Function GetAllNeighbors(LangCell: PLangMapCell): TPMapCellVec;
321:     Function GetNeighborsDifferentLanguage(LangCell: PLangMapCell): TPLangMapCellVec;
322:     Function CountNeighborsSameLanguage(LangCell: PLangMapCell): Integer;
323:     Function GetNeighborsSameLanguage(LangCell: PLangMapCell): TPLangMapCellVec;
                overload;
324:     Function GetNeighborsSameLanguage(Language: PLanguage; Neighbors: TPMapCellVec): TPLa
    ngMapCellVec; overload;
325:
326:     Function CreateNewLanguage(AncestorLang: PLanguage; SubsistModeProp: TDblVector): PLa
    nguage; overload;
327:     Function CreateNewLanguage(AncestorLang: PLanguage; Cell: PMapCell; SubsistModeProp:
    TDblVector): PLanguage; overload;
328:     Function CreateNewLanguage(AncestorLang: PLanguage; CellOccup: TPMapCellVec; SubsistM
    odeProp: TDblVector): PLanguage; overload;
329:     Function ChangeLanguageIdentity(AncestorLang: PLanguage): PLanguage;
330:
331:     Function CalcCarryingCapacity(Lang: PLanguage; MapCell: PMapCell): Integer; overload;
332:     Function CalcCarryingCapacity(LangMapCell: PLangMapCell): Integer;          overload;
333:     Function CalcCarryingCapacity(MapCell: PMapCell): Integer;                  overload;
334:
335:     Function SampleMaxPopSize: Integer;
336:     Procedure UpdatePopSizeFrequency(Language: PLanguage);
337:
338:     Function GetLangMapCell(Lang: PLanguage; MapCell: PMapCell): PLangMapCell;
339:     Function GetLangMapCellVec(Lang: PLanguage; MapCellVec: TPMapCellVec): TPLangMapCellV
    ec;
340:     Function IntRND(Min, Max: Double): Integer;
341:
342:     Function ReScale: TModelReScaleResults;
343:     Function CalcRepFitStats: TModelFitStatsResults;
344:     Procedure CalcAvgRepFitStats;
345:     Procedure StoreRepResults;
346:     Function CalcRepFitIndex: Double;
347:     Function CalcThreadFitIndex: Double;
348:
349:     Procedure ExportParComboEstResults;
350:
351:     Function WritePhy(Parent: Integer; t: Integer): String;
352:     Procedure ExportResults;
353:   protected
354:     procedure Execute; override;
355:     Procedure UpdateNextStep;
356:     Procedure UpdateNextRep;
357:     Procedure UpdateSimFinished;
358:     Procedure ShowError;
```

```
359:   End;
360:
361:  TSimParallel = Class
362:    ModelParameters: TModelParameters;
363:    GibbsSamp: TGibbs;
364:
365:    SimVec: Array of TSim;
366:    HandleVec: Array of THandle;
367:
368:    FileOutReplicate: TextFile;
369:    FileOutReplicateWrite: TRTLCriticalSection;
370:    FileOutSim: TextFile;
371:
372:    AvgFitStats: TModelFitStatsResults;
373:    FitIndex: Double;
374:    SimFinished: TEvent;
375:
376:    MapShape: TShapeFile;
377:    MapShapeSmallRes: TShapeFile;
378:    MapCells: Array of TMapCell;        // All cells in the map
379:    CellSeq: TIntVector;
380:    TotNumCells: Integer;
381:    TotNumEnvVars: Integer;
382:
383:    MinLinParValSearch, MaxLinParValSearch: Double;
384:    MinQuadParValSearch, MaxQuadParValSearch: Double;
385:
386:    BestFitLinParVal: Double;
387:    BestFitLinParFitIndex: Double;
388:    BestFitQuadParVal: Double;
389:    BestFitQuadParFitIndex: Double;
390:
391:    private
392:     // Only used by optimization/search functions
393:     tnReps, tnThreads: Integer;
394:
395:    public
396:      Function Execute(ModelParameters: TModelParameters;
397:                       nReps: Integer = 120;
398:                       nThreads: Integer = -1): Double;
399:
400:      Function ExecToGibbsSamp(ProposalParams: TDblVector): Double;
401:      Procedure SearchGibbs(ModelParameters: TModelParameters;
402:                            DisturbFuncs: TDisturbFuncs;
403:                            var MCMCChain: TChain;
404:                            nReps: Integer = 120;
405:                            nThreads: Integer = -1);
406:   End;
407:
408: Type
409:   TPatchCellItems = Record
410:     Found: Boolean;
411:     IdCell: Integer;
412:     AltPatchNum: Integer;
413:     AltPatchPos: Integer;
414:     Link: PMapCell;
415:    End;
416:
417:   TOnePatchControl = Record
418:     PatchesLostCellsFrom: TIntVector; // List of patches that had cells now in this patch
419:     PatchCells: Array of TPatchCellItems;
420:    End;
421:
422:   TPatchControl = Array of TOnePatchControl;
423:
424: Type
425:   TLangPatchInfo = Record
426:     PatchSeparationTime: TIntVector;    // Temporal distance to other patches
427:     Cells: TPMapCellVec;
428:    end;
429:
430:
431: {$IfDef CONSOLE}
432: Procedure ErrorMessage(Msg: String);
```

```
433: {$EndIf}
434:
435: implementation
436:
437: {$IfNDef CONSOLE}
438: Uses
439:   UntControl, UntMap;
440: {$EndIf}
441:
442: Constructor TSim.Create(CreateSuspended: Boolean;
443:                         pStopRepWhenHalt: Boolean = False;
444:                         pRunStepsContinuosly: Boolean = False;
445:                         pRunRepsContinuosly: Boolean = False);
446:  begin
447:
448:   inherited Create(CreateSuspended);
449:   Self.FreeOnTerminate:= False;
450:
451:   Halted:= False;
452:
453:   MapAnim:= True;
454:   GIFAnim:= False;
455:   AVIAnim:= True;
456:   GraphAnim:= False;
457:   PlotPhy:= True;
458:   ModelSet:= False;
459:   AutoProg:= False;
460:   TimeStep:= 0;
461:   SimSpeed:= 0.001;
462:
463:   TotNumReps:= 1;
464:   TotNumTimeSteps:= MaxInt;
465:   Rep:= 0;
466:
467:   Self.StopRepWhenHalt:= pStopRepWhenHalt;
468:   Self.RunStepsContinuosly:= pRunStepsContinuosly;
469:   Self.RunRepsContinuosly:= pRunRepsContinuosly;
470:
471:   RunNextStepEvent:= TEvent.Create;
472:   RunNextStepEvent.ResetEvent;
473:
474:   RunNextRepEvent:= TEvent.Create;
475:   RunNextRepEvent.ResetEvent;
476:
477:   StepFinished:= TEvent.Create;
478:   StepFinished.ResetEvent;
479:
480:   RepFinished:= TEvent.Create;
481:   RepFinished.ResetEvent;
482:
483:   SimFinished:= TEvent.Create;
484:   SimFinished.ResetEvent;
485:
486:  end;
487:
488: Procedure TSim.Execute;
489:  begin
490:   // If any replicate fails, the actual number of replicates is reduced
491:   ActNumReps:= TotNumReps;
492:
493:   ClearParComboResults;
494:
495:   // Simulation Loop
496:   While True do
497:    begin
498:     Rep:= Rep + 1;
499:
500:     PrjName:= 'Rep' + IntToStr(Rep);
501:
502:     If Rep > TotNumReps then
503:       Break;
504:
505:     If Finished or Terminated then
506:       Break;
```

```
507:
508:     ReSetModel;
509:
510:     // Steps loop
511:     While True do
512:      begin
513:       // Waiting for the command to run the next step
514:       If (not RunStepsContinuosly) then
515:        begin
516:         RunNextStepEvent.ResetEvent;
517:
518:         // Update status of the simulation here
519:         UpdateNextStep;
520:
521:         WaitForSingleObject(RunNextStepEvent.Handle, INFINITE);
522:        end;
523:
524:       If Finished or Terminated then
525:         Break;
526:
527:       // Run the next step here.
528:       RunNextStep;
529:
530:       StepFinished.SetEvent;
531:       StepFinished.ResetEvent;
532:
533:       If Finished or Terminated then
534:         Break;
535:
536:       // Continuous run until halt?
537:       If (StopRepWhenHalt and Halted) or (TimeStep > TotNumTimeSteps) then
538:         Break;
539:      end;
540:
541:     RepFinished.SetEvent;
542:     RepFinished.ResetEvent;
543:
544:     If Finished or Terminated then
545:       Break;
546:
547:     ReScaledRep:= ReScale;
548:     FitStatsRep:= CalcRepFitStats;
549:     CalcRepFitIndex;
550:     CalcAvgRepFitStats;
551:     StoreRepResults;
552:
553:     UpdateNextRep;
554:
555:     // Waiting for the command to run the next simulation
556:     If (not RunRepsContinuosly) then
557:      begin
558:       RunNextRepEvent.ResetEvent;
559:       WaitForSingleObject(RunNextRepEvent.Handle, INFINITE);
560:      end;
561:    end;
562:
563:   CalcThreadFitIndex;
564:
565:   // Should be disabled in case of MCMC search!
566:   ExportParComboEstResults;
567:
568:   UpdateSimFinished;
569:
570:   SimFinished.SetEvent;
571:   SimFinished.ResetEvent;
572:  end;
573:
574: Procedure TSim.UpdateNextStep;
575:  begin
576:   UpdateNextStepExternalProcedure;
577:  end;
578:
579: Procedure TSim.UpdateNextRep;
580:  var
```

```
581:   i: Integer;
582:  begin
583:   Try
584:     If TTextRec(FileOutReplicate^).Mode <> 0 then
585:      begin
586:       EnterCriticalSection(FileOutReplicateWrite^);
587:
588:       WriteLn(FileOutReplicate^);
589:       Write(FileOutReplicate^,
590:             LstRandSeed:0, #9,
591:             InitCell:0, #9);
592:
593:       for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
594:         Write(FileOutReplicate^,
595:             ModelParameters.CarryingCapParams[i]:0:10, #9);
596:
597:       Write(FileOutReplicate^,
598:             FitIndex:0:5, #9,
599:             FitStatsRep.nLangs:0:0, #9,
600:             FitStatsRep.RegResAvgRich.r2:0:5, #9,
601:             FitStatsRep.RegResAvgRS.r2:0:5, #9,
602:             FitStatsRep.KS:0:5);
603:
604:       LeaveCriticalSection(FileOutReplicateWrite^);
605:      end;
606:   Except
607:     LeaveCriticalSection(FileOutReplicateWrite^);
608:    End;
609:  end;
610:
611: Procedure TSim.UpdateSimFinished;
612:  begin
613:   // Empty function
614:  end;
615:
616: Procedure TSim.RunNextStep;
617:  var
618:   TmpNeighborCells: TPMapCellVec;
619:   TmpNeighborCells2: TPMapCellVec;
620:   TmpNeighborLangCells: TPLangMapCellVec;
621:
622:   TmpGroupCells: TPMapCellVec;
623:   PrevLangPatches: Array of TLangPatchInfo;
624:   NewLangPatches: Array of TLangPatchInfo;
625:   SplitLanguage: TLangPatchInfo;
626:
627:   KVec: TIntVector; SumK: Int64;
628:   NVec: TIntVector; SumN, SumN2: Int64;
629:   DifNKVec: TIntVector; SumDifNK: Int64;
630:   IncVec: TIntVector;
631:   PropIncVec: TDblVector;
632:   IncN, DeltaPop: Int64;
633:   LangSeq2: TIntVector;
634:   CellSeq2: TIntVector;
635:
636:   PopCap: Integer;
637:   ProbSpeciate: Double;
638:
639:   TmpDbl: Double;
640:   TmpInt: Integer;
641:   TmpIntVec: TIntVector;
642:   TmpDblMat: TDblMatrix;
643:
644:   Cell1, Cell2: PMapCell;
645:   D1, D2: Double;
646:
647:   Found1: Boolean;
648:   Found2: Boolean;
649:
650:   RndLangSeq: Boolean;
651:
652:   PrevPatchControl: TPatchControl;
653:   NewPatchControl: TPatchControl;
654:
```

```
655:   Language: PLanguage;
656:   NewLanguage: PLanguage;
657:
658:   AncestorColor: TColor;
659:
660:  var
661:   c, l, s: Integer;
662:   c1, c2, c3, lt, s1: Integer;
663:
664:   i, j, k, w: Integer;
665:   p, n: Integer;
666:   a1, a2: Integer;
667:
668:   TmpFile: TextFile;
669:  Label
670:   Lbl1;
671:  begin
672:
673:   Try
674:
675:     // Do we want to shuffle the order of languages at each time step?
676:     RndLangSeq:= False;
677:
678:     // Re-shuffle the order of languages
679:     If RndLangSeq then
680:       LangSeq:= Sample(WrkLangs);
681:
682:     // Next time step in the simulation
683:     TimeStep:= TimeStep + 1;
684:
685:     If TimeStep > 10000 then
686:      begin
687: {$IfNDef CONSOLE}
688:     If AVIRec <> Nil then
689:       AVIRec.StopRecording;
690: {$EndIf}
691:
692: {$IfDef CONSOLE}
693:       WriteLn('');
694:       Write('Trapped! RandomSeed: ', Self.LstRandSeed:0, '   ');
695:       for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
696:         Write(ModelParameters.CarryingCapParams[i]:0:7, '   ');
697: {$EndIf}
698:
699:       Randomize;
700:       ModelParameters.RandomSeed:= RandSeed;
701:
702:       ResetModel;
703:       Exit;
704:
705: {$IfDef CONSOLE}
706:       WriteLn('');
707:       Write('Trapped! RandomSeed: ', Self.LstRandSeed:0, '   ');
708:       for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
709:         Write(ModelParameters.CarryingCapParams[i]:0:7, '   ');
710:       ReadLn(TimeStep);
711: {$EndIf}
712:      end;
713:
714:
715:
716:
717:
718:
719:
720:
721:     lt:= -1;
722:     While lt+1 < Length(WrkLangs) do
723:      begin
724:       lt:= lt + 1;
725:
726:       Langs[WrkLangs[lt]].ChangedLastStep:= Langs[WrkLangs[lt]].ChangedThisStep;
727:       Langs[WrkLangs[lt]].ChangedThisStep:= False;
728:      end;
```

```
729:
730:
731:
732:
733:     // Shuffle cell order
734:     CellSeq:= Sample(WrkCells);
735:     For c2:= 0 to Length(CellSeq)-1 do
736:      begin
737:       c3:= CellSeq[c2];
738:
739:       // Is the cell occupied by a language?
740:       If MapCells[c3].Languages = Nil then
741:         Continue;
742:
743:       If Length(MapCells[c3].Languages) > 1 then
744:         c3:= c3;
745:
746:       // This cell and all neighbors are already saturated
747:       If (MapCells[c3].PopSaturated) or
748:          (MapCells[c3].TotPopSize = 0) then
749:         Continue;
750:
751:       // Copy the sequence of languages in cell c3 to vector
752:       SetLength(LangSeq2, Length(MapCells[c3].Languages));
753:       For lt:= 0 to Length(LangSeq2)-1 do
754:         LangSeq2[lt]:= lt;
755:
756:       // Shuffle sequence of languages existing in cell c3
757:       If Length(LangSeq2) > 1 then
758:         LangSeq2:= Sample(LangSeq2);
759:
760:       // For each language in cell c3, in a random order
761:       For lt:= 0 to Length(LangSeq2)-1 do
762:        begin
763:         // Get language id
764:         l:= MapCells[c3].Languages[LangSeq2[lt]].IdLang;
765:
766:         // Search for the record of the cell in the list of cells occupied by the languag
    e
767:         c:= -1;
768:         For c1:= 0 to Length(Langs[l].OccupCells)-1 do
769:          begin
770:           If MapCells[c3].IdCell = Langs[l].OccupCells[c1].IdCell then
771:            begin
772:             c:= c1;
773:             Break;
774:            end;
775:          end;
776:
777:         // Get information of sourounding cells, given a language
778:
779:         // First in the vector is the current cell
780:         TmpNeighborCells:= GetAllNeighbors(@Langs[l].OccupCells[c]);
781:
782:         // Get all neighboring cells. Cells not occupied by Langs[l] are Nil
783:         TmpNeighborLangCells:= GetNeighborsSameLanguage(@Langs[l], TmpNeighborCells);
784:
785:         // Stores Carrying Capacity
786:         SumK:= 0;
787:         SumN:= 0;
788:         SetLength(KVec, Length(TmpNeighborCells));
789:         For i:= 0 to Length(TmpNeighborCells)-1 do
790:          begin
791:           KVec[i]:= 0;
792:
793:           // Calculates carrying capacity given the environment
794:           // Each language may respond differently to the environment (subsistence mode)
795:           KVec[i]:= CalcCarryingCapacity(@Langs[l], TmpNeighborCells[i]);
796:
797:           // Regional carrying capacity
798:           SumK:= SumK + KVec[i];
799:
800:           // Total current population in the neighborhood
801:           SumN:= SumN + TmpNeighborCells[i].TotPopSize;
```

```
802:           end;
803:
804:          // This cell and its neighbours are fully saturated
805:          If SumN = SumK then
806:           begin
807:            // Remove cell c3 from the list of cells to be checked.
808:            // The cell is saturated and does not need re-checking
809:            RemoveFromWrkCells(MapCells[c3]);
810:
811:            // Set as saturated.
812:            MapCells[c3].PopSaturated:= True;
813:
814:            // Skip loop
815:            Continue;;
816:           end;
817:
818:          SumN:= 0;
819:          SumN2:= 0;
820:          SumDifNK:= 0;
821:
822:          SetLength(NVec, Length(TmpNeighborLangCells));
823:          SetLength(DifNKVec, Length(TmpNeighborLangCells));
824:          SetLength(CellSeq2, Length(TmpNeighborLangCells));
825:
826:          // Population size change
827:          SetLength(IncVec, Length(TmpNeighborCells));
828:          For i:= 0 to Length(TmpNeighborLangCells)-1 do
829:           begin
830:            IncVec[i]:= 0;
831:            NVec[i]:= 0;
832:
833:            CellSeq2[i]:= i;
834:
835:            If TmpNeighborLangCells[i] <> Nil then
836:             begin
837:              // Get current population size
838:              NVec[i]:= TmpNeighborLangCells[i].LangPopSizeInCell;
839:
840:              // if it is the focal cell
841:              If i = 0 then
842:               begin
843:                // If the cell population is above carrying capacity
844:                If NVec[i] > KVec[i] then
845:                  // Reduce population to carrying capacity
846:                  IncVec[i]:= KVec[i] - NVec[i];
847:               end;
848:
849:              // Regional population size
850:              SumN:= SumN + NVec[i] + IncVec[i];
851:
852:              DeltaPop:= TmpNeighborLangCells[i].DeltaLangPop;
853:             end
854:            Else
855:             begin
856:              NVec[i]:= 0;
857:              DeltaPop:= 0;
858:             end;
859:
860:            // Difference between carrying capacity and current population,
861:            // also take into account population change in the next cycle
862:            DifNKVec[i]:= KVec[i] - (NVec[i] + DeltaPop);
863:            SumDifNK:= SumDifNK + DifNKVec[i];
864:
865:            SumN2:= SumN2 + NVec[i] + DeltaPop;
866:           end;
867:
868:          // Population increase
869:          If SumDifNK > 0 then
870:           begin
871:            If ((SumN2 - SumK) <> 0) and (SumK <> 0) then
872:             begin
873:              // Regional population increase. Current population limits pop increase
874:              //IncN:= Trunc((r*SumN)*(1-(SumN/SumK)));
875:
```

```
876:             // Regional population increase. Current and future population limits pop inc
    rease
877:             //IncN:= Trunc((r*SumN) * (1-(SumN2/SumK)));
878:
879:             // Regional population increase. Current and future population limits pop inc
    rease
880:             IncN:= Round((ModelParameters.r*Langs[l].OccupCells[c].LangPopSizeInCell) * (
    1-(SumN2/SumK)));
881:
882:             // Fix possible rounding errors at very low values
883:             If IncN = 0 then
884:               IncN:= 1;
885:            end
886:           Else
887:             IncN:= 0;
888:
889:
890:
891:           // Fix any rounding error
892:           If IncN > SumDifNK then
893:             IncN:= SumDifNK;
894:
895:           SumN:= 0;
896:           SumN2:= 0;
897:
898:           SetLength(PropIncVec, Length(DifNKVec));
899:           While SumN < IncN do
900:            begin
901:             CellSeq2:= Sample(CellSeq2);
902:             For i:= 0 to Length(CellSeq2)-1 do
903:              begin
904:               PropIncVec[CellSeq2[i]]:= 0;
905:
906:               // Define population change in next cycle
907:               If IncVec[CellSeq2[i]] < 0 then
908:                begin
909:                 CellChangeDeltaPop(@Langs[l], TmpNeighborCells[CellSeq2[i]], IncVec[CellS
    eq2[i]]);
910:                 IncVec[CellSeq2[i]]:= 0;
911:                end Else
912:
913:               If DifNKVec[CellSeq2[i]] = 0 then
914:                begin
915:                 Continue;
916:                end
917:
918:               Else
919:                begin
920:                 // Proportion of increase in each cell
921:                 PropIncVec[CellSeq2[i]]:= (DifNKVec[CellSeq2[i]] / SumDifNK);
922:
923:                 // Actual increase in each cell
924:                 IncVec[CellSeq2[i]]:= IncVec[CellSeq2[i]] + Round(PropIncVec[CellSeq2[i]]
     * IncN);
925:
926:                 SumN:= SumN + IncVec[CellSeq2[i]];
927:
928:                 If SumN > IncN then
929:                  begin
930:                   IncVec[CellSeq2[i]]:= IncVec[CellSeq2[i]] - (SumN - IncN);
931:                   SumN:= IncN;
932:                  end;
933:
934:                 // Define population change in next cycle
935:                 If IncVec[CellSeq2[i]] <> 0 then
936:                   CellChangeDeltaPop(@Langs[l], TmpNeighborCells[CellSeq2[i]], IncVec[Cel
    lSeq2[i]]);
937:                end;
938:              end;
939:
940:           // There has been no increase in population since last loop
941:           // Possible cause is rounding errors
942:           If SumN = SumN2 then
943:             Break;
```

```
944:
945:             SumN2:= SumN;
946:            end;
947:
948:           // Due to rounding errors, population has not increased properly
949:           If SumN <> IncN then
950:            begin
951:             // Distribute the new population randomly among cells
952:             CellSeq2:= Sample(CellSeq2);
953:             For i:= 0 to Length(CellSeq2)-1 do
954:              begin
955:               If PropIncVec[CellSeq2[i]] = 0 then
956:                 Continue;
957:               IncVec[CellSeq2[i]]:= IncVec[CellSeq2[i]] + 1;
958:               SumN:= SumN + 1;
959:               CellChangeDeltaPop(@Langs[l], TmpNeighborCells[CellSeq2[i]], IncVec[CellSeq
    2[i]]);
960:               If SumN = IncN then
961:                 Break;
962:              end;
963:            end;
964:
965:          end Else
966:
967:         // Population decrease
968:         If SumDifNK < 0 then
969:          begin
970:           For i:= 0 to Length(IncVec)-1 do
971:            begin
972:             If IncVec[i] <> 0 then
973:               CellChangeDeltaPop(@Langs[l], TmpNeighborCells[i], IncVec[i]);
974:            end;
975:          end;
976:        end;
977:      end;
978:
979:
980:
981:
982:
983:     // Check if the simulation has reached a halt
984:     // The simulation reaches an end when all cells have been colonized by at least one l
    anguage
985:     // So that no language can expand in any direction
986:     l:= 0;
987:     lt:= -1;
988:     // For each working language
989:     While lt+1 < Length(WrkLangs) do
990:      begin
991:       lt:= lt + 1;
992:
993:       // If at least one language has been changed during this time step
994:       If Langs[WrkLangs[lt]].ChangedThisStep then
995:        begin
996:         // Sum one here
997:         l:= 1;
998:
999:         // Skip the search
1000:         Break;
1001:        end;
1002:      end;
1003:
1004:
1005:
1006:
1007:
1008:
1009:     // No languages have been changed during this time step
1010:     // This means there is no longer any dynamic in the simulation
1011:     If l = 0 then
1012:      begin
1013:
1014:       // Check all existing languages to see if ANY of them have an empty cell around its
     range
```

```
1015:       // If there is any cell with a sourouding empty cell, a new language emerges at thi
      s cell
1016:
1017:       LangSeq2:= Nil;
1018:       For i:= 0 to Length(Langs)-1 do
1019:        begin
1020:         If Langs[i].TotPopSize > 0 then
1021:          begin
1022:           SetLength(LangSeq2, Length(LangSeq2)+1);
1023:           LangSeq2[Length(LangSeq2)-1]:= i;
1024:          end;
1025:        end;
1026:
1027:       LangSeq:= Sample(LangSeq2);
1028:
1029:       For lt:= 0 to Length(LangSeq)-1 do
1030:        begin
1031:         Language:= @Langs[LangSeq[lt]];
1032:
1033:         TmpNeighborCells2:= Nil;
1034:
1035:         // For each cell occupied by the language
1036:         For c:= 0 to Length(Language.OccupCells)-1 do
1037:          begin
1038:
1039:           // Is it a cell at the border?
1040:           If Language.OccupCells[c].BorderCell then
1041:            begin
1042:             // Get all neighboring cells of the border cell
1043:             TmpNeighborCells:= Nil;
1044:             TmpNeighborCells:= GetAllNeighbors(@Language.OccupCells[c]);
1045:
1046:             If TmpNeighborCells = Nil then
1047:               Continue;
1048:
1049:             For i:= 0 to Length(TmpNeighborCells)-1 do
1050:              begin
1051:               // Is this neighboring cell unoccupied?
1052:               If TmpNeighborCells[i].Languages = Nil then
1053:                begin
1054:                 // For each already recorded cell
1055:                 For j:= 0 to Length(TmpNeighborCells2)-1 do
1056:                  begin
1057:                   // Has this cell already recorded to be sampled?
1058:                   If TmpNeighborCells[i] = TmpNeighborCells2[j] then
1059:                    begin
1060:                     // If yes, then erase it
1061:                     TmpNeighborCells[i]:= Nil;
1062:                     Break;
1063:                    end;
1064:                  end;
1065:
1066:                 // If this possible cell has not been erased, record it
1067:                 If TmpNeighborCells[i] <> Nil then
1068:                  begin
1069:                   SetLength(TmpNeighborCells2, Length(TmpNeighborCells2)+1);
1070:                   TmpNeighborCells2[Length(TmpNeighborCells2)-1]:= TmpNeighborCells[i];
1071:                  end;
1072:                end;
1073:              end;
1074:            end;
1075:          end;
1076:
1077:         // Is there an empty neighbor?
1078:
1079:         // If this is true multiple new languages will be generated at each time step, as
      long as there are
1080:         // enough empty neighbor cells. If this is false, then only a single language wil
      l be generated at the
1081:         // edge of the geographic distribution
1082:         If TmpNeighborCells2 <> Nil then
1083:          begin;
1084:
1085:           // Sample one of the cells occupied by the language with an empty neighbor cell
```

```
1086:           Cell1:= TmpNeighborCells2[IntRND(0,Length(TmpNeighborCells2)-1)];
1087:
1088:           // Create a new language with the population and cells of patch n
1089:           NewLanguage:= CreateNewLanguage(Language,
1090:                                           Cell1,
1091:                                           Language.SubsistModeProp);
1092:
1093:           NewLanguage.RecentLang:= True;
1094:
1095:           NewLanguage.ChangedThisStep:= True;
1096:
1097:           For i:= 0 to Length(Language.OccupCells)-1 do
1098:            begin
1099:             Language.OccupCells[i].MapCell.PopSaturated:= True;
1100:             RemoveFromWrkCells(Language.OccupCells[i].MapCell^);
1101:            end;
1102:
1103:           SetLength(WrkCells, Length(WrkCells)+1);
1104:           WrkCells[Length(WrkCells)-1]:= Cell1.IdCell;
1105:
1106:           Language.ChangedThisStep:= True;
1107:
1108:           Break;
1109:          end
1110:
1111:         // Impossible to create a new language, as the old language is sourounded
1112:         Else
1113:          begin
1114:           // Has all languages been tested, and none of them have surounding empty cells?
1115:           If lt = Length(LangSeq)-1 then
1116:             // Finish the simulation!
1117:             Halted:= True;
1118:          end;
1119:        end;
1120:      end;
1121:
1122:
1123:
1124:
1125:     // Check all languages again
1126:     // For each existing language
1127:     lt:= -1;
1128:     While lt+1 < Length(WrkLangs) do
1129:      begin
1130:       lt:= lt + 1;
1131:
1132:       // Progress with the next language sequentially
1133:       l:= lt;
1134:
1135:       // Pick a language in a random order
1136:       l:= WrkLangs[lt];
1137:       If RndLangSeq then
1138:         l:= LangSeq[lt];
1139:
1140:       // Language is good?
1141:       If (Langs[l].NeverExisted) or (Langs[l].ExtinctLang) or
1142:          (Langs[l].RecentLang) or (not Langs[l].ExtantLang) or
1143:          (Langs[l].TotPopSize = 0) then
1144:         Continue;
1145:
1146:       // Any change in this language?
1147:       If not Langs[l].ChangedThisStep then
1148:         Continue;
1149:
1150:       Langs[l].Area:= 0;
1151:       Langs[l].Perimeter:= 0;
1152:
1153:       // For each cell occupied by a language
1154:       For c:= 0 to Length(Langs[l].OccupCells)-1 do
1155:        begin
1156:
1157:         If Langs[l].OccupCells[c].MapCell.PopSaturated then
1158:           Continue;
1159:
```

```
1160:         // Just colonized a new cell
1161:         If (Langs[l].OccupCells[c].LangPopSizeInCell = 0) and
1162:            (Langs[l].OccupCells[c].DeltaLangPop > 0) then
1163:          begin
1164:           // Language colonizes a new cell.
1165:
1166:           Found1:= False;
1167:           For i:= 0 to Length(Langs[l].OccupCells[c].MapCell.Languages)-1 do
1168:            begin
1169:             If Langs[l].OccupCells[c].MapCell.Languages[i].IdLang = Langs[l].IdLang then
1170:              begin
1171:               // Language already registered as present in that cell
1172:               Found1:= True;
1173:               Break;
1174:              end;
1175:            end;
1176:
1177:           If Not Found1 then
1178:            begin
1179:             // Language not registered as present in the cell
1180:             SetLength(Langs[l].OccupCells[c].MapCell.Languages, Length(Langs[l].OccupCell
      s[c].MapCell.Languages)+1);
1181:              Langs[l].OccupCells[c].MapCell.Languages[Length(Langs[l].OccupCells[c].MapCel
      l.Languages)-1]:= @Langs[l];
1182:            end;
1183:          end;
1184:
1185:         // Update language population, in each cell
1186:         Langs[l].OccupCells[c].LangPopSizeInCell:= Langs[l].OccupCells[c].LangPopSizeInCe
      ll +
1187:                                                    Langs[l].OccupCells[c].DeltaLangPop;
1188:         If Langs[l].OccupCells[c].LangPopSizeInCell <= 0 then
1189:           Langs[l].OccupCells[c].LangPopSizeInCell:= 0;
1190:
1191:         // Update population in the cell, across languages
1192:         Langs[l].OccupCells[c].MapCell.TotPopSize:= Langs[l].OccupCells[c].MapCell.TotPop
      Size +
1193:                                                     Langs[l].OccupCells[c].DeltaLangPop;
1194:         If Langs[l].OccupCells[c].MapCell.TotPopSize <= 0 then
1195:           Langs[l].OccupCells[c].MapCell.TotPopSize:= 0;
1196:
1197:         // Update language population across cells
1198:         Langs[l].TotPopSize:= Langs[l].TotPopSize +
1199:                               Langs[l].OccupCells[c].DeltaLangPop;
1200:         If Langs[l].TotPopSize < 0 then
1201:           Langs[l].TotPopSize:= 0;
1202:
1203:
1204:         // Language is extinct at the cell
1205:         If Langs[l].OccupCells[c].LangPopSizeInCell = 0 then
1206:           Langs[l].OccupCells[c].IdCell:= -1;
1207:
1208:         Langs[l].OccupCells[c].MapCell.DeltaPop:= Langs[l].OccupCells[c].MapCell.DeltaPop
       -
1209:                                                   Langs[l].OccupCells[c].DeltaLangPop;
1210:
1211:         Langs[l].OccupCells[c].DeltaLangPop:= 0;
1212:        end;
1213:
1214:       Langs[l].DeltaPop:= 0;
1215:
1216:       Langs[l].Area:= Length(Langs[l].OccupCells);
1217:       Langs[l].Perimeter:= 0;
1218:       For c:= 0 to Langs[l].Area-1 do
1219:        begin
1220:         // At the edge of the continent?
1221:         If Length(Langs[l].OccupCells[c].MapCell.Neighbors) < 6 then
1222:           Langs[l].OccupCells[c].BorderCell:= True
1223:
1224:         // Inland at the edge of the distribution?
1225:         Else
1226:           Langs[l].OccupCells[c].BorderCell:= (CountNeighborsSameLanguage(@Langs[l].Occup
      Cells[c]) < 6);
1227:
```

```
1228:         If Langs[l].OccupCells[c].BorderCell then
1229:           Langs[l].Perimeter:= Langs[l].Perimeter + 1;
1230:        end;
1231:      end;
1232:
1233:
1234:
1235: // Process 5: Languages have maximum population size
1236:     lt:= -1;
1237:     While lt+1 < Length(WrkLangs) do
1238:      begin
1239:       lt:= lt + 1;
1240:
1241:       // Progress with the next language sequentially
1242:       l:= lt;
1243:
1244:       // Pick a language in a random order
1245:       l:= WrkLangs[lt];
1246:       If RndLangSeq then
1247:         l:= LangSeq[lt];
1248:
1249:       Language:= @Langs[l];
1250:
1251:       // Language is good?
1252:       If (Language.NeverExisted) or (Language.ExtinctLang) or
1253:          (Language.RecentLang) or (not Language.ExtantLang) or
1254:          (Language.TotPopSize = 0) then
1255:         Continue;
1256:
1257:       // Any change in this language?
1258:       If not Language.ChangedThisStep then
1259:         Continue;
1260:
1261:
1262:
1263:
1264:
1265:       // Has the population reached its maximum population size?
1266:       If (Language.TotPopSize > Language.MaxPopSize) and
1267:          (Length(Language.OccupCells) > 1) then
1268:        begin
1269:
1270:         // If new language arise at the border
1271:         // In the following code, below, the language stops expanding and a new language
      arise at the border
1272:         TmpNeighborCells2:= Nil;
1273:
1274:         // For each cell occupied by the language
1275:         For c:= 0 to Length(Language.OccupCells)-1 do
1276:          begin
1277:
1278:           // Is it a cell at the border?
1279:           If Language.OccupCells[c].BorderCell then
1280:            begin
1281:             // Get all neighboring cells of the border cell
1282:             TmpNeighborCells:= Nil;
1283:             TmpNeighborCells:= GetAllNeighbors(@Language.OccupCells[c]);
1284:
1285:             If TmpNeighborCells = Nil then
1286:               Continue;
1287:
1288:             For i:= 0 to Length(TmpNeighborCells)-1 do
1289:              begin
1290:               // Is this neighboring cell unoccupied?
1291:               If TmpNeighborCells[i].Languages = Nil then
1292:                begin
1293:                 // For each already recorded cell
1294:                 For j:= 0 to Length(TmpNeighborCells2)-1 do
1295:                  begin
1296:                   // Has this cell already recorded to be sampled?
1297:                   If TmpNeighborCells[i] = TmpNeighborCells2[j] then
1298:                    begin
1299:                     // If yes, then erase it
1300:                     TmpNeighborCells[i]:= Nil;
```

```
1301:                     Break;
1302:                    end;
1303:                  end;
1304:
1305:                 // If this possible cell has not been erased, record it
1306:                 If TmpNeighborCells[i] <> Nil then
1307:                  begin
1308:                   SetLength(TmpNeighborCells2, Length(TmpNeighborCells2)+1);
1309:                   TmpNeighborCells2[Length(TmpNeighborCells2)-1]:= TmpNeighborCells[i];
1310:                  end;
1311:                end;
1312:              end;
1313:            end;
1314:          end;
1315:
1316:         // Is there an empty neighbor
1317:
1318:         // If this is true multiple new languages will be generated at each time step, as
      long as there are
1319:         // enough empty neighbor cells. If this is false, then only a single language wil
     l be generated at the
1320:         // edge of the geographic distribution
1321:         If TmpNeighborCells2 <> Nil then
1322:          begin;
1323:
1324:           // Sample one of the cells occupied by the language with an empty neighbor cell
1325:           Cell1:= TmpNeighborCells2[IntRND(0,Length(TmpNeighborCells2)-1)];
1326:
1327:           // Create a new language with the population and cells of patch n
1328:           NewLanguage:= CreateNewLanguage(Language,
1329:                                           Cell1,
1330:                                           Language.SubsistModeProp);
1331:
1332:           NewLanguage.RecentLang:= True;
1333:
1334:           NewLanguage.ChangedThisStep:= True;
1335:
1336:           For i:= 0 to Length(Language.OccupCells)-1 do
1337:            begin
1338:             Language.OccupCells[i].MapCell.PopSaturated:= True;
1339:             RemoveFromWrkCells(Language.OccupCells[i].MapCell^);
1340:            end;
1341:
1342:           SetLength(WrkCells, Length(WrkCells)+1);
1343:           WrkCells[Length(WrkCells)-1]:= Cell1.IdCell;
1344:
1345:           Language.ChangedThisStep:= True;
1346:          end
1347:
1348:         // Impossible to create a new language, as the old language is sourounded
1349:         Else
1350:          begin
1351:           // Make this false because there is no room of a new language at the edge
1352:           Language.ChangedThisStep:= False;
1353:
1354:           // Update the realized distribution of language population sizes
1355:           UpdatePopSizeFrequency(Language);
1356:          end;
1357: //}
1358:        end;
1359:      end;
1360:
1361:
1362:
1363:
1364:
1365:
1366:
1367:
1368:     For l:= 0 to Length(WrkLangs)-1 do
1369:      begin
1370:       If Langs[WrkLangs[l]].RecentLang then
1371:        begin
1372:         TotNumPatches:= TotNumPatches + Langs[WrkLangs[l]].NumPatches;
```

```
1373:          Langs[WrkLangs[l]].RecentLang:= False;
1374:          Langs[WrkLangs[l]].ExtantLang:= True;
1375:         end;
1376:       end;
1377:
1378:
1379:
1380:
1381:      lt:= -1;
1382:      While lt+1 < Length(WrkLangs) do
1383:       begin
1384:        lt:= lt + 1;
1385:
1386:        If (not Langs[WrkLangs[lt]].ChangedLastStep) and
1387:           (not Langs[WrkLangs[lt]].ChangedThisStep) then
1388:         begin
1389:          RemoveFromWrkLangs(@Langs[WrkLangs[lt]]);
1390:          lt:= lt - 1;
1391:          Continue;
1392:         end;
1393:       end;
1394:
1395:
1396:
1397:      // All cells have been colonized at least once
1398:      // Time to finish the simulation
1399:      If ((TimeStep > 5000) and (Length(WrkLangs) = 0)) or (TotNumCellsColonizedOnce >= (To
      tNumCells-30)) then
1400:       begin
1401:        Halted:= True;
1402:       end;
1403:
1404:    Except
1405:      On E: Exception do
1406:       begin
1407:        ErrMsg:= E.Message;
1408:        {$IfNDef CONSOLE}
1409:        //Synchronize(ShowError);
1410:        {$EndIf}
1411:
1412:        {$IfDef CONSOLE}
1413:        ShowError;
1414:        {$EndIf}
1415:       end;
1416:     end;
1417:   end;
1418:
1419: Procedure TSim.ShowError;
1420:   begin
1421:    If ErrMsg = '' then
1422:     begin
1423:      ErrorMessage('Unknown Error! Please try again.');
1424:     end
1425:    Else
1426:     begin
1427:      ErrorMessage(ErrMsg);
1428:     end;
1429:   end;
1430:
1431: {$IfDef CONSOLE}
1432: Procedure ErrorMessage(Msg: String);
1433:   var
1434:    i: Integer;
1435:   begin
1436:    WriteLn(Msg);
1437:    ReadLn(i);
1438:   end;
1439: {$EndIf}
1440:
1441:
1442:
1443:
1444:
1445:
```

```
1446:
1447: Function TSim.WritePhy(Parent: Integer; t: Integer): String;
1448:  var
1449:   l, z: Integer;
1450:   fst: Boolean;
1451:  begin
1452:   fst:= True;
1453:   Result:= '()';
1454:
1455:   For l:= 0 to Length(Langs)-1 do
1456:    begin
1457:     If Langs[l].NeverExisted then
1458:       Continue;
1459:
1460:     If Langs[l].DaughterOf = Parent then
1461:      begin
1462:
1463:       if not fst then
1464:         Insert(',', Result, Length(Result))
1465:       Else
1466:         fst:= not fst;
1467:
1468:       If Langs[l].ExtantLang then
1469:         z:= TimeStep - Langs[l].BornAt
1470:       Else
1471:         z:= Langs[l].DiedAt - Langs[l].BornAt;
1472:
1473:       // Does species SpeciesIndex have children?
1474:       If Langs[l].AncestorOf = Nil then
1475:         // Because species Sp does not have any children, writes its name and close the n
      ode
1476:         Insert('Lg' + IntToStr(Langs[l].IdLang) + ':' + IntToStr(z), Result, Length(Resul
      t))
1477:
1478:       Else                                          // Species SpeciesIndex has children
1479:         // Recursively call WritePhy to enter all the children of species SpeciesIndex
1480:         Insert(WritePhy(Langs[l].IdLang, Langs[l].DiedAt) + 'Lg' + IntToStr(Langs[l].IdLa
      ng) + ':' + IntToStr(z), Result, Length(Result));
1481:      end;
1482:    end;
1483:  end;
1484:
1485:
1486: Function TSim.GetPopSize(Lang: PLanguage; MapCell: PMapCell): Integer;
1487:  var
1488:   i: Integer;
1489:  begin
1490:   Result:= 0;
1491:   For i:= 0 to Length(MapCell.Languages)-1 do
1492:    begin
1493:     If MapCell.Languages[i].IdLang = Lang.IdLang then
1494:      begin
1495:       Result:= MapCell.Languages[i].TotPopSize;
1496:       Break;
1497:      end;
1498:    end;
1499:  end;
1500:
1501:
1502: Function TSim.GetPopSize(LangMapCell: PLangMapCell): Integer;
1503:  begin
1504:   Result:= LangMapCell.LangPopSizeInCell;
1505:  end;
1506:
1507:
1508: Function TSim.GetDeltaPopSize(Lang: PLanguage; MapCell: PMapCell): Integer;
1509:  var
1510:   i: Integer;
1511:  begin
1512:   Result:= 0;
1513:   For i:= 0 to Length(MapCell.Languages)-1 do
1514:    begin
1515:     If MapCell.Languages[i].IdLang = Lang.IdLang then
1516:      begin
```

```
1517:       Result:= MapCell.Languages[i].DeltaPop;
1518:       Break;
1519:      end;
1520:    end;
1521:  end;
1522:
1523:
1524: Function TSim.GetDeltaPopSize(LangMapCell: PLangMapCell): Integer;
1525:  begin
1526:   Result:= LangMapCell.DeltaLangPop;
1527:  end;
1528:
1529: Function TSim.GetAllNeighbors(LangCell: PLangMapCell): TPMapCellVec;
1530:  var
1531:   i: Integer;
1532:   TmpIntVec: TIntVector;
1533:   TmpMapCellVec: TPMapCellVec;
1534:  begin
1535:   Result:= Nil;
1536:   SetLength(Result, Length(LangCell.MapCell.Neighbors)+1);
1537:   Result[0]:= LangCell.MapCell;
1538:
1539:   For i:= 1 to Length(Result)-1 do
1540:     Result[i]:= LangCell.MapCell.Neighbors[i-1];
1541:
1542:   Exit;
1543:
1544:
1545:
1546:
1547:   TmpMapCellVec:= Nil;
1548:   SetLength(TmpMapCellVec, Length(Result));
1549:
1550:   TmpIntVec:= Nil;
1551:   SetLength(TmpIntVec, Length(Result));
1552:
1553:   For i:= 0 to Length(Result)-1 do
1554:    begin
1555:     TmpMapCellVec[i]:= Result[i];
1556:     TmpIntVec[i]:= i;
1557:    end;
1558:
1559:   TmpIntVec:= Sample(TmpIntVec);
1560:
1561:   For i:= 0 to Length(Result)-1 do
1562:    begin
1563:     If TmpIntVec[i] > Length(Result)-1 then
1564:       TmpIntVec[i]:= Length(Result)-1;
1565:
1566:     Result[i]:= TmpMapCellVec[TmpIntVec[i]];
1567:
1568:     If Result[i].IdCell = -1 then
1569:       Result[i].IdCell:= -2;
1570:    end;
1571:  end;
1572:
1573:
1574: // Given a language, occupying a certain cell, list all neighbouring cell occupied by a d
     ifferent language.
1575: // Returns a vector of cells occupied by other languages
1576: // Vector is empty (Nil) if surounding cells are occupied by the same language
1577: Function TSim.GetNeighborsDifferentLanguage(LangCell: PLangMapCell): TPLangMapCellVec;
1578:  var
1579:   i, j, k: Integer;
1580:   Lang: PLanguage;
1581:   TmpIntVec: TIntVector;
1582:   TmpMapCellVec: TPLangMapCellVec;
1583:  begin
1584:   Result:= Nil;
1585:
1586:   Lang:= LangCell.Language;
1587:   For i:= 0 to Length(LangCell.MapCell.Neighbors)-1 do
1588:    begin
1589:     For j:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages)-1 do
```

```
1590:      begin
1591:       If LangCell.MapCell.Neighbors[i].Languages[j] <> Lang then
1592:        begin
1593:         For k:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages[j].OccupCells)-1 do
1594:          begin
1595:           If LangCell.MapCell.Neighbors[i].Languages[j].OccupCells[k].IdCell =
1596:              LangCell.MapCell.Neighbors[i].IdCell  then
1597:            begin
1598:             SetLength(Result, Length(Result)+1);
1599:             Result[Length(Result)-1]:= @LangCell.MapCell.Neighbors[i].Languages[j].OccupC
      ells[k];
1600:
1601:             Break;
1602:            end;
1603:          end;
1604:
1605:         Break;
1606:        end;
1607:      end;
1608:    end;
1609:  end;
1610:
1611: Function TSim.CountNeighborsSameLanguage(LangCell: PLangMapCell): Integer;
1612:  var
1613:   i, j, k: Integer;
1614:   Lang: PLanguage;
1615:  begin
1616:   Result:= 0;
1617:
1618:   Lang:= LangCell.Language;
1619:
1620:   For i:= 0 to Length(LangCell.MapCell.Neighbors)-1 do
1621:    begin
1622:     For j:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages)-1 do
1623:      begin
1624:       If LangCell.MapCell.Neighbors[i].Languages[j] = Lang then
1625:        begin
1626:         For k:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages[j].OccupCells)-1 do
1627:          begin
1628:           If LangCell.MapCell.Neighbors[i].Languages[j].OccupCells[k].IdCell =
1629:              LangCell.MapCell.Neighbors[i].IdCell  then
1630:            begin
1631:             Result:= Result + 1;
1632:             Break;
1633:            end;
1634:          end;
1635:
1636:         Break;
1637:        end;
1638:      end;
1639:    end;
1640:  end;
1641:
1642:
1643: // Get all neighboring cells. Cells not occupied by Langs[l] are Nil
1644: Function TSim.GetNeighborsSameLanguage(Language: PLanguage; Neighbors: TPMapCellVec): TPL
      angMapCellVec;
1645:  var
1646:   i, j, k: Integer;
1647:   TmpIntVec: TIntVector;
1648:   TmpMapCellVec: TPLangMapCellVec;
1649:  begin
1650:   SetLength(Result, Length(Neighbors));
1651:
1652:   For i:= 0 to Length(Neighbors)-1 do
1653:    begin
1654:     Result[i]:= Nil;
1655:     For j:= 0 to Length(Neighbors[i].Languages)-1 do
1656:      begin
1657:       If Neighbors[i].Languages[j] = Language then
1658:        begin
1659:         For k:= 0 to Length(Neighbors[i].Languages[j].OccupCells)-1 do
1660:          begin
1661:           If Neighbors[i].Languages[j].OccupCells[k].IdCell =
```

```
1662:              Neighbors[i].IdCell  then
1663:             begin
1664:              Result[i]:= @Neighbors[i].Languages[j].OccupCells[k];
1665:
1666:              Break;
1667:             end;
1668:           end;
1669:
1670:          Break;
1671:         end;
1672:       end;
1673:     end;
1674:  end;
1675:
1676: // Given a language, occupying a certain cell, list all neighbouring cell occupied by the
       same language.
1677: // Returns a vector of cells occupied by the same language languages
1678: // Vector is empty (Nil) if surounding cells are occupied by different languages
1679: Function TSim.GetNeighborsSameLanguage(LangCell: PLangMapCell): TPLangMapCellVec;
1680:  var
1681:   i, j, k: Integer;
1682:   Lang: PLanguage;
1683:   TmpIntVec: TIntVector;
1684:   TmpMapCellVec: TPLangMapCellVec;
1685:  begin
1686:   SetLength(Result, 1);
1687:   Result[0]:= LangCell;
1688:
1689:   Lang:= LangCell.Language;
1690:
1691:   For i:= 0 to Length(LangCell.MapCell.Neighbors)-1 do
1692:    begin
1693:     For j:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages)-1 do
1694:      begin
1695:       If LangCell.MapCell.Neighbors[i].Languages[j] = Lang then
1696:        begin
1697:         For k:= 0 to Length(LangCell.MapCell.Neighbors[i].Languages[j].OccupCells)-1 do
1698:          begin
1699:           If LangCell.MapCell.Neighbors[i].Languages[j].OccupCells[k].IdCell =
1700:              LangCell.MapCell.Neighbors[i].IdCell  then
1701:            begin
1702:             SetLength(Result, Length(Result)+1);
1703:             Result[Length(Result)-1]:= @LangCell.MapCell.Neighbors[i].Languages[j].OccupC
       ells[k];
1704:
1705:             Break;
1706:            end;
1707:          end;
1708:
1709:         Break;
1710:        end;
1711:      end;
1712:    end;
1713:  end;
1714:
1715: Function TSim.CalcCarryingCapacity(Lang: PLanguage; MapCell: PMapCell): Integer;
1716:  var
1717:   i, j, k: Integer;
1718:   Min, Max, Mid, Range, X, w: Double;
1719:   AreaInKm2: Double;
1720:   t: Double;
1721:   LangMapCell: PLangMapCell;
1722:  begin
1723:
1724:   If MapCell.Environment <> Nil then
1725:    begin
1726:
1727:     // What is the environment of the cell?
1728:     Result:= 0;
1729:
1730:     For i:= 0 to TotNumEnvVars-1 do
1731:       If IsNaN(MapCell.Environment[i]) then
1732:         Exit;
1733:
```

```
1734:
1735:
1736:     // Function between environment and carrying capacity
1737:
1738:     If MapCell.Area = 0 then
1739:       AreaInKm2:= 1
1740:     Else
1741:       AreaInKm2:= MapCell.Area;
1742:
1743:     // Power function
1744:     if ModelParameters.ModelFunction = TMFPower then
1745:      begin
1746:       // Calculate the log of expected density given the precipitation
1747:       X:= ModelParameters.CarryingCapParams[0] + (ModelParameters.CarryingCapParams[1] *
      Log10(MapCell.Environment[0]));
1748:
1749:       // Calculate the expected density in units of humans / km^2
1750:       X:= Power(10, X);
1751:      end Else
1752:
1753:
1754:      //Uses and estimated density from  41. Kavanagh PH, et al. 2018 Hindcasting global p
      opulation densities reveals forces enabling the origin of agriculture. Nat Hum Behav 2(7)
      :478-484.
1755:     if ModelParameters.ModelFunction = EstimatedDensity then
1756:      begin
1757:        X:= MapCell.Environment[0];
1758:      end Else
1759:
1760:
1761:     // Logistic function
1762:     if ModelParameters.ModelFunction = TMFLogistic then
1763:      begin
1764:       X:= (ModelParameters.CarryingCapParams[2]) / (1 + EXP((-1 * Power(10, ModelParamete
      rs.CarryingCapParams[1])) * (MapCell.Environment[0] - ModelParameters.CarryingCapParams[0
      ])));
1765:      end Else
1766:
1767:
1768:     // Exponential function
1769:     if ModelParameters.ModelFunction = TMFExponential then
1770:      begin
1771:       X:= ModelParameters.CarryingCapParams[0] + Exp(Power(10, ModelParameters.CarryingCa
      pParams[1]) * MapCell.Environment[0]);
1772:       if X < 0 then
1773:         X:= 0;
1774:      end;
1775:
1776:
1777:      // Kavanagh PH, et al. 2018 Hindcasting global population densities reveals forces e
      nabling the origin of agriculture. Nat Hum Behav 2(7):478-484.
1778:     X:= MapCell.Environment[0];
1779:
1780:     // Calculate the expected number of humans given the size of the cell
1781:     Result:= Round(X * AreaInKm2);
1782:
1783:     //Almost never happens but if the result is < than 2 people, then set the
1784:     //carrying capacity to 2
1785:     If Result < 2 then
1786:       Result:= 2;
1787:
1788:
1789:     //!!!!! NOT USED IN THIS VERSION
1790:     // Other people in the cell?
1791:     For i:= 0 to Length(MapCell.Languages)-1 do
1792:      begin
1793:       If MapCell.Languages[i].IdLang = Lang.IdLang then
1794:         Continue;
1795:
1796:       // Maximum Interference competition !!!
1797:       // There can be no two languages in a single cell
1798:       Result:= 0;
1799:       Break;
1800:       end;
```

```
1801:
1802:    end
1803:   Else
1804:     Result:= 0;
1805:  end;
1806:
1807: Function TSim.CalcCarryingCapacity(LangMapCell: PLangMapCell): Integer;
1808:  begin
1809:   Result:= CalcCarryingCapacity(LangMapCell.Language, LangMapCell.MapCell);
1810:  end;
1811:
1812: Function TSim.CalcCarryingCapacity(MapCell: PMapCell): Integer;
1813:  begin
1814:   Result:= CalcCarryingCapacity(Nil, MapCell);
1815:  end;
1816:
1817: Procedure TSim.UpdatePopSizeFrequency(Language: PLanguage);
1818:  var
1819:   BinObs, BinSim: Integer;
1820:   i, t: Integer;
1821:  begin
1822:   Try
1823:     If Language.PopSizeAccounted then
1824:       Exit;
1825:
1826:
1827:
1828:     // For each population size bin
1829:     For i:= 0 to Length(MaxPopDistrib)-1 do
1830:      begin
1831:       // Has the language SAMPLED MAXIMUM population size fallen within this bin?
1832:       If (Language.MaxPopSize > MaxPopDistrib[i].Min) and (Language.MaxPopSize <= MaxPopD
      istrib[i].Max) then
1833:        begin
1834:         // If so, then one language is added to the bin, therefore increasing its frequen
      cy
1835:         MaxPopDistrib[i].SimCount:= MaxPopDistrib[i].SimCount + 1;
1836:         Break;
1837:        end;
1838:      end;
1839:
1840:
1841:
1842:     // For each population size bin
1843:     For i:= 0 to Length(MaxPopDistrib)-1 do
1844:      begin
1845:       // Has the language SIMULATED population size fallen within this bin?
1846:       If (Language.TotPopSize > MaxPopDistrib[i].Min) and (Language.TotPopSize <= MaxPopD
      istrib[i].Max) then
1847:        begin
1848:         // If so, then one language is subtracted from the bin, therefore reducing its fr
      equency
1849:         MaxPopDistrib[i].SimCount:= MaxPopDistrib[i].SimCount - 1;
1850:         Break;
1851:        end;
1852:      end;
1853:
1854:
1855:
1856:
1857:     // Now, recalculates the frequency distributions
1858:     t:= 0;
1859:     // For each bin
1860:     For i:= 0 to Length(MaxPopDistrib)-1 do
1861:       // Counts the number of languages inside.
1862:       // The sum over all bins should be constant along the simulation
1863:       t:= t + MaxPopDistrib[i].SimCount;
1864:
1865:
1866:
1867:     // Re-calculates the frequency, which will be used as a probability
1868:     For i:= 0 to Length(MaxPopDistrib)-1 do
1869:      begin
1870:       // Ratio between count and total
```

```
1871:       MaxPopDistrib[i].SimFrequency:= MaxPopDistrib[i].SimCount / t;
1872:
1873:       // Because bins are being subtracted, they may become negative probabilities
1874:       // In this case, zero them
1875:       If MaxPopDistrib[i].SimFrequency < 0 then
1876:         MaxPopDistrib[i].SimFrequency:= 0;
1877:      end;
1878:
1879:
1880:
1881:
1882:
1883:     Language.PopSizeAccounted:= True;
1884:   Except
1885:     {$IfDef CONSOLE}
1886:       WriteLn('Problem UpdatePopSizeFrequency');
1887:       WriteLn(LstRandSeed, #9, TimeStep);
1888:       WriteLn(i, #9, t, #9, MaxPopDistrib[i].ObsCount, #9, MaxPopDistrib[i].ObsFrequency)
      ;
1889:       WriteLn(MaxPopDistrib[i].SimCount, #9, MaxPopDistrib[i].SimFrequency);
1890:       ReadLn(i);
1891:     {$EndIf}
1892:    End;
1893:  end;
1894:
1895: Function TSim.SampleMaxPopSize: Integer;
1896:  var
1897:   i, j, Bin: Integer;
1898:   W, C: TDblVector;
1899:   SW, t: Double;
1900:  begin
1901:   Try
1902:     SetLength(W, Length(MaxPopDistrib));
1903:
1904:     SW:= 0;
1905:     For i:= 0 to Length(W)-1 do
1906:      begin
1907:       W[i]:= MaxPopDistrib[i].SimFrequency;
1908:       SW:= SW + W[i];
1909:      end;
1910:
1911:     SetLength(C, Length(MaxPopDistrib)+1);
1912:     C[0]:= 0;
1913:     For i:= 1 to Length(W) do
1914:       C[i]:= C[i-1] + (W[i-1] / SW);
1915:
1916:     t:= Random;
1917:
1918:     For j:= 0 to Length(C)-2 do
1919:      begin
1920:       If (t >= C[j]) and (t <= C[j+1]) then
1921:        begin
1922:         Bin:= j;
1923:         Break;
1924:        end;
1925:      end;
1926:
1927:     Result:= Trunc(MaxPopDistrib[Bin].MaxPopVec[IntRND(0,Length(MaxPopDistrib[Bin].MaxPop
      Vec)-1)]);
1928:   Except
1929:     {$IfDef CONSOLE}
1930:       WriteLn('Problem SampleMaxPopSize');
1931:       WriteLn(LstRandSeed, #9, TimeStep);
1932:       WriteLn(i, #9, j, #9, t, #9, Bin);
1933:       ReadLn(i);
1934:     {$EndIf}
1935:    End;
1936:  end;
1937:
1938: Function TSim.GetLangMapCell(Lang: PLanguage; MapCell: PMapCell): PLangMapCell;
1939:  var
1940:   c, z: Integer;
1941:  begin
1942:   Result:= Nil;
```

```
1943:
1944:   z:= -1;
1945:   For c:= 0 to Length(Lang.OccupCells)-1 do
1946:    begin
1947:     If Lang.OccupCells[c].IdCell = MapCell.IdCell then
1948:      begin
1949:       z:= c;
1950:       Break;
1951:      end;
1952:    end;
1953:
1954:   If z = -1 then
1955:    begin
1956:     c:= Length(Lang.OccupCells);
1957:     SetLength(Lang.OccupCells, c+1);
1958:
1959:     z:= c;
1960:
1961:     Lang.OccupCells[z].IdCell:= MapCell.IdCell;
1962:     Lang.OccupCells[z].PosVec:= z;
1963:
1964:     Lang.OccupCells[z].MapCell:= MapCell;
1965:     Lang.OccupCells[z].Language:= Lang;
1966:
1967:     Lang.OccupCells[z].DeltaLangPop:= 0;
1968:     Lang.OccupCells[z].K:= 0;
1969:     Lang.OccupCells[z].LangPopSizeInCell:= 0;
1970:
1971:     Lang.OccupCells[z].PatchTag:= -1;
1972:     Lang.OccupCells[z].PrevPatchTag:= -1;
1973:    end;
1974:
1975:   Result:= @Lang.OccupCells[z];
1976:  end;
1977:
1978: Function TSim.GetLangMapCellVec(Lang: PLanguage; MapCellVec: TPMapCellVec): TPLangMapCell
      Vec;
1979:  var
1980:   i: Integer;
1981:  begin
1982:   SetLength(Result, Length(MapCellVec));
1983:   For i:= 0 to Length(Result)-1 do
1984:    begin
1985:     Result[i]:= GetLangMapCell(Lang, MapCellVec[i]);
1986:    end;
1987:  end;
1988:
1989:
1990: Function TSim.CellChangePop(Lang: PLanguage; MapCell: PMapCell; DeltaNIndiv: Integer): PL
      angMapCell;
1991:  var
1992:   i: Integer;
1993:   FoundLang: Boolean;
1994:  begin
1995:   Result:= GetLangMapCell(Lang, MapCell);
1996:
1997:   If (MapCell.DeltaPop = 0) and
1998:      (MapCell.TotPopSize = 0) then
1999:    begin
2000:     SetLength(WrkCells, Length(WrkCells)+1);
2001:     WrkCells[Length(WrkCells)-1]:= MapCell.IdCell;
2002:    end;
2003:
2004:   FoundLang:= False;
2005:   For i:= 0 to Length(MapCell.Languages)-1 do
2006:    begin
2007:     If MapCell.Languages[i].IdLang = Lang.IdLang then
2008:      begin
2009:       FoundLang:= True;
2010:       Break;
2011:      end;
2012:    end;
2013:   If Not FoundLang then
2014:    begin
```

```
2015:     SetLength(MapCell.Languages, Length(MapCell.Languages)+1);
2016:     MapCell.Languages[Length(MapCell.Languages)-1]:= Lang;
2017:    end;
2018:
2019:   If MapCell.ColonizedOnce = False then
2020:    begin
2021:     MapCell.ColonizedOnce:= True;
2022:     TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2023:    end;
2024:
2025:   Lang.TotPopSize:= Lang.TotPopSize + DeltaNIndiv;
2026:   Result.LangPopSizeInCell:= Result.LangPopSizeInCell + DeltaNIndiv;
2027:   Result.MapCell.TotPopSize:= Result.MapCell.TotPopSize + DeltaNIndiv;
2028:  end;
2029:
2030:   // This function is not being used
2031: Function TSim.CellChangePop(LangMapCell: PLangMapCell; DeltaNIndiv: Integer): PLangMapCel
      l;
2032:  begin
2033:   Result:= LangMapCell;
2034:   Result.Language.TotPopSize:= Result.Language.TotPopSize + DeltaNIndiv;
2035:   Result.LangPopSizeInCell:= Result.LangPopSizeInCell + DeltaNIndiv;
2036:   Result.MapCell.TotPopSize:= Result.MapCell.TotPopSize + DeltaNIndiv;
2037:  end;
2038:
2039: Function TSim.CellChangeDeltaPop(Lang: PLanguage; MapCell: PMapCell; DeltaNIndiv: Integer
      ): PLangMapCell;
2040:  var
2041:   i: Integer;
2042:   FoundLang: Boolean;
2043:  begin
2044:   If (MapCell.DeltaPop = 0) and
2045:      (MapCell.TotPopSize = 0) then
2046:    begin
2047:     SetLength(WrkCells, Length(WrkCells)+1);
2048:     WrkCells[Length(WrkCells)-1]:= MapCell.IdCell;
2049:    end;
2050:
2051:   Result:= GetLangMapCell(Lang, MapCell);
2052:
2053:   FoundLang:= False;
2054:   For i:= 0 to Length(MapCell.Languages)-1 do
2055:    begin
2056:     If MapCell.Languages[i].IdLang = Lang.IdLang then
2057:      begin
2058:       FoundLang:= True;
2059:       Break;
2060:      end;
2061:    end;
2062:
2063:   If Not FoundLang then
2064:    begin
2065:
2066:     If Length(MapCell.Languages) > 0 then
2067:      begin
2068:       {$IfDef CONSOLE}
2069:        WriteLn('');
2070:        WriteLn('Error! Two languages are trying to colonize the same cell...');
2071:        WriteLn('Error! Check SumK integer overflow!');
2072:        WriteLn('');
2073:        WriteLn(LstRandSeed, #9, TimeStep, #9, MapCell.IdCell, #9, Lang.IdLang);
2074:        ReadLn(i);
2075:       {$EndIf}
2076:      end;
2077:
2078:     SetLength(MapCell.Languages, Length(MapCell.Languages)+1);
2079:     MapCell.Languages[Length(MapCell.Languages)-1]:= Lang;
2080:    end;
2081:
2082:   If MapCell.ColonizedOnce = False then
2083:    begin
2084:     MapCell.ColonizedOnce:= True;
2085:     TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2086:    end;
```

```
2087:
2088:   Lang.ChangedThisStep:= True;
2089:
2090:   Lang.DeltaPop:= Lang.DeltaPop + DeltaNIndiv;
2091:   If Lang.DeltaPop + Lang.TotPopSize < 0 then
2092:     Lang.DeltaPop:= -Lang.TotPopSize;
2093:
2094:   MapCell.DeltaPop:= MapCell.DeltaPop + DeltaNIndiv;
2095:   If MapCell.DeltaPop + MapCell.TotPopSize < 0 then
2096:     MapCell.DeltaPop:= -MapCell.TotPopSize;
2097:
2098:   Result.DeltaLangPop:= Result.DeltaLangPop + DeltaNIndiv;
2099:   If Result.DeltaLangPop + Result.LangPopSizeInCell < 0 then
2100:     Result.DeltaLangPop:= -Result.LangPopSizeInCell;
2101:  end;
2102:
2103: Procedure TSim.RemoveFromWrkCells(MapCell: TMapCell);
2104:  var
2105:   i, j: Integer;
2106:  begin
2107:   For i:= 0 to Length(WrkCells)-1 do
2108:    begin
2109:     If WrkCells[i] = MapCell.IdCell then
2110:      begin
2111:       For j:= i to Length(WrkCells)-2 do
2112:        begin
2113:         WrkCells[j]:= WrkCells[j+1];
2114:        end;
2115:       SetLength(WrkCells, Length(WrkCells)-1);
2116:       Break;
2117:      end;
2118:    end;
2119:  end;
2120:
2121: Procedure TSim.RemoveFromWrkLangs(Language: PLanguage);
2122:  var
2123:   i, j: Integer;
2124:  begin
2125:   For i:= 0 to Length(WrkLangs)-1 do
2126:    begin
2127:     If WrkLangs[i] = Language.IdLang then
2128:      begin
2129:       For j:= i to Length(WrkLangs)-2 do
2130:        begin
2131:         WrkLangs[j]:= WrkLangs[j+1];
2132:        end;
2133:       SetLength(WrkLangs, Length(WrkLangs)-1);
2134:       Break;
2135:      end;
2136:    end;
2137:
2138:   UpdatePopSizeFrequency(Language);
2139:  end;
2140:
2141: Procedure TSim.AddtoWrkCells(MapCell: TMapCell);
2142:  var
2143:   i, j: Integer;
2144:  begin
2145:   For i:= 0 to Length(WrkCells)-1 do
2146:     If WrkCells[i] = MapCell.IdCell then
2147:       Exit;
2148:
2149:   SetLength(WrkCells, Length(WrkCells)+1);
2150:   WrkCells[Length(WrkCells)-1]:= MapCell.IdCell;
2151:  end;
2152:
2153: Procedure TSim.AddToWrkLangs(Language: PLanguage);
2154:  var
2155:   i, j: Integer;
2156:  begin
2157:   For i:= 0 to Length(WrkLangs)-1 do
2158:     If WrkLangs[i] = Language.IdLang then
2159:       Exit;
2160:
```

```
2161:    SetLength(WrkLangs, Length(WrkLangs)+1);
2162:    WrkLangs[Length(WrkLangs)-1]:= Language.IdLang;
2163:   end;
2164:
2165: // This function is not being used
2166: Function TSim.CellChangeDeltaPop(LangMapCell: PLangMapCell; DeltaNIndiv: Integer): PLangM
      apCell;
2167:   var
2168:    i: Integer;
2169:    FoundLang: Boolean;
2170:   begin
2171:    Result:= LangMapCell;
2172:
2173:    FoundLang:= False;
2174:    For i:= 0 to Length(LangMapCell.MapCell.Languages)-1 do
2175:     begin
2176:      If LangMapCell.MapCell.Languages[i].IdLang = Result.Language.IdLang then
2177:       begin
2178:        FoundLang:= True;
2179:        Break;
2180:       end;
2181:     end;
2182:    If Not FoundLang then
2183:     begin
2184:      SetLength(LangMapCell.MapCell.Languages, Length(LangMapCell.MapCell.Languages)+1);
2185:      LangMapCell.MapCell.Languages[Length(LangMapCell.MapCell.Languages)-1]:= Result.Langu
      age;
2186:     end;
2187:
2188:    If LangMapCell.MapCell.ColonizedOnce = False then
2189:     begin
2190:      LangMapCell.MapCell.ColonizedOnce:= True;
2191:      TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2192:     end;
2193:
2194:    Result.Language.DeltaPop:= Result.Language.DeltaPop + DeltaNIndiv;
2195:    Result.DeltaLangPop:= Result.DeltaLangPop + DeltaNIndiv;
2196:    Result.MapCell.DeltaPop:= Result.MapCell.DeltaPop + DeltaNIndiv;
2197:   end;
2198:
2199: Function TSim.ChangeLanguageIdentity(AncestorLang: PLanguage): PLanguage;
2200:   var
2201:    i, j: Integer;
2202:   begin
2203:    TotNumLangs:= TotNumLangs; // Does not change
2204:    TotNumLangsEver:= TotNumLangsEver + 1;
2205:
2206:    If TotNumLangsEver > Length(Langs) then
2207:      Raise Exception.Create('Maximum number of languages reached in ChangeLanguageIdentity
      ');
2208:
2209:    For i:= 0 to Length(WrkLangs)-1 do
2210:     begin
2211:      If WrkLangs[i] = AncestorLang.IdLang then
2212:       begin
2213:        For j:= i to Length(WrkLangs)-2 do
2214:          WrkLangs[j]:= WrkLangs[j+1];
2215:        Break;
2216:       end;
2217:     end;
2218:
2219:    WrkLangs[Length(WrkLangs)-1]:= TotNumLangsEver-1;
2220:
2221:    Langs[TotNumLangsEver-1]:= AncestorLang^;
2222:
2223:    Result:= @Langs[TotNumLangsEver-1];
2224:
2225:    SetLength(AncestorLang.AncestorOf, Length(AncestorLang.AncestorOf)+1);
2226:    AncestorLang.AncestorOf[Length(AncestorLang.AncestorOf)-1]:= Result;
2227:
2228:    AncestorLang.ExtantLang:= False;
2229:    AncestorLang.TransformedLang:= True;
2230:    AncestorLang.NeverExisted:= False;
2231:    AncestorLang.ExtinctLang:= False;
```

```
2232:   AncestorLang.RecentLang:= False;
2233:   AncestorLang.DiedAt:= TimeStep;
2234:
2235:   Result.IdLang:= TotNumLangsEver-1;
2236:
2237:   Result.ExtantLang:= True;
2238:   Result.TransformedLang:= False;
2239:   Result.NeverExisted:= False;
2240:   Result.ExtinctLang:= False;
2241:   Result.RecentLang:= True;
2242:   Result.BornAt:= TimeStep;
2243:   Result.DaughterOf:= AncestorLang.IdLang;
2244:
2245:   SetLength(Result.OccupCells, Length(Result.OccupCells));
2246:   For i:= 0 to Length(Result.OccupCells)-1 do
2247:    begin
2248:     Result.OccupCells[i].Language:= Result;
2249:
2250:     If Result.OccupCells[i].MapCell.ColonizedOnce = False then
2251:      begin
2252:       Result.OccupCells[i].MapCell.ColonizedOnce:= True;
2253:       TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2254:      end;
2255:
2256:     For j:= 0 to Length(Result.OccupCells[i].MapCell.Languages)-1 do
2257:      begin
2258:       If Result.OccupCells[i].MapCell.Languages[j] = AncestorLang then
2259:        begin
2260:         Result.OccupCells[i].MapCell.Languages[j]:= Result;
2261:
2262:         Break;
2263:        end;
2264:      end;
2265:    end;
2266:
2267:   SetLength(Result.SubsistModeProp, Length(Result.SubsistModeProp));
2268:   SetLength(Result.Patches, Length(Result.Patches));
2269:
2270:   Result.AncestorOf:= Nil;
2271:   Result.Color:= RGB(IntRND(0,255), IntRND(0,255), IntRND(0,255));
2272:  end;
2273:
2274: Function TSim.CreateNewLanguage(AncestorLang: PLanguage; SubsistModeProp: TDblVector): PL
     anguage;
2275:  begin
2276:   TotNumLangs:= TotNumLangs + 1;
2277:   TotNumLangsEver:= TotNumLangsEver + 1;
2278:
2279:   If TotNumLangsEver > Length(Langs) then
2280:     Raise Exception.Create('Maximum number of languages reached in CreateNewLanguage');
2281:
2282:   SetLength(WrkLangs, Length(WrkLangs)+1);
2283:   WrkLangs[Length(WrkLangs)-1]:= TotNumLangsEver-1;
2284:
2285:   Result:= @Langs[TotNumLangsEver-1];
2286:
2287:   Result.IdLang:= TotNumLangsEver-1;
2288:   Result.TotPopSize:= 0;
2289:   Result.MaxPopSize:= SampleMaxPopSize;
2290:   Result.PopSizeAccounted:= False;
2291:   Result.DeltaPop:= 0;
2292:   Result.NumPatches:= 0;
2293:
2294:   Result.ExtantLang:= True;
2295:   Result.NeverExisted:= False;
2296:   Result.ExtinctLang:= False;
2297:   Result.RecentLang:= False;
2298:   Result.TransformedLang:= False;
2299:
2300:   If DebugMode then
2301:    begin
2302:     If TotNumLangsEver = 1 then
2303:       Result.Color:= RGB(255, 0, 0) else
2304:     If TotNumLangsEver = 2 then
```

```
2305:       Result.Color:= RGB(0, 255, 0) else
2306:     If TotNumLangsEver = 3 then
2307:       Result.Color:= RGB(0, 0, 255)
2308:     else
2309:       Result.Color:= RGB(IntRND(0,255), IntRND(0,255), IntRND(0,255));
2310:    end
2311:   Else
2312:     Result.Color:= RGB(IntRND(0,255), IntRND(0,255), IntRND(0,255));
2313:
2314:   Result.SubsistModeProp:= SubsistModeProp;
2315:   SetLength(Result.SubsistModeProp, TotNumCarryingCapParams);
2316:
2317:   Result.OccupCells:= Nil;
2318:   Result.Patches:= Nil;
2319:
2320:   Result.NumPatches:= 1;
2321:
2322:   If AncestorLang = Nil then
2323:     Result.DaughterOf:= -1
2324:   Else
2325:     Result.DaughterOf:= AncestorLang.IdLang;
2326:
2327:   Result.ChangedThisStep:= True;
2328:   Result.ChangedLastStep:= True;
2329:
2330:   Result.BornAt:= TimeStep;
2331:   Result.DiedAt:= -1;
2332:  end;
2333:
2334: Function TSim.CreateNewLanguage(AncestorLang: PLanguage; Cell: PMapCell; SubsistModeProp:
       TDblVector): PLanguage;
2335:  var
2336:   PopSize: Integer;
2337:  begin
2338:   Result:= CreateNewLanguage(AncestorLang, SubsistModeProp);
2339:
2340:   PopSize:= CalcCarryingCapacity(Result, Cell);
2341:
2342:   SetLength(AncestorLang.AncestorOf, Length(AncestorLang.AncestorOf)+1);
2343:   AncestorLang.AncestorOf[Length(AncestorLang.AncestorOf)-1]:= Result;
2344:
2345:   Result.ChangedThisStep:= True;
2346:
2347:   Result.TotPopSize:= PopSize;
2348:   SetLength(Result.OccupCells, 1);
2349:   Result.OccupCells[0].IdCell:= Cell.IdCell;
2350:   Result.OccupCells[0].PosVec:= 0;
2351:   Result.OccupCells[0].MapCell:= Cell;
2352:   Result.OccupCells[0].Language:= Result;
2353:   Result.OccupCells[0].K:= 0;
2354:   Result.OccupCells[0].LangPopSizeInCell:= PopSize;
2355:   Result.OccupCells[0].DeltaLangPop:= 0;
2356:   Result.OccupCells[0].PatchTag:= 0;
2357:   Result.OccupCells[0].PrevPatchTag:= 0;
2358:   Result.OccupCells[0].BorderCell:= True;
2359:
2360:   SetLength(Cell.Languages, 1);
2361:   Cell.Languages[0]:= Result;
2362:   Cell.TotPopSize:= PopSize;
2363:   Cell.PopSaturated:= False;
2364:
2365:   If Cell.ColonizedOnce = False then
2366:    begin
2367:     Cell.ColonizedOnce:= True;
2368:     TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2369:    end;
2370:
2371:   SetLength(Result.Patches, 1);
2372:   SetLength(Result.Patches[0].Cells, 1);
2373:   Result.Patches[0].Cells[0]:= Cell;
2374:  end;
2375:
2376:
2377: Function TSim.CreateNewLanguage(AncestorLang: PLanguage; CellOccup: TPMapCellVec; Subsist
```

```
      ModeProp: TDblVector): PLanguage;
2378:  var
2379:   i, j, c: Integer;
2380:   TmpMapCellVec: TPLangMapCellVec;
2381:  begin
2382:   Result:= CreateNewLanguage(AncestorLang, SubsistModeProp);
2383:
2384:   SetLength(AncestorLang.AncestorOf, Length(AncestorLang.AncestorOf)+1);
2385:   AncestorLang.AncestorOf[Length(AncestorLang.AncestorOf)-1]:= Result;
2386:
2387:   TmpMapCellVec:= GetLangMapCellVec(AncestorLang, CellOccup);
2388:
2389:   SetLength(Result.OccupCells, Length(TmpMapCellVec));
2390:   For i:= 0 to Length(TmpMapCellVec)-1 do
2391:    begin
2392:     Result.TotPopSize:= Result.TotPopSize + TmpMapCellVec[i].LangPopSizeInCell;
2393:
2394:     Result.OccupCells[i].IdCell:= TmpMapCellVec[i].IdCell;
2395:     Result.OccupCells[i].PosVec:= i;
2396:     Result.OccupCells[i].MapCell:= TmpMapCellVec[i].MapCell;
2397:     Result.OccupCells[i].Language:= Result;
2398:     Result.OccupCells[i].K:= 0;
2399:     Result.OccupCells[i].LangPopSizeInCell:= TmpMapCellVec[i].LangPopSizeInCell;
2400:     Result.OccupCells[i].DeltaLangPop:= 0;
2401:     Result.OccupCells[i].PatchTag:= 0;
2402:     Result.OccupCells[i].PrevPatchTag:= 0;
2403:
2404:     AncestorLang.TotPopSize:= AncestorLang.TotPopSize - TmpMapCellVec[i].LangPopSizeInCel
      l;
2405:
2406:     // Temporary flag to be "deleted" below
2407:     TmpMapCellVec[i].IdCell:= -5;
2408:    end;
2409:
2410:   For i:= 0 to Length(CellOccup)-1 do
2411:    begin
2412:     For j:= 0 to Length(CellOccup[i].Languages)-1 do
2413:      begin
2414:       If CellOccup[i].Languages[j] = AncestorLang then
2415:        begin
2416:         CellOccup[i].Languages[j]:= Result;
2417:         Break;
2418:        end;
2419:      end;
2420:
2421:     If CellOccup[i].ColonizedOnce = False then
2422:      begin
2423:       CellOccup[i].ColonizedOnce:= True;
2424:       TotNumCellsColonizedOnce:= TotNumCellsColonizedOnce + 1;
2425:      end;
2426:    end;
2427:
2428:
2429:   SetLength(Result.Patches, 1);
2430:   Result.Patches[0].Cells:= CellOccup;
2431:   SetLength(Result.Patches[0].Cells, Length(CellOccup));
2432:
2433:   SetLength(Result.Patches[0].PatchSeparationTime, 1);
2434:   Result.Patches[0].PatchSeparationTime[0]:= 0;
2435:
2436:
2437:
2438:
2439:   c:= 0;
2440:   While c <= Length(AncestorLang.OccupCells)-1 do
2441:    begin
2442:     // If there is some... erase it from the language cell list
2443:     If AncestorLang.OccupCells[c].IdCell = -5 then
2444:      begin
2445:       For i:= c to Length(AncestorLang.OccupCells)-2 do
2446:        begin
2447:         AncestorLang.OccupCells[i]:= AncestorLang.OccupCells[i+1];
2448:        end;
2449:
```

```
2450:       SetLength(AncestorLang.OccupCells, Length(AncestorLang.OccupCells)-1);
2451:
2452:
2453:       If (AncestorLang.OccupCells = Nil) and (AncestorLang.IdLang <> -2) then
2454:        begin
2455:         TotNumLangs:= TotNumLangs - 1;
2456:         AncestorLang.ExtantLang:= False;
2457:         AncestorLang.ExtinctLang:= True;
2458:         AncestorLang.DiedAt:= TimeStep;
2459:        end;
2460:      end
2461:
2462:     Else
2463:       c:= c + 1;
2464:    end;
2465:  end;
2466:
2467: Function TSim.IntRND(Min, Max: Double): Integer;
2468:  begin
2469:   Result := Trunc(((Max - Min + 1) * (Random) ) + Min);
2470:  end;
2471:
2472: Function TSim.SetUpModel(ReSetMapData: Boolean = True): Boolean;
2473:  var
2474:   i, j, x: Integer;
2475:   TmpEnvMat: TDblMatrix;
2476:   TmpAreaMat: TDblMatrix;
2477:   TmpDblMat: TDblMatrix;
2478:   TmpStr: String;
2479:
2480:   c, K, n: Integer;
2481:   NewLang: PLanguage;
2482:   NewOccupCell: PLangMapCell;
2483:   SubsistModeProp: TDblVector;
2484:  begin
2485:
2486:   Result:= False;
2487:
2488:
2489:   FitStatsRep.nLangs:= 0;
2490:
2491:   FitStatsRep.KS:= 0;
2492:
2493:   FitStatsRep.RegResAvgRS.b:= Nil;
2494:   FitStatsRep.RegResAvgRS.bStd:= Nil;
2495:   FitStatsRep.RegResAvgRS.t:= Nil;
2496:   FitStatsRep.RegResAvgRS.SEb:= Nil;
2497:   FitStatsRep.RegResAvgRS.Est:= Nil;
2498:   FitStatsRep.RegResAvgRS.Res:= Nil;
2499:   FitStatsRep.RegResAvgRS.r2:= 0;
2500:   FitStatsRep.RegResAvgRS.r2Adj:= 0;
2501:   FitStatsRep.RegResAvgRS.F:= 0;
2502:   FitStatsRep.RegResAvgRS.Probr2:= 0;
2503:   FitStatsRep.RegResAvgRS.AIC:= 0;
2504:   FitStatsRep.RegResAvgRS.ESS:= 0;
2505:   FitStatsRep.RegResAvgRS.KL:= 0;
2506:   FitStatsRep.RegResAvgRS.JS:= 0;
2507:   FitStatsRep.RegResAvgRS.Hel:= 0;
2508:   FitStatsRep.RegResAvgRS.Euclid:= 0;
2509:   FitStatsRep.RegResAvgRS.L1:= 0;
2510:
2511:   FitStatsRep.RegResAvgRich.b:= Nil;
2512:   FitStatsRep.RegResAvgRich.bStd:= Nil;
2513:   FitStatsRep.RegResAvgRich.t:= Nil;
2514:   FitStatsRep.RegResAvgRich.SEb:= Nil;
2515:   FitStatsRep.RegResAvgRich.Est:= Nil;
2516:   FitStatsRep.RegResAvgRich.Res:= Nil;
2517:   FitStatsRep.RegResAvgRich.r2:= 0;
2518:   FitStatsRep.RegResAvgRich.r2Adj:= 0;
2519:   FitStatsRep.RegResAvgRich.F:= 0;
2520:   FitStatsRep.RegResAvgRich.Probr2:= 0;
2521:   FitStatsRep.RegResAvgRich.AIC:= 0;
2522:   FitStatsRep.RegResAvgRich.ESS:= 0;
2523:   FitStatsRep.RegResAvgRich.KL:= 0;
```

```
2524:   FitStatsRep.RegResAvgRich.JS:= 0;
2525:   FitStatsRep.RegResAvgRich.Hel:= 0;
2526:   FitStatsRep.RegResAvgRich.Euclid:= 0;
2527:   FitStatsRep.RegResAvgRich.L1:= 0;
2528:
2529:
2530:   TotNumCarryingCapParams:= Length(ModelParameters.CarryingCapParams);
2531:   If TotNumCarryingCapParams = 0 then
2532:    begin
2533:     ModelParameters.ModelFunction:= TMFPower;
2534:     TotNumCarryingCapParams:= 2;
2535:     SetLength(ModelParameters.CarryingCapParams, TotNumCarryingCapParams);
2536:     ModelParameters.CarryingCapParams[0]:= -8.0;
2537:     ModelParameters.CarryingCapParams[1]:= +2.6;
2538:    end;
2539:
2540:   SubsistModeProp:= Nil;
2541:   SetLength(SubsistModeProp, TotNumCarryingCapParams);
2542:   For i:= 0 to TotNumCarryingCapParams-1 do
2543:     SubsistModeProp[i]:= 0;
2544:   SubsistModeProp[0]:= 1;
2545:
2546:
2547:
2548:
2549:
2550:
2551:   MaxPopVec:= OpenSimple(ModelParameters.DataPath + ModelParameters.MaxPopSizeFile);
2552:
2553:
2554:     // Create 40 frequency bins for the distribution of language max pop size
2555:     SetLength(MaxPopDistrib, 22);
2556:
2557:     // Create the limits of each frequency class
2558:     MaxPopDistrib[0].Min:= 0;
2559:     MaxPopDistrib[0].Max:= 150;
2560:
2561:     For i:= 1 to 17 do
2562:       MaxPopDistrib[i].Max:= MaxPopDistrib[i-1].Max + 150;
2563:
2564:     For i:= 18 to Length(MaxPopDistrib)-1 do
2565:       MaxPopDistrib[i].Max:= MaxPopDistrib[i-1].Max + 500;
2566:
2567:   For i:= 1 to Length(MaxPopDistrib)-1 do
2568:    begin
2569:     MaxPopDistrib[i].MaxPopVec:= Nil;
2570:     MaxPopDistrib[i].Min:= MaxPopDistrib[i-1].Max;
2571:    end;
2572:
2573:   // Allocate each language size in the respective bin
2574:   For j:= 0 to Length(MaxPopVec)-1 do
2575:    begin
2576:     For i:= 0 to Length(MaxPopDistrib)-1 do
2577:      begin
2578:       If (MaxPopVec[j,0] > MaxPopDistrib[i].Min) and (MaxPopVec[j,0] <= MaxPopDistrib[i].
      Max) then
2579:        begin
2580:         SetLength(MaxPopDistrib[i].MaxPopVec, Length(MaxPopDistrib[i].MaxPopVec)+1);
2581:         MaxPopDistrib[i].MaxPopVec[Length(MaxPopDistrib[i].MaxPopVec)-1]:= MaxPopVec[j,0]
      ;
2582:
2583:         Break;
2584:        end;
2585:      end;
2586:    end;
2587:
2588:
2589:   // Calculate the frequency of each bin
2590:   For i:= 0 to Length(MaxPopDistrib)-1 do
2591:    begin
2592:     MaxPopDistrib[i].ObsCount:= Length(MaxPopDistrib[i].MaxPopVec);
2593:     MaxPopDistrib[i].ObsFrequency:= MaxPopDistrib[i].ObsCount / Length(MaxPopVec);
2594:
2595:     MaxPopDistrib[i].SimCount:= 0;
```

```
2596:     MaxPopDistrib[i].SimFrequency:= MaxPopDistrib[i].ObsFrequency;
2597:    end;
2598:
2599:
2600:
2601:   If ModelParameters.AreaParamFile <> '' then
2602:     TmpAreaMat:= OpenSimple(ModelParameters.DataPath + ModelParameters.AreaParamFile);
2603:
2604:   TmpEnvMat:= OpenSimple(ModelParameters.DataPath + ModelParameters.EnvDataFile);
2605:   TotNumEnvVars:= Length(TmpEnvMat[0]);
2606:
2607:   If ReSetMapData then
2608:    begin
2609:     MapShapeSmallRes:= ReadShapeFile(ModelParameters.DataPath + ModelParameters.MapShapeS
      mallResFile);
2610:     MapShape:= ReadShapeFile(ModelParameters.DataPath + ModelParameters.MapShapeFile);
2611:
2612:     TotNumCells:= Length(MapShape.GEOData.ShpPolygon.Polygon);
2613:
2614:     MapCells:= Nil;
2615:     SetLength(MapCells, TotNumCells);
2616:
2617:     CellSeq:= Nil;
2618:     SetLength(CellSeq, TotNumCells);
2619:     For i:= 0 to Length(MapCells)-1 do
2620:      begin
2621:       MapCells[i].IdCell:= i;
2622:
2623:       MapCells[i].Coord:= Centroid(MapShape.GEOData.ShpPolygon.Polygon[i].Part[0].Vertice
      s);
2624:
2625:       MapCells[i].Environment:= TmpEnvMat[i];
2626:
2627:       If TmpAreaMat <> Nil then
2628:         MapCells[i].Area:= TmpAreaMat[i,0];
2629:
2630:       SetLength(MapCells[i].Environment, Length(MapCells[i].Environment));
2631:
2632:       CellSeq[i]:= i;
2633:      end;
2634:
2635:     For i:= 0 to TotNumCells-1 do
2636:      begin
2637:       For j:= i+1 to TotNumCells-1 do
2638:        begin
2639:         If Distance(MapCells[i].Coord, MapCells[j].Coord) <= ModelParameters.NeighborDist
      ance then
2640:           begin
2641:            SetLength(MapCells[i].Neighbors, Length(MapCells[i].Neighbors)+1);
2642:            MapCells[i].Neighbors[Length(MapCells[i].Neighbors)-1]:= @MapCells[j];
2643:
2644:            SetLength(MapCells[j].Neighbors, Length(MapCells[j].Neighbors)+1);
2645:            MapCells[j].Neighbors[Length(MapCells[j].Neighbors)-1]:= @MapCells[i];
2646:           end;
2647:        end;
2648:      end;
2649:    end;
2650:
2651:  end;
2652:
2653:
2654: Function TSim.ReSetModel: Boolean;
2655:  var
2656:   i, j, x: Integer;
2657:   TmpEnvMat: TDblMatrix;
2658:   TmpAreaMat: TDblMatrix;
2659:   TmpStr: String;
2660:
2661:   K, n: Integer;
2662:   NewLang: PLanguage;
2663:   NewOccupCell: PLangMapCell;
2664:   SubsistModeProp: TDblVector;
2665:  begin
2666:   // Creates a new random seed
```

```
2667:   If ModelParameters.RandomSeed = -1 then
2668:     Randomize
2669:   Else
2670:     RandSeed:= ModelParameters.RandomSeed;
2671:
2672:   // Stores the random seed at the begining of the simulation
2673:   LstRandSeed:= RandSeed;
2674:
2675:   Result:= False;
2676:
2677:   Halted:= False;
2678:   TimeStep:= 0;
2679:   TotNumLangs:= 0;
2680:   TotNumLangsEver:= 0;
2681:
2682:
2683:   ReScaledRep.AvRangeSizePerCell:= Nil;
2684:   ReScaledRep.RichPerCell:= Nil;
2685:   ReScaledRep.RSFD:= Nil;
2686:
2687:   FitStatsRep.RegResAvgRS.b:= Nil;
2688:   FitStatsRep.RegResAvgRS.bStd:= Nil;
2689:   FitStatsRep.RegResAvgRS.t:= Nil;
2690:   FitStatsRep.RegResAvgRS.SEb:= Nil;
2691:   FitStatsRep.RegResAvgRS.Est:= Nil;
2692:   FitStatsRep.RegResAvgRS.Res:= Nil;
2693:
2694:   FitStatsRep.RegResAvgRich.b:= Nil;
2695:   FitStatsRep.RegResAvgRich.bStd:= Nil;
2696:   FitStatsRep.RegResAvgRich.t:= Nil;
2697:   FitStatsRep.RegResAvgRich.SEb:= Nil;
2698:   FitStatsRep.RegResAvgRich.Est:= Nil;
2699:   FitStatsRep.RegResAvgRich.Res:= Nil;
2700:
2701:   FitIndex:= 0;
2702:
2703:   j:= ModelParameters.NumInitLang;
2704:
2705:   For i:= 0 to Length(MaxPopDistrib)-1 do
2706:    begin
2707:     MaxPopDistrib[i].SimCount:= MaxPopDistrib[i].ObsCount;
2708:     MaxPopDistrib[i].SimFrequency:= MaxPopDistrib[i].ObsFrequency;
2709:    end;
2710:
2711:   Langs:= Nil;
2712:   SetLength(Langs, 1000000); // maximum number of languages in a singulation
2713:
2714:   LangSeq:= Nil;
2715:   SetLength(LangSeq, Length(Langs));
2716:   For i:= 0 to Length(Langs)-1 do
2717:    begin
2718:     Langs[i].ExtantLang:= False;
2719:     Langs[i].NeverExisted:= True;
2720:     Langs[i].ExtinctLang:= False;
2721:     Langs[i].RecentLang:= False;
2722:     Langs[i].TransformedLang:= False;
2723:     Langs[i].IdLang:= -99999;
2724:     Langs[i].BornAt:= -1;
2725:     Langs[i].DaughterOf:= -99999;
2726:     Langs[i].DiedAt:= -1;
2727:     LangSeq[i]:= i;
2728:    end;
2729:
2730:   For i:= 0 to Length(MapCells)-1 do
2731:    begin
2732:     MapCells[i].Languages:= Nil;
2733:     MapCells[i].TotPopSize:= 0;
2734:     MapCells[i].DeltaPop:= 0;
2735:     MapCells[i].PopSaturated:= False;
2736:     MapCells[i].ColonizedOnce:= False;
2737:    end;
2738:
2739:   TotNumCellsColonizedOnce:= 0;
2740:
```

```
2741:   WrkCells:= Nil;
2742:   WrkLangs:= Nil;
2743:
2744:   j:= 1;
2745:   For i:= 1 to j do
2746:    begin
2747:     NewLang:= CreateNewLanguage(Nil, SubsistModeProp);
2748:
2749:     // Seed Cell defined randomly
2750:     If ModelParameters.InitCell = -1 then
2751:       Self.InitCell:= IntRND(0, TotNumCells-1)
2752:     Else
2753:       Self.InitCell:= ModelParameters.InitCell;
2754:
2755:     While True do
2756:      begin
2757:       If (Self.InitCell > Length(MapCells)-1) or
2758:          (MapCells[Self.InitCell].Environment = Nil) or
2759:          (IsNaN(MapCells[Self.InitCell].Environment[0])) then
2760:        begin
2761:         Self.InitCell:= IntRND(0, TotNumCells-1)
2762:        end
2763:       Else
2764:         Break;
2765:      end;
2766:
2767:     K:= CalcCarryingCapacity(NewLang, @MapCells[Self.InitCell]);
2768:     n:= ModelParameters.InitPopSize;
2769:     NewOccupCell:= CellChangePop(NewLang, @MapCells[Self.InitCell], n);
2770:     NewOccupCell.K:= K;
2771:
2772:     NewLang.TotPopSize:= NewOccupCell.LangPopSizeInCell;
2773:
2774:     For x:= 2 to ModelParameters.PopPerLanguage do
2775:      begin
2776:       // Seed Cell defined randomly
2777:       If ModelParameters.InitCell = -1 then
2778:         Self.InitCell:= IntRND(0, TotNumCells-1)
2779:       Else
2780:         Self.InitCell:= ModelParameters.InitCell;
2781:
2782:       While True do
2783:        begin
2784:         If (Self.InitCell > Length(MapCells)-1) or
2785:            (MapCells[Self.InitCell].Environment = Nil) or
2786:            (IsNaN(MapCells[Self.InitCell].Environment[0])) then
2787:          begin
2788:           Self.InitCell:= IntRND(0, TotNumCells-1)
2789:          end
2790:         Else
2791:           Break;
2792:        end;
2793:
2794:       K:= CalcCarryingCapacity(NewLang, @MapCells[Self.InitCell]);
2795:       n:= ModelParameters.InitPopSize;
2796:       NewOccupCell:= CellChangePop(NewLang, @MapCells[Self.InitCell], n);
2797:       NewOccupCell.K:= K;
2798:
2799:       NewLang.TotPopSize:= NewLang.TotPopSize + NewOccupCell.LangPopSizeInCell;
2800:      end;
2801:    end;
2802:
2803:   TotNumPatches:= TotNumLangs;
2804:
2805:   TimeStep:= 0;
2806:
2807:   ModelSet:= True;
2808:
2809:
2810:
2811:
2812: {$IfNDef CONSOLE}
2813:   FrmMap:= TFrmMap.Create(FrmControl);
2814:   FrmMap.SetUpMap(MapShape);
```

```
2815:   FrmMap.Show;
2816:
2817:   FrmMap.UpdateMap;
2818:
2819: {$EndIf}
2820:
2821:
2822:
2823:
2824:
2825:
2826:   Result:= True;
2827:  end;
2828:
2829:
2830:
2831:
2832:
2833:
2834:
2835:
2836:
2837:
2838: Function TSim.ReScale: TModelReScaleResults;
2839:  var
2840:   ObsPresAbs: TDataSet;
2841:   NewPresAbs: TDataSet;
2842:   c, p, s, i, j, k, l: Integer;
2843:   F: TextFile;
2844:  begin
2845:   Try
2846:     ObsPresAbs.NumValues:= Nil;
2847:     ObsPresAbs.NumVarNames:= Nil;
2848:
2849:     SetLength(ObsPresAbs.NumVarNames, TotNumLangs);
2850:     SetLength(ObsPresAbs.NumValues, TotNumCells, TotNumLangs);
2851:
2852:     i:= 0;
2853:     For l:= 0 to Length(Langs)-1 do
2854:      begin
2855:       If (Langs[l].NeverExisted) or (Langs[l].ExtinctLang) or
2856:          (Langs[l].RecentLang) or (not Langs[l].ExtantLang) then
2857:        begin
2858:         Continue;
2859:        end;
2860:
2861:       ObsPresAbs.NumVarNames[i]:= 'Lg' + IntToStr(i);
2862:
2863:       For k:= 0 to Length(Langs[l].OccupCells)-1 do
2864:         If (Langs[l].OccupCells[k].IdCell >= 0) and (Langs[l].OccupCells[k].IdCell < TotN
      umCells) then
2865:           ObsPresAbs.NumValues[Langs[l].OccupCells[k].IdCell, i]:= 1;
2866:
2867:       i:= i + 1;
2868:      end;
2869:
2870:
2871:
2872:
2873:
2874:
2875:     SetLength(Result.AvRangeSizePerCell, Length(MapShapeSmallRes.GEOData.ShpPolygon.Polyg
      on));
2876:     SetLength(Result.RichPerCell, Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon));
2877:     SetLength(Result.RSFD, Length(ObsPresAbs.NumValues[0]));
2878:     SetLength(NewPresAbs.NumValues, Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon),
      Length(ObsPresAbs.NumValues[0]));
2879:     SetLength(NewPresAbs.NumVarNames, Length(ObsPresAbs.NumValues[0]));
2880:
2881:     For i:= 0 to Length(NewPresAbs.NumValues)-1 do
2882:      begin
2883:       Result.AvRangeSizePerCell[i]:= 0;
2884:       Result.RichPerCell[i]:= 0;
2885:       For j:= 0 to Length(NewPresAbs.NumValues[i])-1 do
```

```
2886:         NewPresAbs.NumValues[i,j]:= 0;
2887:       end;
2888:
2889:      For i:= 0 to Length(NewPresAbs.NumVarNames)-1 do
2890:       begin
2891:        NewPresAbs.NumVarNames[i]:= ObsPresAbs.NumVarNames[i];
2892:        Result.RSFD[i]:= 0;
2893:       end;
2894:
2895:      For c:= 0 to Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon)-1 do
2896:       begin
2897:        For p:= 0 to Length(MapShape.GEOData.ShpPolygon.Polygon)-1 do
2898:         begin
2899:          If PointInPolygon(MapShape.GEOData.ShpPolygon.Polygon[p].Centroid,
2900:                            MapShapeSmallRes.GEOData.ShpPolygon.Polygon[c].Part[0].Vertices
      ) then
2901:           begin
2902:            For s:= 0 to Length(NewPresAbs.NumVarNames)-1 do
2903:             begin
2904:              If ObsPresAbs.NumValues[p,s] = 1 then
2905:               begin
2906:                NewPresAbs.NumValues[c,s]:= 1;
2907:               end;
2908:             end;
2909:           end;
2910:         end;
2911:       end;
2912:
2913:
2914:
2915:
2916:      For s:= 0 to Length(NewPresAbs.NumVarNames)-1 do
2917:       begin
2918:        c:= 0;
2919:        For i:= 0 to Length(NewPresAbs.NumValues)-1 do
2920:          c:= c + Trunc(NewPresAbs.NumValues[i,s]);
2921:
2922:        Result.RSFD[s]:= c;
2923:       end;
2924:
2925:
2926:
2927:      // Calculate language richness in each cell
2928:      For i:= 0 to Length(NewPresAbs.NumValues)-1 do
2929:       begin
2930:        For s:= 0 to Length(NewPresAbs.NumVarNames)-1 do
2931:         begin
2932:          If NewPresAbs.NumValues[i,s] > 0 then
2933:            Result.RichPerCell[i]:= Result.RichPerCell[i] + 1;
2934:         end;
2935:       end;
2936:
2937:
2938:
2939:
2940:
2941:
2942:      // Calculate mean range size in each cell
2943:      For i:= 0 to Length(NewPresAbs.NumValues)-1 do
2944:       begin
2945:        c:= 0;
2946:        For s:= 0 to Length(NewPresAbs.NumVarNames)-1 do
2947:         begin
2948:          c:= c + Trunc(NewPresAbs.NumValues[i,s]);
2949:          Result.AvRangeSizePerCell[i]:= Result.AvRangeSizePerCell[i] + (NewPresAbs.NumValu
      es[i,s] * Result.RSFD[s]);
2950:         end;
2951:
2952:        If c > 0 then
2953:         begin
2954:          Result.AvRangeSizePerCell[i]:= Result.AvRangeSizePerCell[i] / c;
2955:         end
2956:        Else
2957:         begin
```

```
2958:         Result.AvRangeSizePerCell[i]:= 0;
2959:        end;
2960:      end;
2961:   Except
2962:     s:= 0;
2963:    End;
2964:  end;
2965:
2966:
2967: Procedure TSim.ClearParComboResults;
2968:  var
2969:   i, n: Integer;
2970:  begin
2971:   n:= Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon);
2972:
2973:   SetLength(ParComboResults.SumRichPerCell, n);
2974:   SetLength(ParComboResults.Sum2RichPerCell, n);
2975:   SetLength(ParComboResults.SumAvRangeSizePerCell, n);
2976:   SetLength(ParComboResults.Sum2AvRangeSizePerCell, n);
2977:
2978:   For i:= 0 to n-1 do
2979:    begin
2980:     ParComboResults.SumRichPerCell[i]:= 0;
2981:     ParComboResults.Sum2RichPerCell[i]:= 0;
2982:     ParComboResults.SumAvRangeSizePerCell[i]:= 0;
2983:     ParComboResults.Sum2AvRangeSizePerCell[i]:= 0;
2984:    end;
2985:
2986:   ParComboResults.AcumRSFD:= Nil;
2987:   ParComboResults.NumLangs:= Nil;
2988:
2989:   ParComboResults.RepCount:= 0;
2990:  end;
2991:
2992:
2993: Procedure TSim.StoreRepResults;
2994:  var
2995:   i, n: Integer;
2996:  begin
2997:   n:= Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon);
2998:
2999:   For i:= 0 to n-1 do
3000:    begin
3001:     ParComboResults.SumRichPerCell[i]:= ParComboResults.SumRichPerCell[i] + ReScaledRep.R
      ichPerCell[i];
3002:     ParComboResults.Sum2RichPerCell[i]:= ParComboResults.Sum2RichPerCell[i] + Sqr(ReScale
      dRep.RichPerCell[i]);
3003:     ParComboResults.SumAvRangeSizePerCell[i]:= ParComboResults.SumAvRangeSizePerCell[i] +
       ReScaledRep.AvRangeSizePerCell[i];
3004:     ParComboResults.Sum2AvRangeSizePerCell[i]:= ParComboResults.Sum2AvRangeSizePerCell[i]
       + Sqr(ReScaledRep.AvRangeSizePerCell[i]);
3005:    end;
3006:
3007:   n:= Length(ParComboResults.AcumRSFD);
3008:   SetLength(ParComboResults.AcumRSFD, Length(ParComboResults.AcumRSFD) + Length(ReScaledR
      ep.RSFD));
3009:
3010:   For i:= n to Length(ParComboResults.AcumRSFD)-1 do
3011:     ParComboResults.AcumRSFD[i]:= ReScaledRep.RSFD[i-n];
3012:
3013:   SetLength(ParComboResults.NumLangs, Length(ParComboResults.NumLangs)+1);
3014:   ParComboResults.NumLangs[Length(ParComboResults.NumLangs)-1]:= TotNumLangs;
3015:
3016:   ParComboResults.RepCount:= ParComboResults.RepCount + 1;
3017:  end;
3018:
3019:
3020:
3021: Function TSim.CalcRepFitStats: TModelFitStatsResults;
3022:  begin
3023:   Result.nLangs:= TotNumLangs;
3024:
3025:   If ModelParameters.ObsRichMap <> Nil then
3026:    begin
```

```
3027:     Try
3028:       // Regression between observed and predicted average range size in each cell
3029:       Result.RegResAvgRich:= Regression(ModelParameters.ObsRichMap, VectorToMatrix(ReScal
      edRep.RichPerCell));
3030:     Except
3031:
3032:      End;
3033:    end;
3034:
3035:   If ModelParameters.ObsAvgRSMap <> Nil then
3036:    begin
3037:     Try
3038:       // Regression between observed and predicted average range size in each cell
3039:       Result.RegResAvgRS:= Regression(ModelParameters.ObsAvgRSMap, VectorToMatrix(ReScale
      dRep.AvRangeSizePerCell));
3040:     Except
3041:
3042:      End;
3043:    end;
3044:
3045:   // KS test between observed and predicted range size frequency distributions
3046:   If ReScaledRep.RSFD <> Nil then
3047:     Result.KS:= KSTest(ReScaledRep.RSFD, ModelParameters.ObsRSFD);
3048:
3049:  end;
3050:
3051: Procedure TSim.CalcAvgRepFitStats;
3052:  var
3053:   i: Integer;
3054:  begin
3055:   If (FitStatsRep.nLangs = 0) or
3056:      IsNaN(FitStatsRep.RegResAvgRich.r2) then
3057:    begin
3058:     ActNumReps:= ActNumReps - 1;
3059:     Exit;
3060:    end;
3061:
3062:   AvgFitStats.nLangs:= AvgFitStats.nLangs + FitStatsRep.nLangs;
3063:
3064:   AvgFitStats.KS:= AvgFitStats.KS + FitStatsRep.KS;
3065:
3066:   AvgFitStats.RegResAvgRS.r2:= AvgFitStats.RegResAvgRS.r2 + FitStatsRep.RegResAvgRS.r2;
3067:   AvgFitStats.RegResAvgRS.r2Adj:= AvgFitStats.RegResAvgRS.r2Adj + FitStatsRep.RegResAvgRS
      .r2Adj;
3068:   AvgFitStats.RegResAvgRS.F:= AvgFitStats.RegResAvgRS.F + FitStatsRep.RegResAvgRS.F;
3069:   AvgFitStats.RegResAvgRS.Probr2:= AvgFitStats.RegResAvgRS.Probr2 + FitStatsRep.RegResAvg
      RS.Probr2;
3070:   AvgFitStats.RegResAvgRS.AIC:= AvgFitStats.RegResAvgRS.AIC + FitStatsRep.RegResAvgRS.AIC
      ;
3071:   AvgFitStats.RegResAvgRS.ESS:= AvgFitStats.RegResAvgRS.ESS + FitStatsRep.RegResAvgRS.ESS
      ;
3072:   AvgFitStats.RegResAvgRS.KL:= AvgFitStats.RegResAvgRS.KL + FitStatsRep.RegResAvgRS.KL;
3073:   AvgFitStats.RegResAvgRS.JS:= AvgFitStats.RegResAvgRS.JS + FitStatsRep.RegResAvgRS.JS;
3074:   AvgFitStats.RegResAvgRS.Hel:= AvgFitStats.RegResAvgRS.Hel + FitStatsRep.RegResAvgRS.Hel
      ;
3075:   AvgFitStats.RegResAvgRS.Euclid:= AvgFitStats.RegResAvgRS.Euclid + FitStatsRep.RegResAvg
      RS.Euclid;
3076:   AvgFitStats.RegResAvgRS.L1:= AvgFitStats.RegResAvgRS.L1 + FitStatsRep.RegResAvgRS.L1;
3077:
3078:   AvgFitStats.RegResAvgRich.r2:= AvgFitStats.RegResAvgRich.r2 + FitStatsRep.RegResAvgRich
      .r2;
3079:   AvgFitStats.RegResAvgRich.r2Adj:= AvgFitStats.RegResAvgRich.r2Adj + FitStatsRep.RegResA
      vgRich.r2Adj;
3080:   AvgFitStats.RegResAvgRich.F:= AvgFitStats.RegResAvgRich.F + FitStatsRep.RegResAvgRich.F
      ;
3081:   AvgFitStats.RegResAvgRich.Probr2:= AvgFitStats.RegResAvgRich.Probr2 + FitStatsRep.RegRe
      sAvgRich.Probr2;
3082:   AvgFitStats.RegResAvgRich.AIC:= AvgFitStats.RegResAvgRich.AIC + FitStatsRep.RegResAvgRi
      ch.AIC;
3083:   AvgFitStats.RegResAvgRich.ESS:= AvgFitStats.RegResAvgRich.ESS + FitStatsRep.RegResAvgRi
      ch.ESS;
3084:   AvgFitStats.RegResAvgRich.KL:= AvgFitStats.RegResAvgRich.KL + FitStatsRep.RegResAvgRich
      .KL;
3085:   AvgFitStats.RegResAvgRich.JS:= AvgFitStats.RegResAvgRich.JS + FitStatsRep.RegResAvgRich
```

```
.JS;
3086:   AvgFitStats.RegResAvgRich.Hel:= AvgFitStats.RegResAvgRich.Hel + FitStatsRep.RegResAvgRi
    ch.Hel;
3087:   AvgFitStats.RegResAvgRich.Euclid:= AvgFitStats.RegResAvgRich.Euclid + FitStatsRep.RegRe
    sAvgRich.Euclid;
3088:   AvgFitStats.RegResAvgRich.L1:= AvgFitStats.RegResAvgRich.L1 + FitStatsRep.RegResAvgRich
    .L1;
3089:  end;
3090:
3091: Function TSim.CalcRepFitIndex: Double;
3092:  var
3093:   t: Double;
3094:  begin
3095:   If IsNan(FitStatsRep.RegResAvgRich.r2) then
3096:    begin
3097:     Result:= NaN;
3098:     Exit;
3099:    end;
3100:
3101:   // Fit index is an arbitrary value that combines multiple measures of fit
3102:   // Values closer to zero indicate best fit
3103:   FitIndex:= 0;
3104:
3105:   // r2 betwee observed and estimated language richness
3106:   FitIndex:= FitIndex + ((1-FitStatsRep.RegResAvgRich.r2) * 0.5);
3107:
3108:   // Number of languages
3109:   t:= Abs(Length(ModelParameters.ObsRSFD) - FitStatsRep.nLangs);  // Absoluute difference
    in number of langauges
3110:   t:= t / Length(ModelParameters.ObsRSFD);                        // Proportion between d
    ifference and number of languages
3111:
3112:   // Calculate fit index
3113:   FitIndex:= FitIndex + (t * 0.5);
3114:
3115:
3116:
3117:   FitIndex:= 1 - FitIndex;
3118:   Result:= FitIndex;
3119:  end;
3120:
3121: Function TSim.CalcThreadFitIndex: Double;
3122:  var
3123:   i, n: Integer;
3124:   t: Double;
3125:   v: TDblVector;
3126:  begin
3127:   If ActNumReps = 0 then
3128:    begin
3129:     Result:= 0;
3130:     Exit;
3131:    end;
3132:
3133:   AvgFitStats.nLangs:= AvgFitStats.nLangs / ActNumReps;
3134:
3135:   AvgFitStats.KS:= AvgFitStats.KS / ActNumReps;
3136:
3137:   // The average of r2 of AvRS
3138:   AvgFitStats.RegResAvgRS.r2:= AvgFitStats.RegResAvgRS.r2 / ActNumReps;
3139:   AvgFitStats.RegResAvgRS.r2Adj:= AvgFitStats.RegResAvgRS.r2Adj / ActNumReps;
3140:   AvgFitStats.RegResAvgRS.F:= AvgFitStats.RegResAvgRS.F / ActNumReps;
3141:   AvgFitStats.RegResAvgRS.Probr2:= AvgFitStats.RegResAvgRS.Probr2 / ActNumReps;
3142:   AvgFitStats.RegResAvgRS.AIC:= AvgFitStats.RegResAvgRS.AIC / ActNumReps;
3143:   AvgFitStats.RegResAvgRS.ESS:= AvgFitStats.RegResAvgRS.ESS / ActNumReps;
3144:   AvgFitStats.RegResAvgRS.KL:= AvgFitStats.RegResAvgRS.KL / ActNumReps;
3145:   AvgFitStats.RegResAvgRS.JS:= AvgFitStats.RegResAvgRS.JS / ActNumReps;
3146:   AvgFitStats.RegResAvgRS.Hel:= AvgFitStats.RegResAvgRS.Hel / ActNumReps;
3147:   AvgFitStats.RegResAvgRS.Euclid:= AvgFitStats.RegResAvgRS.Euclid / ActNumReps;
3148:   AvgFitStats.RegResAvgRS.L1:= AvgFitStats.RegResAvgRS.L1 / ActNumReps;
3149:
3150:
3151:
3152:   // Average of r2 of the replicates
3153:   AvgFitStats.RegResAvgRich.r2:= AvgFitStats.RegResAvgRich.r2 / ActNumReps;
```

```
3154:   AvgFitStats.RegResAvgRich.r2Adj:= AvgFitStats.RegResAvgRich.r2Adj / ActNumReps;
3155:   AvgFitStats.RegResAvgRich.F:= AvgFitStats.RegResAvgRich.F / ActNumReps;
3156:   AvgFitStats.RegResAvgRich.Probr2:= AvgFitStats.RegResAvgRich.Probr2 / ActNumReps;
3157:   AvgFitStats.RegResAvgRich.AIC:= AvgFitStats.RegResAvgRich.AIC / ActNumReps;
3158:   AvgFitStats.RegResAvgRich.ESS:= AvgFitStats.RegResAvgRich.ESS / ActNumReps;
3159:   AvgFitStats.RegResAvgRich.KL:= AvgFitStats.RegResAvgRich.KL / ActNumReps;
3160:   AvgFitStats.RegResAvgRich.JS:= AvgFitStats.RegResAvgRich.JS / ActNumReps;
3161:   AvgFitStats.RegResAvgRich.Hel:= AvgFitStats.RegResAvgRich.Hel / ActNumReps;
3162:   AvgFitStats.RegResAvgRich.Euclid:= AvgFitStats.RegResAvgRich.Euclid / ActNumReps;
3163:   AvgFitStats.RegResAvgRich.L1:= AvgFitStats.RegResAvgRich.L1 / ActNumReps;
3164:   t:= AvgFitStats.RegResAvgRich.r2;
3165:
3166:
3167:
3168:   // OR
3169:
3170:
3171:   // Average the language richness of the replicates, and calculate one single r2
3172:   n:= Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon);
3173:   SetLength(v, n);
3174:
3175:   For i:= 0 to n-1 do
3176:     ParComboResults.SumRichPerCell[i]:= ParComboResults.SumRichPerCell[i] / ParComboResul
      ts.RepCount;
3177:
3178:   AvgFitStats.RegResAvgRich:= Regression(ModelParameters.ObsRichMap, VectorToMatrix(ParCo
      mboResults.SumRichPerCell));
3179:   t:= AvgFitStats.RegResAvgRich.r2;
3180:
3181:
3182:   For i:= 0 to n-1 do
3183:     ParComboResults.SumAvRangeSizePerCell[i]:= ParComboResults.SumAvRangeSizePerCell[i] /
       ParComboResults.RepCount;
3184:
3185:   AvgFitStats.RegResAvgRS:= Regression(ModelParameters.ObsAvgRSMap, VectorToMatrix(ParCom
      boResults.SumAvRangeSizePerCell));
3186:
3187:
3188:
3189:
3190:
3191:   // Fit index is an arbitrary value that combines multiple measures of fit
3192:
3193:   // Values closer to zero indicate best fit
3194:   FitIndex:= 0;
3195:
3196:   // r2 between observed and estimated richness map
3197:   FitIndex:= FitIndex + ((1-t) * 0.50);
3198:
3199:
3200:
3201:
3202:
3203:   // Number of languages
3204:   t:= Abs(Length(ModelParameters.ObsRSFD) - AvgFitStats.nLangs);  // Absoluute difference
       in number of langauges
3205:   t:= t / Length(ModelParameters.ObsRSFD);                        // Proportion between d
      ifference and number of languages
3206:
3207:
3208:
3209:   FitIndex:= FitIndex + (t * 0.5);
3210:
3211:
3212:   FitIndex:= 1 - FitIndex;
3213:   Result:= FitIndex;
3214:  end;
3215:
3216:
3217:
3218:
3219: Procedure TSim.ExportParComboEstResults;
3220:  var
3221:   i, n: Integer;
```

```
3222:   m, sd: Double;
3223:   FileOutSim: TextFile;
3224:  begin
3225:   n:= Length(MapShapeSmallRes.GEOData.ShpPolygon.Polygon);
3226:
3227:   AssignFile(FileOutSim, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName
      + ' - xEstRichMap.txt');
3228:   ReWrite(FileOutSim);
3229:   Write(FileOutSim,
3230:         'AvRich' + #9 +
3231:         'StdDevRich');
3232:
3233:   For i:= 0 to n-1 do
3234:    begin
3235:     WriteLn(FileOutSim);
3236:
3237:     m:= ParComboResults.SumRichPerCell[i];
3238:     Sd:= ((ParComboResults.Sum2RichPerCell[i]*ParComboResults.RepCount) / ParComboResults
      .RepCount) - Sqr(m);
3239:     Sd:= Sqrt(Sd);
3240:
3241:     Write(FileOutSim,
3242:           m:0:5, #9,
3243:           Sd:0:5);
3244:    end;
3245:
3246:   CloseFile(FileOutSim);
3247:
3248:
3249:
3250:
3251:
3252:
3253:
3254:   AssignFile(FileOutSim, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName
      + ' - xEstAvRSFDMap.txt');
3255:   ReWrite(FileOutSim);
3256:   Write(FileOutSim,
3257:         'AvRSFD' + #9 +
3258:         'StdDevRSFD');
3259:
3260:   For i:= 0 to n-1 do
3261:    begin
3262:     WriteLn(FileOutSim);
3263:
3264:     m:= ParComboResults.SumAvRangeSizePerCell[i];
3265:     Sd:= ((ParComboResults.Sum2AvRangeSizePerCell[i]*ParComboResults.RepCount) / ParCombo
      Results.RepCount) - Sqr(m);
3266:     Sd:= Sqrt(Sd);
3267:
3268:     Write(FileOutSim,
3269:           m:0:5, #9,
3270:           Sd:0:5);
3271:    end;
3272:
3273:   CloseFile(FileOutSim);
3274:
3275:
3276:
3277:
3278:
3279:
3280:
3281:   AssignFile(FileOutSim, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName
      + ' - xEstRSFD.txt');
3282:   ReWrite(FileOutSim);
3283:   Write(FileOutSim,
3284:         'RangeSize');
3285:
3286:   n:= Length(ParComboResults.AcumRSFD);
3287:   For i:= 0 to n-1 do
3288:    begin
3289:     WriteLn(FileOutSim);
3290:
```

```
3291:     Write(FileOutSim,
3292:           ParComboResults.AcumRSFD[i]:0:0);
3293:    end;
3294:
3295:   CloseFile(FileOutSim);
3296:
3297:
3298:
3299:
3300:
3301:   AssignFile(FileOutSim, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName
      + ' - xEstN.txt');
3302:   ReWrite(FileOutSim);
3303:   Write(FileOutSim,
3304:         'NumLangs');
3305:
3306:   n:= Length(ParComboResults.NumLangs);
3307:   For i:= 0 to n-1 do
3308:    begin
3309:     WriteLn(FileOutSim);
3310:
3311:     Write(FileOutSim,
3312:           ParComboResults.NumLangs[i]:0:0);
3313:    end;
3314:
3315:   CloseFile(FileOutSim);
3316:  end;
3317:
3318:
3319:
3320:
3321: Procedure TSim.Randomize;
3322:  var
3323:    Counter: Int64;
3324:  begin
3325:    if QueryPerformanceCounter(Counter) then
3326:      RandSeed := Counter
3327:    else
3328:      RandSeed := GetTickCount;
3329:  end;
3330:
3331: Function TSim.Random: Extended;
3332:  const
3333:    two2neg32: double = ((1.0/$10000) / $10000);
3334:  var
3335:    Temp: Longint;
3336:    F: Extended;
3337:  begin
3338:    Temp := RandSeed * $08088405 + 1;
3339:    RandSeed := Temp;
3340:    F := Int64(Cardinal(Temp));
3341:    Result := F * two2neg32;
3342:  end;
3343:
3344: Function TSimParallel.Execute(ModelParameters: TModelParameters;
3345:                               nReps: Integer = 120;
3346:                               nThreads: Integer = -1): Double;
3347:  var
3348:   i, j, c, d: Integer;
3349:   s: Double;
3350:   SysInfo: SYSTEM_INFO;
3351:   RepsPerThreads: Integer;
3352:   TmpEnvMat: TDblMatrix;
3353:   TmpAreaMat: TDblMatrix;
3354:  begin
3355:   Self.ModelParameters:= ModelParameters;
3356:
3357:   If nThreads <= 0 then
3358:    begin
3359:     GetSystemInfo(SysInfo);
3360:     nThreads:= SysInfo.dwNumberOfProcessors;
3361:    end;
3362:
3363:   If nThreads > nReps then
```

```
3364:     nThreads:= nReps;
3365:
3366:   RepsPerThreads:= nReps div nThreads;
3367:
3368:
3369:
3370: // Open dataset only once, as it is slow to be re-opened by each thread
3371:   If MapCells = Nil then
3372:    begin
3373:     If ModelParameters.AreaParamFile <> '' then
3374:       TmpAreaMat:= OpenSimple(ModelParameters.DataPath + ModelParameters.AreaParamFile);
3375:
3376:     TmpEnvMat:= OpenSimple(ModelParameters.DataPath + ModelParameters.EnvDataFile);
3377:     TotNumEnvVars:= Length(TmpEnvMat[0]);
3378:
3379:     MapShape:= ReadShapeFile(ModelParameters.DataPath + ModelParameters.MapShapeFile);
3380:     MapShapeSmallRes:= ReadShapeFile(ModelParameters.DataPath + ModelParameters.MapShapeS
      mallResFile);
3381:
3382:     TotNumCells:= Length(MapShape.GEOData.ShpPolygon.Polygon);
3383:
3384:     MapCells:= Nil;
3385:     SetLength(MapCells, TotNumCells);
3386:
3387:     CellSeq:= Nil;
3388:     SetLength(CellSeq, TotNumCells);
3389:     For i:= 0 to Length(MapCells)-1 do
3390:      begin
3391:       MapCells[i].IdCell:= i;
3392:
3393:       MapCells[i].Coord:= Centroid(MapShape.GEOData.ShpPolygon.Polygon[i].Part[0].Vertice
      s);
3394:
3395:       MapCells[i].Environment:= TmpEnvMat[i];
3396:
3397:       If TmpAreaMat <> Nil then
3398:         MapCells[i].Area:= TmpAreaMat[i,0];
3399:
3400:       SetLength(MapCells[i].Environment, Length(MapCells[i].Environment));
3401:
3402:       CellSeq[i]:= i;
3403:      end;
3404:
3405:     For i:= 0 to TotNumCells-1 do
3406:      begin
3407:       For j:= i+1 to TotNumCells-1 do
3408:        begin
3409:         If Distance(MapCells[i].Coord, MapCells[j].Coord) <= ModelParameters.NeighborDist
      ance then
3410:           begin
3411:            SetLength(MapCells[i].Neighbors, Length(MapCells[i].Neighbors)+1);
3412:            MapCells[i].Neighbors[Length(MapCells[i].Neighbors)-1]:= @MapCells[j];
3413:
3414:            SetLength(MapCells[j].Neighbors, Length(MapCells[j].Neighbors)+1);
3415:            MapCells[j].Neighbors[Length(MapCells[j].Neighbors)-1]:= @MapCells[i];
3416:           end;
3417:        end;
3418:      end;
3419:    end;
3420: //
3421:
3422:
3423:
3424:   If (ModelParameters.OutPutFolder <> '') and (ModelParameters.SimName <> '') then
3425:    begin
3426:     InitializeCriticalSection(FileOutReplicateWrite);
3427:
3428:     AssignFile(FileOutReplicate, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.
      SimName + ' - Reps.txt');
3429:     If FileExists(ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName + ' - R
      eps.txt') then
3430:       Append(FileOutReplicate)
3431:     Else
3432:      begin
```

```
3433:       ReWrite(FileOutReplicate);
3434:       Write(FileOutReplicate,
3435:             'RndSeed' + #9 +
3436:             'InitCell' + #9);
3437:
3438:       for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
3439:         Write(FileOutReplicate,
3440:               'CarCapParam' + IntToStr(i+1) + #9);
3441:
3442:       Write(FileOutReplicate,
3443:             'FitIdx' + #9 +
3444:             'nLangs' + #9 +
3445:             'r2 Rich' + #9 +
3446:             'r2 RS' + #9 +
3447:             'D');
3448:      end;
3449:    end;
3450:
3451:
3452:
3453:
3454:
3455:
3456:   SetLength(SimVec, nThreads);
3457:   SetLength(HandleVec, nThreads);
3458:   For i:= 0 to nThreads-1 do
3459:    begin
3460:     SimVec[i]:= TSim.Create(True, True, True, True);
3461:     SimVec[i].FreeOnTerminate:= True;
3462:     SimVec[i].Priority:= tpLower;
3463:     SimVec[i].SimId:= i;
3464:     HandleVec[i]:= SimVec[i].Handle;
3465:
3466:     SimVec[i].TotNumReps:= RepsPerThreads;
3467:
3468:     SimVec[i].ModelParameters:= ModelParameters;
3469:
3470: // Duplicate map data to each thread
3471:     SimVec[i].MapShape:= MapShape;
3472:     SimVec[i].MapShapeSmallRes:= MapShapeSmallRes;
3473:
3474:     If TTextRec(FileOutReplicate).Mode <> 0 then
3475:      begin
3476:       SimVec[i].FileOutReplicate:= @FileOutReplicate;
3477:       SimVec[i].FileOutReplicateWrite:= @FileOutReplicateWrite;
3478:      end;
3479:
3480:     SimVec[i].TotNumCells:= TotNumCells;
3481:     SimVec[i].TotNumEnvVars:= TotNumEnvVars;
3482:     SetLength(SimVec[i].MapCells, TotNumCells);
3483:     SetLength(SimVec[i].CellSeq, TotNumCells);
3484:     For c:= 0 to TotNumCells-1 do
3485:      begin
3486:       SimVec[i].MapCells[c].IdCell:= MapCells[c].IdCell;
3487:       SimVec[i].MapCells[c].IdCell:= MapCells[c].IdCell;
3488:       SimVec[i].MapCells[c].Coord.X:= MapCells[c].Coord.X;
3489:       SimVec[i].MapCells[c].Coord.Y:= MapCells[c].Coord.Y;
3490:       SimVec[i].MapCells[c].Area:= MapCells[c].Area;
3491:       SimVec[i].CellSeq[c]:= CellSeq[c];
3492:
3493:       SetLength(SimVec[i].MapCells[c].Environment, TotNumEnvVars);
3494:       For d:= 0 to TotNumEnvVars-1 do
3495:         SimVec[i].MapCells[c].Environment[d]:= MapCells[c].Environment[d];
3496:
3497:       SetLength(SimVec[i].MapCells[c].Neighbors, Length(MapCells[c].Neighbors));
3498:       For d:= 0 to Length(MapCells[c].Neighbors)-1 do
3499:         SimVec[i].MapCells[c].Neighbors[d]:= @SimVec[i].MapCells[MapCells[c].Neighbors[d]
      .IdCell];
3500:      end;
3501:
3502:     SimVec[i].SetUpModel(False);
3503:    end;
3504:
3505:   // If the total number of replicates is not a multiple of the number of threads
```

```
3506:   j:= RepsPerThreads * nThreads;
3507:   i:= 0;
3508:   while j <> nReps do
3509:    begin
3510:     // Add additional reps to the first threads, until the total number of reps add up to
      nReps
3511:     SimVec[i].TotNumReps:= SimVec[i].TotNumReps + 1;
3512:     i:= i + 1;
3513:     j:= j + 1;
3514:    end;
3515:
3516:   For i:= 0 to nThreads-1 do
3517:     SimVec[i].Start;
3518:
3519:   WaitForMultipleObjects(nThreads, Pointer(HandleVec), True, INFINITE);
3520:
3521:
3522:
3523:
3524:
3525:
3526:
3527:
3528:
3529:
3530:
3531:   // Average the fit index calculated by each thread
3532:   Result:= 0;
3533:   s:= 0;
3534:
3535:   AvgFitStats.nLangs:= 0;
3536:
3537:   AvgFitStats.KS:= 0;
3538:   AvgFitStats.RegResAvgRS.r2:= 0;
3539:   AvgFitStats.RegResAvgRS.r2Adj:= 0;
3540:
3541:   AvgFitStats.RegResAvgRich.r2:= 0;
3542:   AvgFitStats.RegResAvgRich.r2Adj:= 0;
3543:
3544:   For i:= 0 to Length(SimVec)-1 do
3545:    begin
3546:     AvgFitStats.nLangs:= AvgFitStats.nLangs + (SimVec[i].AvgFitStats.nLangs * SimVec[i].T
      otNumReps);
3547:
3548:     AvgFitStats.KS:= AvgFitStats.KS + (SimVec[i].AvgFitStats.KS * SimVec[i].TotNumReps);
3549:     AvgFitStats.RegResAvgRS.r2:= AvgFitStats.RegResAvgRS.r2 + (SimVec[i].AvgFitStats.RegR
      esAvgRS.r2 * SimVec[i].TotNumReps);
3550:     AvgFitStats.RegResAvgRS.r2Adj:= AvgFitStats.RegResAvgRS.r2Adj + (SimVec[i].AvgFitStat
      s.RegResAvgRS.r2Adj * SimVec[i].TotNumReps);
3551:
3552:     AvgFitStats.RegResAvgRich.r2:= AvgFitStats.RegResAvgRich.r2 + (SimVec[i].AvgFitStats.
      RegResAvgRich.r2 * SimVec[i].TotNumReps);
3553:     AvgFitStats.RegResAvgRich.r2Adj:= AvgFitStats.RegResAvgRich.r2Adj + (SimVec[i].AvgFit
      Stats.RegResAvgRich.r2Adj * SimVec[i].TotNumReps);
3554:
3555:     Result:= Result + (SimVec[i].FitIndex * SimVec[i].TotNumReps);
3556:     s:= s + SimVec[i].TotNumReps;
3557:
3558:     //SimVec[i].Free;
3559:    end;
3560:
3561:   AvgFitStats.nLangs:= AvgFitStats.nLangs / s;
3562:
3563:   AvgFitStats.KS:= AvgFitStats.KS / s;
3564:   AvgFitStats.RegResAvgRS.r2:= AvgFitStats.RegResAvgRS.r2 / s;
3565:   AvgFitStats.RegResAvgRS.r2Adj:= AvgFitStats.RegResAvgRS.r2Adj / s;
3566:
3567:   AvgFitStats.RegResAvgRich.r2:= AvgFitStats.RegResAvgRich.r2 / s;
3568:   AvgFitStats.RegResAvgRich.r2Adj:= AvgFitStats.RegResAvgRich.r2Adj / s;
3569:
3570:   Result:= Result / s;
3571:   FitIndex:= Result;
3572:
3573:
```

```
3574:
3575:
3576:
3577:
3578: //
3579: // These are the final results for each parameter combination, averaged among replicates
      of a thread and among threads
3580: //  WriteLn('');
3581:   WriteLn('');
3582:   Write('  --->   ',
3583:         ModelParameters.InitCell:0, '  ');
3584:
3585:   for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
3586:     Write(ModelParameters.CarryingCapParams[i]:0:3, '  ');
3587:
3588:   Write(FitIndex:0:3, '  ',
3589:         AvgFitStats.nLangs:0:0, '  ',
3590:         AvgFitStats.RegResAvgRich.r2:0:3, '  ',
3591:         AvgFitStats.RegResAvgRS.r2:0:3, '  ',
3592:         AvgFitStats.KS:0:3);
3593:
3594:
3595:
3596:
3597:   // These are the final results, averaged among replicates, for a given parameter combin
      ation
3598:   AssignFile(FileOutSim, ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName
      + ' - Averages.txt');
3599:   If FileExists(ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimName + ' - Ave
      rages.txt') then
3600:     Append(FileOutSim)
3601:   Else
3602:    begin
3603:     ReWrite(FileOutSim);
3604:
3605:     Write(FileOutSim, 'CarCapFunc' + #9);
3606:
3607:     for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
3608:       Write(FileOutSim,
3609:             'CarCapParam' + IntToStr(i+1) + #9);
3610:
3611:     Write(FileOutSim,
3612:           'FitIdx' + #9 +
3613:           'nLangs' + #9 +
3614:           'r2 Rich' + #9 +
3615:           'r2 RS' + #9 +
3616:           'D');
3617:    end;
3618:
3619:   WriteLn(FileOutSim);
3620:
3621:   if ModelParameters.ModelFunction = TMFPower then
3622:     Write(FileOutSim, 'Power', #9) Else
3623:   if ModelParameters.ModelFunction = TMFLogistic then
3624:     Write(FileOutSim, 'Logistic', #9) Else
3625:   if ModelParameters.ModelFunction = TMFExponential then
3626:     Write(FileOutSim, 'Exp', #9);
3627:
3628:   for i := 0 to Length(ModelParameters.CarryingCapParams)-1 do
3629:     Write(FileOutSim,
3630:           ModelParameters.CarryingCapParams[i]:0:7, #9);
3631:
3632:   Write(FileOutSim,
3633:         FitIndex:0:5, #9,
3634:         AvgFitStats.nLangs:0:3, #9,
3635:         AvgFitStats.RegResAvgRich.r2:0:5, #9,
3636:         AvgFitStats.RegResAvgRS.r2:0:5, #9,
3637:         AvgFitStats.KS:0:5);
3638:
3639:   CloseFile(FileOutSim);
3640: //
3641:
3642:
3643:   // This should never happen!
```

```
3644:   If IsNan(Result) then
3645:    begin
3646:     CloseFile(FileOutReplicate);
3647:     WriteLn('Result is NaN');
3648:     ReadLn(Result);
3649:    end;
3650:
3651:
3652:
3653:
3654:
3655:
3656:   SimVec:= Nil;
3657:
3658:   If TTextRec(FileOutReplicate).Mode <> 0 then
3659:    begin
3660:     DeleteCriticalSection(FileOutReplicateWrite);
3661:     CloseFile(FileOutReplicate);
3662:    end;
3663:  end;
3664:
3665: Function TSim.Sample(List: TIntVector; n: Integer = -1; WithReplacement: Boolean = False)
     : TIntVector;
3666:  Function IntRND(Min, Max: TFloat): Integer;
3667:   begin
3668:    Result := Trunc(((Max - Min + 1) * (Random) ) + Min);
3669:   end;
3670:  var
3671:   i: Integer;
3672:   SeqOrder: TDblMatrix;
3673:  begin
3674:   If List = Nil then
3675:    begin
3676:     Result:= Nil;
3677:     Exit;
3678:    end;
3679:
3680:   If n = -1 then
3681:     n:= Length(List);
3682:
3683:   SetLength(Result, n);
3684:   If WithReplacement then
3685:    begin
3686:     For i:= 0 to n-1 do
3687:      begin
3688:       Result[i]:= List[IntRND(0,Length(List)-1)];
3689:      end;
3690:    end
3691:   Else
3692:    begin
3693:     If n > Length(List) then
3694:       raise Exception.Create('Impossible to sample without replacement more than the popu
     lation size');
3695:     SetLength(SeqOrder, Length(List), 2);
3696:     For i:= 0 to Length(SeqOrder)-1 do
3697:      begin
3698:       SeqOrder[i,0]:= i;
3699:       SeqOrder[i,1]:= Random;
3700:      end;
3701:     SeqOrder:= QuickSort(SeqOrder, 1);
3702:
3703:     SetLength(Result, n);
3704:     For i:= 0 to n-1 do
3705:      begin
3706:       Result[i]:= List[Trunc(SeqOrder[i,0])];
3707:      end;
3708:    end;
3709:  end;
3710:
3711: Function TSimParallel.ExecToGibbsSamp(ProposalParams: TDblVector): Double;
3712:  var
3713:   i: Integer;
3714:  begin
3715:   for i:= 0 to Length(ProposalParams)-1 do
```

```
3716:     ModelParameters.CarryingCapParams[i]:= ProposalParams[i];
3717:
3718:   Execute(ModelParameters,
3719:           tnReps,
3720:           tnThreads);
3721:
3722:   Result:= FitIndex;
3723:  end;
3724:
3725:
3726:
3727:
3728:
3729:
3730:
3731:
3732:
3733:
3734:
3735:
3736:
3737:
3738:
3739:
3740:
3741:
3742:
3743:
3744:
3745:
3746:
3747:
3748:
3749:
3750:
3751:
3752:
3753:
3754:
3755:
3756:
3757:
3758:
3759:
3760:
3761:
3762:
3763:
3764: //MCMC search only used in GEB paper of 2017
3765: Procedure TSimParallel.SearchGibbs(ModelParameters: TModelParameters;
3766:                                    DisturbFuncs: TDisturbFuncs;
3767:                                    var MCMCChain: TChain;
3768:                                    nReps: Integer = 120;
3769:                                    nThreads: Integer = -1);
3770:  var
3771:   StartValues: TDblVector;
3772:   Priors: TPriors;
3773:   i: Integer;
3774:  begin
3775:   SetLength(DisturbFuncs, Length(ModelParameters.CarryingCapParams));
3776:
3777:   if ModelParameters.ModelFunction = TMFPower then
3778:    begin
3779:     // Carrying Capacity Scale Parameter
3780:     DisturbFuncs[0].RndDistrib:= TRndNorm;
3781:     SetLength(DisturbFuncs[0].SampFuncPars, 2);
3782:     DisturbFuncs[0].SampFuncPars[0]:= 0;
3783:     DisturbFuncs[0].SampFuncPars[1]:= 0.05;
3784:
3785:     // Carrying Capacity Slope-Rate Parameter
3786:     DisturbFuncs[1].RndDistrib:= TRndNorm;
3787:     SetLength(DisturbFuncs[1].SampFuncPars, 2);
3788:     DisturbFuncs[1].SampFuncPars[0]:= 0;
3789:     DisturbFuncs[1].SampFuncPars[1]:= 0.05;
```

```
3790:    end Else
3791:
3792:   if ModelParameters.ModelFunction = TMFLogistic then
3793:    begin
3794:     // x0 is the value of precipitation in which the curve is in its midpoint (point of i
      nflection of the curve)
3795:     DisturbFuncs[0].RndDistrib:= TRndNorm;
3796:     SetLength(DisturbFuncs[0].SampFuncPars, 2);
3797:     DisturbFuncs[0].SampFuncPars[0]:= 0;
3798:     DisturbFuncs[0].SampFuncPars[1]:= 10;
3799:
3800:     // s is the steepness of the curve ; because s is a very small number, we estimate lo
      g10(s)
3801:     DisturbFuncs[1].RndDistrib:= TRndNorm;
3802:     SetLength(DisturbFuncs[1].SampFuncPars, 2);
3803:     DisturbFuncs[1].SampFuncPars[0]:= 0;
3804:     DisturbFuncs[1].SampFuncPars[1]:= 0.01;
3805:
3806:     // m is the maximum carrying capacity (saturation level), in units of carrying capaci
      ty
3807:     DisturbFuncs[2].RndDistrib:= TRndNorm;
3808:     SetLength(DisturbFuncs[2].SampFuncPars, 2);
3809:     DisturbFuncs[2].SampFuncPars[0]:= 0;
3810:     DisturbFuncs[2].SampFuncPars[1]:= 1;
3811:    end Else
3812:
3813:   if ModelParameters.ModelFunction = TMFExponential then
3814:    begin
3815:     // a is a scaler
3816:     DisturbFuncs[0].RndDistrib:= TRndNorm;
3817:     SetLength(DisturbFuncs[0].SampFuncPars, 2);
3818:     DisturbFuncs[0].SampFuncPars[0]:= 0;
3819:     DisturbFuncs[0].SampFuncPars[1]:= 0.01;
3820:
3821:     // b is a multiplier of precipitation, which is in the exponent of euler number (natu
      ral exponent)
3822:     DisturbFuncs[1].RndDistrib:= TRndNorm;
3823:     SetLength(DisturbFuncs[1].SampFuncPars, 2);
3824:     DisturbFuncs[1].SampFuncPars[0]:= 0;
3825:     DisturbFuncs[1].SampFuncPars[1]:= 0.005;
3826:    end;
3827:
3828:   SetLength(Priors, Length(ModelParameters.CarryingCapParams));
3829:
3830:   Self.ModelParameters:= ModelParameters;
3831:   tnReps:= nReps;
3832:   tnThreads:= nThreads;
3833:
3834:   SetLength(StartValues, Length(ModelParameters.CarryingCapParams));
3835:   for i:= 0 to Length(ModelParameters.CarryingCapParams)-1 do
3836:     StartValues[i]:= ModelParameters.CarryingCapParams[i];
3837:
3838:   GibbsSamp:= TGibbs.Create(StartValues,             // Initial values for the MCMC
3839:                             ExecToGibbsSamp,
3840:                             Priors,
3841:                             DisturbFuncs,
3842:                             ModelParameters.OutPutFolder + '!Sim' + ModelParameters.SimNa
      me + ' - MCMC Chain.txt',
3843:                             100000,
3844:                             0);
3845:  end;
3846:
3847: end.
```