PIANOS testing document

Group Linja

Helsinki 7th September 2005

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

Course

581260 Software Engineering Project (6 cr)

Project Group

Joonas Kukkonen
Marja Hassinen
Eemil Lagerspetz

Client

Marko Salmenkivi

Project Masters

Juha Taina
Vesa Vainio (Instructor)

Homepage

`http://www.cs.helsinki.fi/group/linja`

Contents

1 Preface

This document outlines the testing practices used in the PIANOS project. First the test strategy is described and the material for the unit tests given. Then the rest of the document details the tests. See Appendix 1. for a report on test code coverage (also available in detailed browseable html in [coverage]).

1.1 Version history

Version	Date	Modifications
1.0	30.08.2005	Full-featured version
1.1	31.08.2005	Reviewed and corrected final

1.2 Naming conventions

Fortran: Refers to the fortran programming language, version 90/95 specifically. **fortran** refers to the whole Fortran family, and **FORTRAN** refers to FORTRAN/77 specifically.

Proposal: A new value cancidate obtained from the proposal distribution.

Frequency function is used to refer to a frequency function or a density function when it is irrelevant which one there really is, that is, when it's irrelevant whether the distribution is discrete or continuous.

Variable is used to refer a variable or a parameter when it's irrelevant which one there really is.

Generator: Used to refer to the modules of the software that write out the specific executable Program that carries out the simulation for a given simulation model.

Program: The program that is run to simulate the problem model. Synonym: generated program.

Entity: A data structure of the Generator representing a repetitive structure (indexing structure) of variables. For example $alpha_i$, x_i both are part of the Entity indexed with i.

Variable group: All variables of a group, that is $alpha_i$ for all i.

Parser: The ComputationalModelParser class used for reading the input files for the Generator.

2 Testing strategy

2.1 Unit tests

Testing will follow the bottom-up testing strategy. That is, the phases of the testing process are:

1. Test the data structures
2. Test the FortranWriter
3. Test the ComputationalModelParser
4. Assume that the Parser and the FortranWriter work correctly and use them when testing other Generator classes

The Parser's unit tests are automated. They can be run with the ParserUnitTest class found in the tests package.

Unit test results for most of the generator classes (package PIANOS.generator) can not be validated automatically. In these cases, the Fortran code written by the classes is inspected instead.

Code coverage of all the tests can be found in a report on the PIANOS CD-ROM. This report is compiled by running the MasterTest class and the PIANOS Generator with a single file parameter and eight separate files. See the PIANOS manual for details on running the Generator, See Appendix 1. for a summary of the code coverage report.

2.2 Integration tests

The Parser is tested to produce data structures with valid information. This can be proved by unit tests of the Parser that use the data structures, and system tests that use the parser.

Because most of the generator parts function independently integration tests for them are deemed unnecessary. System and unit tests verify their correctness.

2.3 System tests

System tests are run with these models:

1. the simple bird model
2. the simple bird model with spatial dependence
3. the lip cancer model
4. the bird model

Simulations on these models have been run. It is difficult to determine whether the results are correct since previous results for the models on the used algorithm don't exist or are difficult to obtain. See the next chapter for details.

3 Test models

Since the size of a model is not limited, there are infinitely many models. So it's clearly impossible to test the software with all possible models. Instead the testing should be done by using a representative subset of all the models.

This section defines some models to be used for testing the software. Each model should be tested with no missing values and with missing values.

3.1 Semantically correct models

This section represents models which can be used when testing the Generator and also the Fortran program it produces.

3.1.1 The simple bird model

Model description:

```
REAL alpha ~ continuous_uniform(-1.0, 1.0)

ENTITY square, "squares.txt"
{
    INTEGER northerness(1)
}

ENTITY bird, ""
{

}

ENTITY squarebird, "observations.txt", combines(square, bird)
{
    REAL p = EXP(alpha * northerness) / (1 + EXP(alpha * northerness))
    INTEGER x ~ user_bernoulli(p)
    INTEGER obs(*) ~ user_defined_points(x)
}
```

Proposal distributions and strategies:

```
?? proposal distributions

? alpha
continuous_uniform(-1.0, 1.0)

? x
discrete_uniform(0, 1)
```

Initial values:

```
?? initial values

? alpha
0.5

? x 1:21 1:4
0 0 1 1
0 0 1 1
0 0 1 1
1 0 1 1
1 0 1 1
1 0 1 1
1 0 1 1
0 0 1 1
0 0 1 0
0 0 0 0
1 0 0 0
1 0 1 0
0 0 1 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 1 0
0 0 1 0
0 0 1 1
```

The data for obs is the same matrix as the initial values of x.

Expected results when running the Fortran program: Alpha is negative.

3.1.2 The simple bird model with spatial dependence

Model description:

```
REAL alpha ~ continuous_uniform(-1.0, 1.0)

ENTITY square, "squares.txt"
{
    INTEGER northerness(1)
}

ENTITY bird, ""
{

}
```

```
ENTITY squarebird, "observations.txt", combines(square, bird)
{
    INTEGER q = SUM(&x)
    REAL p = EXP(alpha * northerness) / (1 + EXP(alpha * northerness))
    INTEGER x ~ user_bernoulli(p)
    INTEGER obs(*) ~ user_defined_points(x)
}
```

Other input files stay the same.

Expected results when running the Fortran program: Alpha is negative.

3.1.3 The lip cancer model

Model description:

```
REAL alpha ~ user_normal(0, 100)
REAL sigma ~ user_gamma(1, 1)

ENTITY county, "tests/cancer/lips.data", "tests/cancer/lips.neighbours"
{
REAL x(3)
REAL expected(2)
REAL n = COUNT(&b)
REAL q = SUM(&b)
REAL t = q/n
REAL b ~ user_normal(t, n)
REAL mu = EXP(LOG(expected) + alpha * x / 10 + sigma * b)
INTEGER obs(1) ~ poisson(mu)
}
```

Proposal distributions and strategies:

```
?? proposal distributions

? alpha
user_normal(0, 10) RW

? sigma
continuous_uniform(0.001, 5)

? b
continuous_uniform(-10.0, 10.0)
```

Expected results when running the Fortran program: alpha is 0.37 +/- 0.11, sigma is 0.69 +/- 0.12.

3.1.4 The bird model

Model description:

```
REAL alpha ~ user_gamma(0.25, 0.5)
REAL tau ~ continuous_uniform(0.5, 10)
REAL gamma ~ continuous_uniform(-1.0, 1.0)

ENTITY species, "species.data"
{
    REAL a ~ user_normal(alpha, tau)
    INTEGER waterbird(1)
}

ENTITY cell, "gridcell.data", "neighbours.data"
{
    REAL q = SUM(&x)
    REAL waterpercentage(4)
    INTEGER researchgrade(7)
    INTEGER centeredcoordinate(3)
}

ENTITY observations, "obs.data", combines (cell, species)
{
    REAL p = EXP(waterbird*a*waterpercentage + gamma*centeredcoordinate + 0.25 * q)
    /(1 +  EXP(waterbird*a*waterpercentage + gamma*centeredcoordinate + 0.25 * q))
    # 0.25 could be any other constant. defines spatial weighting.

    INTEGER x ~ user_bernoulli(p)
    INTEGER observation(*) ~ user_defined_points2(x, researchgrade)
}
```

Proposal distributions and strategies, initial values and expected results were not given.

3.2 Other models

This section describes modes which have no sensible semantics but can be used when testing the Generator. These models are useful because it is possible for them to contain whatever dependencies that are useful for testing purposes while semantically correct models are more restricted.

3.2.1 A model with all kinds of stochastic dependencies

This model includes all possible stochastic dependencies as a single-parameter version, that is, each distribution has only one parameter.

Model description:

```
# 1) global parameter with no dependencies
INTEGER a1 ~ user_dist_1(3)

# 2) a2 -> b2, both global
INTEGER a2 ~ user_dist_2(4)
INTEGER b2 ~ user_dist_2-(a2)

# 3) one-dim parameter with no dependencies
ENTITY e3, ""
{
    INTEGER a3 ~ user_dist_3(5)
}

# 4) a4 -> b4, a4 global and b4 one-dim
INTEGER a4 ~ user_dist_4(6)
ENTITY e4, ""
{
    INTEGER b4 ~ user_dist_4-(a4)
}

# 5) a5 -> b5, both one-dim
ENTITY e5, ""
{
    INTEGER a5 ~ user_dist_5(7)
    INTEGER b5 ~ user_dist_5-(a5)
}

# 6) two-dim parameter with no dependencies
ENTITY e6y, ""
{
}
ENTITY e6x, ""
{
}
ENTITY e6, "", combines(e6y, e6x)
{
    INTEGER a6 ~ user_dist_6(5)
}

# 7) a7 -> b7, a7 global and b7 two-dim
INTEGER a7 ~ user_dist_7(8)
ENTITY e7y, ""
{
}
ENTITY e7x, ""
{
}
ENTITY e7, "", combines(e7y, e7x)
{
    INTEGER b7 ~ user_dist_7-(a7)
}
```

```
# 8) a8 -> b8, a8 one-dim and b8 two-dim (case y)
ENTITY e8y, ""
{
    INTEGER a8 ~ user_dist_8(9)
}
ENTITY e8x, ""
{
}
ENTITY e8, "", combines(e8y, e8x)
{
    INTEGER b8 ~ user_dist_8-(a8)
}

# 9) a9 -> b9, a9 one-dim and b9 two-dim (case x)
ENTITY e9y, ""
{
}
ENTITY e9x, ""
{
    INTEGER a9 ~ user_dist_9(10)
}
ENTITY e9, "", combines(e9y, e9x)
{
    INTEGER b9 ~ user_dist_9-(a9)
}

# 9) a10 -> b10, both two-dim
ENTITY e10y, ""
{
}
ENTITY e10x, ""
{
}
ENTITY e10, "", combines(e10y, e10x)
{
    INTEGER a10 ~ user_dist_10(10)
    INTEGER b10 ~ user_dist_10-(a10)
}
```

3.2.2 A model with all kinds of functional non-spatial dependencies

This model includes all possible combinations of functional and stochastic dependencies but no spatial dependencies. It is a single-parameter version, that is, each distribution has only one parameter.

Model description:

```
# 1) a1 - - > b1
REAL a1 ~ user_dist_1(1)
REAL b1 = a1 + 1
```

```
# 2) a2 - - > b2 ----> c2
REAL a2 ~ user_dist_2(2)
REAL b2 = a2 - 2
REAL c2 ~ user_dist_2-(b2)

# 4) | a4 - - > b4 |
ENTITY e4, ""
{
    REAL a4 ~ user_dist_4(4)
    REAL b4 = a4 / 4
}

# 5) | a5 - - > b5 ----> c5 |
ENTITY e5, ""
{
    REAL a5 ~ user_dist_2(5)
    REAL b5 = EXP(a5)
    REAL c5 ~ user_dist_5-(b5)
}

# 7) | | a7 - - > b7 | |
ENTITY e7y, ""
{
}
ENTITY e7x, ""
{
}
ENTITY e7, "", combines(e7y, e7x)
{
    REAL a7 ~ user_dist_7(7)
    REAL b7 = a7 ** 2
}

# 8) | | a8 - - > b8 ----> c8 | |
ENTITY e8y, ""
{
}
ENTITY e8x, ""
{
}
ENTITY e8, "", combines(e8y, e8x)
{
    REAL a8 ~ user_dist_2(8)
    REAL b8 = EXP(a8)
    REAL c8 ~ user_dist_8-(b8)
}

# 10) a10 - - > | b10 ----> c10 |
REAL a10 ~ user_dist_10(10)
ENTITY e10, ""
{
    REAL b10 = SIN(a10)
    REAL c10 ~ user_dist_10-(b10)
}
```

```
# 12) a12 - - > b12 ----> | c12 |
REAL a12 ~ user_dist_12(12)
REAL b12 = 3 - a12
ENTITY e12, ""
{
    REAL c12 ~ user_dist_12-(b12)
}

# 14) | a14 - - > b14 ----> | c14 | | case y
ENTITY e14y, ""
{
    REAL a14 ~ user_dist_14(14)
    REAL b14 = 18 / a14
}
ENTITY e14x, ""
{
}
ENTITY e14, "", combines(e14y, e14x)
{
    REAL c14 ~ user_dist_14-(b14)
}

# 16) | a16 - - > b16 ----> | c16 | | case x
ENTITY e16y, ""
{
}
ENTITY e16x, ""
{
    REAL a16 ~ user_dist_16(16)
    REAL b16 = 18 / a16
}
ENTITY e16, "", combines(e16y, e16x)
{
    REAL c16 ~ user_dist_16-(b16)
}

# 18) | a18 - - > | b18 ----> c18 | | case y
ENTITY e18y, ""
{
    REAL a18 ~ user_dist_18(18)
}
ENTITY e18x, ""
{
}
ENTITY e18, "", combines(e18y, e18x)
{
    REAL b18 = a18
    REAL c18 ~ user_dist_18-(b18)
}

# 20) | a20 - - > | b20 ----> c20 | | case x
ENTITY e20y, ""
{
```

```
}
ENTITY e20x, ""
{
    REAL a20 ~ user_dist_20(20)
}
ENTITY e20, "", combines(e20y, e20x)
{
    REAL b20 = a20
    REAL c20 ~ user_dist_20-(b20)
}

# 22) a22 - - > | | b22 ----> c22 | |
REAL a22 ~ user_dist_22(22)
REAL b22 = 5 * a22
ENTITY e22y, ""
{
}
ENTITY e22x, ""
{
}
ENTITY e22, "", combines(e22y, e22x)
{
    REAL c22 ~ user_dist_22-(b22)
}

# 24) a24 - - > | b24 ----> | c24 | | case y
REAL a24 ~ user_dist_24(24)
ENTITY e24y, ""
{
    REAL b24 = a24 + 1**2
}
ENTITY e24x, ""
{
}
ENTITY e24, "", combines(e24y, e24x)
{
    REAL c24 ~ user_dist_24-(b24)
}

# 26) a26 - - > | b26 ----> | c26 | | case x
REAL a26 ~ user_dist_26(26)
ENTITY e26y, ""
{
}
ENTITY e26x, ""
{
    REAL b26 = a26 + 1**2
}
ENTITY e26, "", combines(e26y, e26x)
{
    REAL c26 ~ user_dist_26-(b26)
}
```

Models with all kinds of spatial dependencies

Since a model can include only one spatial entity, the different test cases must be different models.

Model description (simple1):

```
# simple, functional spatial variable and the referenced one
# one-dimensional
ENTITY simple01 , "" , "tests/model_all_spat/file.txt" {
INTEGER dep01 ~ discrete_uniform(0, 1)
INTEGER s01 = SUM(& dep01)
}
```

Model description (simple2):

```
ENTITY s , "", "tests/model_all_spat/file.txt" {
}
ENTITY n , "" {
}
# two-dimensional (spatial = Y)
ENTITY simple02 , "" , combines(s, n) {
INTEGER dep02 ~ discrete_uniform(0, 1)
INTEGER s02 = SUM(& dep02)
}
# two-dimensional (spatial = X)
ENTITY simple03 , "" , combines(n, s) {
INTEGER dep03 ~ discrete_uniform(0, 1)
INTEGER s03 = SUM(& dep03)
}
```

Model description (cross1):

```
# cross, two functional spatial variables, COUNT and SUM, referencing
# different stochastic variables, one-dimensional:

ENTITY cross01 , "" , "tests/model_all_spat/file.txt" {
INTEGER depsc01 ~ discrete_uniform(0, 1)
REAL depcc01 ~ continuous_uniform(0, 1)
REAL sc01 = SUM(& depsc01)
REAL cc01 = COUNT(& depcc01)
}
```

Model description (cross2):

```
ENTITY s , "", "tests/model_all_spat/file.txt" {
}
```

```
ENTITY n , "" {
}
# two-dimensional (spatial = Y)
ENTITY cross02 , "" , combines(s, n) {
INTEGER depsc02 ~ discrete_uniform(0, 1)
REAL depcc02 ~ continuous_uniform(0, 1)
REAL sc02 = SUM(& depsc02)
REAL cc02 = COUNT(& depcc02)
}
# two-dimensional (spatial = X)
ENTITY cross03 , "" , combines(n, s) {
INTEGER depsc03 ~ discrete_uniform(0, 1)
REAL depcc03 ~ continuous_uniform(0, 1)
REAL sc03 = SUM(& depsc03)
REAL cc03 = COUNT(& depcc03)
}
```

Model description (dia1):

```
# dia, the diamond-shaped structure,
# two functional spatial variables, COUNT and SUM, referencing the same
# stochastic variable, and a functional average variable

# one-dimensional
ENTITY dia01 , "" , "tests/model_all_spat/file.txt" {
INTEGER depd01 ~ discrete_uniform(0, 1)
REAL sd01 = SUM(& depd01)
REAL cd01 = COUNT(& depd01)
REAL avg1 = sd01 / cd01
}
```

Model description (dia2):

```
ENTITY s , "", "tests/model_all_spat/file.txt" {
}
ENTITY n , "" {
}
# two-dimensional (spatial = Y)
ENTITY dia02 , "" , combines(s, n) {
INTEGER depd02 ~ discrete_uniform(0, 1)
INTEGER sd02 = SUM(& depd02)
INTEGER cd02 = COUNT(& depd02)
REAL avg2 = REAL(sd02) / cd02
}

# two-dimensional (spatial = X)
ENTITY dia03 , "" , combines(n, s) {
INTEGER depd03 ~ discrete_uniform(0, 1)
INTEGER sd03 = SUM(& depd03)
INTEGER cd03 = COUNT(& depd03)
REAL avg3 = REAL(sd03) / cd03
}
```

Model description (crossc1):

```
# crossc, two functional spatial SUM variables referencing
# different stochastic variables, and an
# combining functional variable

# one-dimensional
ENTITY crossc01 , "" , "tests/model_all_spat/file.txt" {
INTEGER depscc01 ~ discrete_uniform(0, 1)
REAL depscc02 ~ continuous_uniform(0, 1)
REAL scc01 = SUM(& depscc01)
REAL scc02 = SUM(& depscc02)
REAL comb1 = scc01 + scc02
}
```

Model description (crossc2):

```
ENTITY s , "", "tests/model_all_spat/file.txt" {
}
ENTITY n , "" {
}
#two-dimensional (spatial = Y)
ENTITY crossc02 , "" , combines(s, n) {
INTEGER depscc03 ~ discrete_uniform(0, 1)
REAL depscc04 ~ continuous_uniform(0, 1)
REAL scc03 = SUM(& depscc03)
REAL scc04 = SUM(& depscc04)
REAL comb2 = scc03 + scc04
}
# two-dimensional (spatial = X)
ENTITY crossc03 , "" , combines(n, s) {
INTEGER depscc05 ~ discrete_uniform(0, 1)
REAL depscc06 ~ continuous_uniform(0, 1)
REAL scc05 = SUM(& depscc05)
REAL scc06 = COUNT(& depscc06)
REAL comb3 = scc05 + scc06
}
```

4 Unit tests

4.1 Data structures

Classes Variable, Entity and Equation are deemed trivial enough to not require testing. Their source code will be reviewed.

4.1.1 Distribution subclasses

The Distribution class defines many get and set methods which are not overridden in it's subclasses. Since these methods are simple the are only code reviewed. All distribution subclasses have been tested. Subclass testing is divided to hard coded distribution classes (BetaDistribution, BinomialDistribution, ContinuousUniformDistribution, DiscreteUniformDistribution, PoissonDistribution) and to testing of UserDefinedDistribution class. The UserDefinedDistribution class is tested with the DistributionFactory.

Hard coded distributions, methods to be tested:

All of these methods return Fortran code.

- getIntroduction - Returns the introduction of NAG library functions
- getFreqCode - Returns the Fortran call for frequency function
- getGenCode - Returns the Fortran call for generator function

Strategy:

An object is created for each class. The methods are called for each class. GetIntroduction does not take in any parameters and it can be considered static. Only thing to check from getIntroduction is that it does return valid Fortran code. Methods getFreqCode and getGenCode take a String array as a parameter. The array must be a certain length depending on the class. getFreqCode and getGenCode are called only with valid parameters. The validity of the Fortran code returned by the methods is checked by reviewing and compiling it.

Distribution subclasses, DistributionSkeleton and DistributionFactory have a tester class DistributionTester. The test can be run by calling a public method run().

Expected results:

BetaDistribution

```
The distribution has 2 parameters

Testing introduction

EXTERNAL G01EEF
EXTERNAL G05FEF

Generator function:

ifail = 0
CALL G05FEF(Param1, Param2, SIZE(Result variable), Result variable, ifail)
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to generate random numbers from beta
& distribution, parameters: a = ', Param1, 'b = ', Param2
STOP
END IF

Frequency function:

ifail = -1
CALL G01EEF(Point, Param1, Param2, 1.0_dp, temp_real, temp_real, Result variable,
& ifail)
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to calculate frequency of the beta
& distribution, parameters: a = ', Param1, 'b = ', Param2
STOP
END IF
```

BinomialDistribution

```
The distribution has 2 parameters

Testing introduction

INTEGER G05EYF
EXTERNAL G05EYF
EXTERNAL G05EDF
EXTERNAL G01BJF

Generator function:

ifail = 0
ALLOCATE(binomial_reference(1 : NINT(20 + 20 * SQRT(Param1 * Param2 *
& (1 - Param2)))))
CALL G05EDF(Param1, Param2, binomial_reference, SIZE(binomial_reference), ifail)
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to generate random numbers from binomial
& distribution, parameters: n = ', Param1, ' p = ', Param2
STOP
```

```
END IF
DO k=1, SIZE(Result variable)
Result variable(k)=G05EYF(binomial_reference, SIZE(binomial_reference))
END DO

Frequency function:

ifail=0
CALL G01BJF(Param1, Param2, Point, temp_real, temp_real, Result variable, ifail)
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to calculate frequency of the binomial
& distribution, parameters: n = ',Param1, 'p = ',Param2
STOP
END IF
```

ContinuousUniformDistribution

```
The distribution has 2 parameters

Testing introduction

EXTERNAL G05FAF

Generator function:

CALL G05FAF(Param1, Param2, SIZE(Result variable), Result variable)

Frequency function:

IF (Point < Param1 .OR. Point > Param2) THEN
Result variable = 0
ELSE
Result variable = 1.0 / (Param2 - (Param1))
END IF
```

DiscreteUniformDistribution

```
The distribution has 2 parameters

Testing introduction

INTEGER G05DYF
EXTERNAL G05DYF

Generator function:

DO k=1, SIZE(Result variable)
Result variable(k) = G05DYF(Param1, Param2)
END DO
```

```
Frequency function:

IF (Point < Param1 .OR. Point > Param2) THEN
Result variable = 0
ELSE
Result variable = 1.0 / (REAL(Param2) - (REAL(Param1)) + 1.0)
END IF
```

PoissonDistribution

```
The distribution has 1 parameters

Introduction

EXTERNAL G01BKF
EXTERNAL G05ECF
INTEGER G05EYF
EXTERNAL G05EYF

Generator function:

ifail = 0
ALLOCATE(poisson_reference(1 : NINT(20 + 20 * SQRT(Param1))))
CALL G05ECF(Param1, poisson_reference, SIZE(poisson_reference), ifail)
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to generate random numbers from poisson
& distribution, parameters: mu = ', Param1
STOP
END IF
DO k=1, SIZE(Result variable)
Result variable(k)=G05EYF(poisson_reference, SIZE(poisson_reference))
END DO

Frequency function:

ifail=0
IF (Param1 > 1.0D+6) THEN
CALL G01BKF( 1.0D+6, Point, temp_real, temp_real, Result variable, ifail)
ELSE
CALL G01BKF(Param1, Point, temp_real, temp_real, Result variable, ifail)
END IF
IF (ifail /= 0) THEN
WRITE (*, *) 'Error when trying to calculate frequency of the poisson
& distribution, parameters: mu = ', Param1
STOP
END IF
```

4.1.2 DistributionSkeleton

The DistributionSkeleton only contains a constructor, set and get methods. These methods are simple, so they are not tested except by reading the source code.

4.1.3 DistributionFactory

The DistributionFactory has two functionalities. First one is its constructor, which reads a user-defined distribution file. The constructor creates a DistributionSkeleton object for each distribution defined in the file. The second functionality is returning of correct Distribution objects.

Methods to be tested:

- DistributionFactory - the constructor
- getDistribution - returns a Distribution object

Strategy:

The getDistribution method is tested first. It's called with the names of hard coded distributions. The class name of returned objects is checked to be correct. The constructor is tested next. Multiple DistributionFactory objects are created with different user distribution files. After the creation of an object, the getDistribution method is called with the names of the user defined functions. Some files contain correct user defined distributions, while some hold incorrect ones. The UserDefinedDistribution class is tested at the same time. getFreqCode and getGenCode are called for UserDefinedDistribution objects.

Expected results:

A DistributionFactory object is expected to return correct hard coded distributions when called with correct parameters. The constructor is expected to accept valid user defined distributions and reject incorrect ones. getDistribution method is also expected to return correct user defined distributions. The UserDefinedDistribution objects are expected to return correct Fortran code.

Valid definitions, user defined distribution file: 'user1.txt'.

```
! user_alpha 2 real integer .true.
! user_BETA 2 real integer .true.
! user_GaMmA 2 real integer .true.
! USER_delta 2 real integer .true.
! UsEr_EtA 2 real integer .true.
! user_theta 2 REAL INTEGER .true.
! user_iota 2 ReAl InTeGeR .true.
! user_kappa 1 integer .true.
! user_lambda 3 real integer real .true.
! user_mu 2 real integer .false.
! user_nu 2 integer real .false.
! user_xhi 2 real integer .FALSE.
```

```
! user_tau 2 integer real .TRUE.
!         user_phi       2       real    integer       .true.
!user_sigma 2 integer real .true.
! user_omega 2 REAL INTEGER .TRUE.
```

Expected results of the user defined distribution file: 'user1.txt'.

```
Testing ! user_alpha 2 real integer .true.

Call name is user_AlpHa
Generation code:
CALL user_alpha_gen(Param1, Param2, Result)

Frequency code:
CALL user_alpha_freq(Param1, Param2, Point, Result)


Testing ! user_BETA 2 real integer .true.

Call name is user_beta
Generation code:
CALL user_beta_gen(Param1, Param2, Result)

Frequency code:
CALL user_beta_freq(Param1, Param2, Point, Result)


Testing ! user_GaMmA 2 real integer .true.

Call name is user_GAMMA
Generation code:
CALL user_gamma_gen(Param1, Param2, Result)

Frequency code:
CALL user_gamma_freq(Param1, Param2, Point, Result)


Testing ! USER_delta 2 real integer .true.

Call name is user_delta
Generation code:
CALL user_delta_gen(Param1, Param2, Result)

Frequency code:
CALL user_delta_freq(Param1, Param2, Point, Result)


Testing ! UsEr_EtA 2 real interger .true.

Call name is user_ETA
Generation code:
CALL user_eta_gen(Param1, Param2, Result)
```

```
Frequency code:
CALL user_eta_freq(Param1, Param2, Point, Result)


Testing ! user_theta 2 REAL INTEGER .true.

Call name is user_TheTa
Generation code:
CALL user_theta_gen(Param1, Param2, Result)

Frequency code:
CALL user_theta_freq(Param1, Param2, Point, Result)


Testing ! user_iota 2 ReAl InTeGeR .true.

Call name is user_iota
Generation code:
CALL user_iota_gen(Param1, Param2, Result)

Frequency code:
CALL user_iota_freq(Param1, Param2, Point, Result)


Testing ! user_kappa 1 integer .true.

Call name is user_kappa
Generation code:
CALL user_kappa_gen(Param1, Result)

Frequency code:
CALL user_kappa_freq(Param1, Point, Result)


Testing ! user_lambda 3 real interger real .true.

Call name is user_lambda
Generation code:
CALL user_lambda_gen(Param1, Param2, Param3, Result)

Frequency code:
CALL user_lambda_freq(Param1, Param2, Param3, Point, Result)


Testing ! user_mu 2 real interger .false.

Call name is user_MU
Generation code:
Generator function was not found for MU


Testing ! user_nu 2 integer real .false.

Call name is user_Nu
```

```
Generation code:
Generator function was not found for Nu


Testing ! user_xhi 2 real integer .FALSE.

Call name is user_XhI
Generation code:
Generator function was not found for XhI


Testing ! user_tau 2 integer real .TRUE.

Call name is user_taU
Generation code:
CALL user_tau_gen(Param1, Param2, Result)

Frequency code:
CALL user_tau_freq(Param1, Param2, Point, Result)


Testing !         user_phi       2       real    integer       .true.

Call name is user_PHI
Generation code:
CALL user_phi_gen(Param1, Param2, Result)

Frequency code:
CALL user_phi_freq(Param1, Param2, Point, Result)


Testing !user_sigma 2 integer real .true.

Call name is user_SigMa
Generation code:
CALL user_sigma_gen(Param1, Param2, Result)

Frequency code:
CALL user_sigma_freq(Param1, Param2, Point, Result)


Testing ! user_omega 2 REAL INTEGER .TRUE.

Call name is user_omega
Generation code:
CALL user_omega_gen(Param1, Param2, Result)

Frequency code:
CALL user_omega_freq(Param1, Param2, Point, Result)
```

4.2 package PIANOS.io

4.2.1 FortranWriter

The FortranWriter class is used instead of a usual FileWriter in writing the Fortran source files. Its tasks include cutting long lines into multiple continued lines and appending '&' in the beginning of each continued line. Also when lines are cut from the middle of a character string literal, '&' will be appended before the cut. FortranWriter also indents Fortran code correctly.

NOTE: The output of the Generator speaks for this class, as does the testing summary done with Emma [Emma]. FortranWriter is not a crucial part of the software, and is not tested any further.

4.2.2 Parser

This section outlines the testing strategy for the Parser and elaborates the steps of the testing. Throughout the tests the presence of a functional DistributionFactory is assumed.

Testing strategy

The testing will follow the bottom-up testing strategy. This means that the smallest parts that can be tested will be tested first, and when their functionality has been verified, the testing will move on to the methods using them. The last tested method for the Parser is therefore readModel, which calls all the other methods.

NOTE: Due to the lack of time available only parseVariable was tested thoroughly. The results of the program speak for the other parts of the Parser, as does the testing summary done with Emma [Emma]

parseVariable

Method syntax: parseVariable(File file, boolean isInteger, String toParse, HashMap<String, Variable> variableMapper, DistributionFactory fact)

Testing of this class can be done by running the ParserUnitTest class with correct parameters. Below is a list of tests that the class performs. Should there be an error, refer to the list below to see what goes wrong.

In the testing a functional *DistributionFactory* is given to the method in every case. The *variableMapper* given should be an empty one in every case that does not involve parameter dependencies. To test dependencies we should add some variable names and variables there. *isInteger* should be tested with both *true* and *false* for all test cases. The *file* parameter is used only for error printing, it can be any file, or even null. Every case should print the given file name in the error messages. What drives the tests here is what *toParse* contains. The test are performed with a Java class calling the separate methods with described parameters. For testing purposes the Parser has all methods publicly visible.

Here's a table of the tests to be performed:

case #	isInteger	toParse	Expected result
1	false	'(som?erubbish)'	'In null: Invalid variable name on line 0'
2	false	'alpha'	'In null: No column definition, expression or distribution found for variable on line 0'
3	false	'alpha()'	'In null: Invalid column number on line 0'
4	false	'alpha(10rub?bish)'	'In null: Invalid column number on line 0'
5	false	'alpha(15.23)'	'In null: Invalid column number on line 0'
6	false	'alpha(-1)'	'In null: Invalid column number on line 0 (columns start at 1)'
7	false	'alpha(2)'	A Variable with getName == alpha, isFunctional == false, column == 2, isInteger == false and isData == true.
8	false	'alpha(*)'	A Variable with getName == alpha, isFunctional == false, column == -1, isInteger == false and isData == true.
9	false	'alpha = 1.0'	A Variable with getName == alpha, isFunctional == true, isInteger == false, isData == false that has an equation with equation[0] == 1.0.
10	true	'alpha = 1.0'	This is wrong, but the equation validity is not checked. Should return a Variable such as above but with isInteger = true.
11	false	'alpha ~'	An error about a missing distribution.
12	false	'al3ph6a ~continuous_uniform('	'In null: Mismatching parentheses on line 0'
13	false	'alpha98 ~continuous_uniform( )'	'In null: Too few parameters on line 0'
14	false	'a4lpha ~continuous_uniform((v 1.8)'	'In null: Parameter 1 on line 0 is not a decimal number or a correct variable name'
15	false	'iota~continuous_uniform(1.0)'	'In null: Too few parameters on line 0'
16	false	'iota~discrete_uniform(0, 1.0)'	'In null: Parameter 2 on line 0 is not an integer or a correct variable name'
17	false	'iota~discrete_uniform(0, 1, 18)'	'In null: Too many parameters on line 0'
18	false	'iota~discrete_uniform(gamma, 1)'	'In null: Parameter 1 on line 0 is not defined before it is used'
19	false	'iota~discrete_uniform(gamma, 1)'	(First define gamma as a new Variable with isInteger == false and put it into variableMapper) 'In null: Parameter 1 on line 0 is not of type INTEGER'

20	false	‘iota~discrete_uniform(gamma, 1)’	(First define gamma as a new Variable with isInteger == true and put it into variableMapper) A variable with isInteger == false, getName == iota, getDependsList().get(0) == gamma that has the distribution discrete_uniform.
21	false	‘iota~continuous_uniform (gamma, 1.0)’	(with an empty variableMapper) ‘In null: Parameter 1 on line 0 is not defined before it is used’
22	false	‘iota~continuous_uniform (gamma, 1.0)’	(First define gamma as a new Variable with isInteger == true and put it into variableMapper) A variable with isInteger == false, getName == ‘iota’, getDependsList().get(0) == gamma that has the distribution continuous_uniform. Note: INTEGERS are allowed as parameters for distributions that accept REALs.
23	false	‘iota~continuous_uniform (gamma, 1.0)’	(First define gamma as a new Variable with isInteger == false and put it into variableMapper) A variable with getName() == ‘iota’, isInteger == false, getDependsList().get(0) == gamma, getDistribution.getParameter(0) == gamma.
24	false	‘iota = COUNT (&x) + 24*zeta’	‘In null: ‘COUNT(&varname)’ must be the only element of an equation if it is used (line 0)’
25	false	‘iota = SUM (&x) + 24*zeta’	‘In null: ‘SUM(&varname)’ must be the only element of an equation if it is used (line 0)’
26	false	iota = SUM (&gamma)	The Parser’s spatAffectedMap should now have the variable in it, and trying to find a Variable by the name from spatAffectedMap in variableMapper should return null (because we didn’t define gamma).
27	false	iota = SUM (&gamma)	(First define gamma and put it into variableMapper) The Parser’s spatAffectedMap should now have the variable in it, and trying to find a Variable by the name from spatAffectedMap in variableMapper should return the variable gamma. iota.getEquation().getEquation()[0] should be ‘SUM(&’ and iota.getEquation().getEquation()[1] == ‘gamma’.

28	false	'iota = 24*zeta'	'In null: Variable 1 on line 0 is not defined before it is used'
29	false	'iota = 28 * gamma * gamma *BLAH(24)'	(First define gamma as a new Variable with isInteger == true and put it into variableMapper) should return a Variable with getName() == 'iota', with the getEquation().getEquation() array: '28*', 'gamma', '*', 'gamma', '*BLAH(24)', with iota.getDependsList().get(0) == gamma and and gamma.getAffectsList().get(0) == iota.

The method parseVariable spans lines 359 through 678 in the Parser. The method is divided up like this:

- 359-465 for general variable processing and data column reading
- 466-595 for stochastic variables
- 596-675 for functional variables
- 676-678 are for returning the Variable constructed.

After tests 1-22 succeed, all branches to do with stochastic variables and general processing (361-593) have been tested. Tests 23-29 test the functional variable branches (594-672) thoroughly. After all tests (1-29) complete successfully, parseVariable has been tested with 100% branch coverage.

4.3 package PIANOS.generator

When testing the following classes it is assumed that FortranWriter works correctly.

4.3.1 Definitions

Methods to be tested:

- generateDefinitions

Strategy:

For each semantically valid model used in testing:

- Call generateDefinitions
- Read the source code produced and check that it looks correct

4.3.2 Input

Methods to be tested:

- generateInput

Strategy:
For each semantically valid model used in testing:

- Call generateInput
- Read the source code produced and check that it looks correct

4.3.3 Output

Methods to be tested:

- generateOutput

Strategy:
For each semantically valid model used in testing:

- Call generateOutput
- Read the source code produced and check that it looks correct

4.3.4 Proposal

Methods to be tested:

- generateProposal

Strategy:
For each semantically valid model used in testing:

- Call generateProposal
- Read the source code produced and check that it looks correct

4.3.5 Acceptation

Methods to be tested:

- setTopologicalList (trivial)
- generateNewValueCode
- generateAcceptationCode (calls a private method acceptationAffectedOf, which calls a private method generateUpdateOneFunctional)
- generateNewValuesFunctionalCode (calls private methods topologicalSort and generateNewValueForOneFunctional)
- generateAcceptationFormula (calls private methods generateAcceptationFormulaGlobal, generateAcceptationFormulaOneDimensional and generateAcceptationFormulaTwoDimensional, all of which call private methods generateLikelihoodFormulaGlobal, generateLikelihoodFormulaOneDimensional, generateLikelihoodFormulaTwoDimensional and generateTransitionFormula)

Strategy:

Since the methods assume that the model is correct (that is, all the dependencies are valid and all needed information is given), there is no point testing the Acceptation class with invalid models. (The Parser should identify the incorrect models and stop the execution. In this testing phase the Parser is assumed to work correctly.)

The testing should be done by calling the methods with a valid model constructed by the Parser and then inspecting the code the methods produce. It would be difficult produce a program to check the results automatically.

For each valid model used in testing:

- Call setTopologicalList with a topological variable list of the current model, because it must be called before calling the other methods.
- Call generateNewValueCode once with each non-functional not-from-data parameter
- Call generateNewValuesFunctional once with each non-functional not-from-data parameter
- Call generateAcceptationFormula once with each non-functional not-from-data parameter
- Call generateAcceptationCode once with each non-functional not-from-data parameter

There is a test program AcceptationTest which runs these tests.

Expected results: See Appendix 2.

4.3.6 FortranMain

Methods to be tested:

- generateMain (calls private methods generateUpdateOne, generateUpdateAll, generateSetFunctional)

Strategy:

It's useful to assume that the Acceptation class works correctly when testing the FortranMain class, since the methods call Acceptation methods.

For each semantically valid model used in testing:

- Call generateMain
- Read the source code produced and check that it looks correct

4.3.7 Generator

No unit tests are performed.

5 References

coverage PIANOS code coverage report
http://www.cs.helsinki.fi/group/linja/PIANOScdrom/src/java/coverage/

Emma EMMA: a free Java code coverage tool
http://emma.sourceforge.net/

Appendix 1. Coverage Report

This is a short summary of the code coverage for the tests performed. For details see the testing code coverage report on the PIANOS CD-ROM or [coverage].

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	100% (28/28)	92% (213/231)	88% (18171/20737)	91% (3493/3818)

OVERALL STATS SUMMARY

- total packages: 5
- total executable files: 28
- total classes: 28
- total methods: 231
- total executable lines: 3818

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
PIANOS.io	100% (2/2)	96% (26/27)	80% (4280/5328)	91% (974,2/1070)
PIANOS	100% (1/1)	71% (5/7)	89% (685/773)	88% (148,2/169)
PIANOS.generator	100% (6/6)	90% (54/60)	90% (10211/11322)	93% (1810,2/1955)
PIANOS.datastructures	100% (13/13)	93% (122/131)	90% (2971/3290)	90% (548,3/612)
PIANOS.exceptions	100% (6/6)	100% (6/6)	100% (24/24)	100% (12/12)

Appendix 2. Expected results for the tests of Acceptation

The simple bird model

New value code for alpha:

```
IF (alpha % buffer_index > SIZE(alpha % buffer) .OR. alpha % buffer_index < &
    1) THEN
    CALL generate('alpha', alpha % buffer)
    alpha % buffer_index = 1
END IF

alpha % one_dim(1) % new_value = alpha % buffer(alpha % buffer_index)
alpha % buffer_index = alpha % buffer_index + 1
```

New values for functional parameters depending on alpha:

```
! calculating new value(s) for p

DO i2 = 1, 21
    DO j2 = 1, 4
        p % two_dim(i2, j2) % new_value = EXP(alpha % one_dim(1) % &
            new_value*northerness % one_dim(i2) % value)/(1+EXP(alpha % &
            one_dim(1) % new_value*northerness % one_dim(i2) % value))
    END DO
END DO
```

Acceptation formula for alpha:

```
!  P(alpha') / P(alpha)

p_acc = 1
IF (alpha % one_dim(1) % new_value < -1.0_dp .OR. alpha % one_dim(1) % &
    new_value > 1.0_dp) THEN
    new_frequency = 0
ELSE
    new_frequency = 1.0 / (1.0_dp - (-1.0_dp))
END IF
IF (alpha % one_dim(1) % value < -1.0_dp .OR. alpha % one_dim(1) % value > &
    1.0_dp) THEN
    frequency = 0
ELSE
    frequency = 1.0 / (1.0_dp - (-1.0_dp))
END IF
p_acc = p_acc * new_frequency / frequency

! P( x | alpha') / P( x | alpha)

DO i=1, 21
    DO j=1, 4
        CALL user_bernoulli_freq(p % two_dim(i, j) % new_value, x % &
```

```
            two_dim(i, j) % value, new_frequency)
        CALL user_bernoulli_freq(p % two_dim(i, j) % value, x % two_dim(i, j) % &
            value, frequency)
        p_acc = p_acc * new_frequency / frequency
    END DO
END DO

! q( alpha', alpha) / q(alpha, alpha')

IF (alpha % one_dim(1) % value < -1.0_dp .OR. alpha % one_dim(1) % value > &
    1.0_dp) THEN
    frequency = 0
ELSE
    frequency = 1.0 / (1.0_dp - (-1.0_dp))
END IF
IF (alpha % one_dim(1) % new_value < -1.0_dp .OR. alpha % one_dim(1) % &
    new_value > 1.0_dp) THEN
    new_frequency = 0
ELSE
    new_frequency = 1.0 / (1.0_dp - (-1.0_dp))
END IF
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for alpha (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    alpha % one_dim(1) % value = alpha % one_dim(1) % new_value
    alpha % one_dim(1) % successful_changes = alpha % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 21
        DO j2 = 1, 4
            p % two_dim(i2, j2) % value = p % two_dim(i2, j2) % new_value
        END DO
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for alpha (if the update strategy is random):

```
IF (p_acc > prob(prob_index)) THEN
    alpha % one_dim(1) % value = alpha % one_dim(1) % new_value
    alpha % one_dim(1) % successful_changes = alpha % one_dim(1) % &
        successful_changes + 1

    IF (alpha % one_dim(1) % successful_changes == alpha % one_dim(1) % &
        update_count) THEN
        parameters_achieved = parameters_achieved + 1
```

```
    END IF
    DO i2 = 1, 21
        DO j2 = 1, 4
            p % two_dim(i2, j2) % value = p % two_dim(i2, j2) % new_value
        END DO
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

New value code for x:

```
IF (x % buffer_index > SIZE(x % buffer) .OR. x % buffer_index < 1) THEN
    CALL generate('x', x % buffer)
    x % buffer_index = 1
END IF

x % two_dim(i, j) % new_value = x % buffer(x % buffer_index)
x % buffer_index = x % buffer_index + 1
```

New values for functional parameters depending on x:

Nothing should be printed.

Acceptation formula for x:

```
!  P(x') / P(x)

p_acc = 1
CALL user_bernoulli_freq(p % two_dim(i, j) % value, x % two_dim(i, j) % &
    new_value, new_frequency)
CALL user_bernoulli_freq(p % two_dim(i, j) % value, x % two_dim(i, j) % &
    value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( obs | x') / P( obs | x)

CALL user_defined_points_freq(x % two_dim(i, j) % new_value, obs % two_dim(i, &
    j) % value, new_frequency)
CALL user_defined_points_freq(x % two_dim(i, j) % value, obs % two_dim(i, j) &
    % value, frequency)
p_acc = p_acc * new_frequency / frequency

! q( x', x) / q(x, x')

IF (x % two_dim(i, j) % value < 0 .OR. x % two_dim(i, j) % value > 1) THEN
    frequency = 0
ELSE
    frequency = 1.0 / (REAL(1) - (REAL(0)) + 1.0)
END IF
```

```
IF (x % two_dim(i, j) % new_value < 0 .OR. x % two_dim(i, j) % new_value > 1) &
    THEN
    new_frequency = 0
ELSE
    new_frequency = 1.0 / (REAL(1) - (REAL(0)) + 1.0)
END IF
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for x (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    x % two_dim(i, j) % value = x % two_dim(i, j) % new_value
    x % two_dim(i, j) % successful_changes = x % two_dim(i, j) % &
        successful_changes + 1

END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for x (if the update strategy is random):

```
IF (p_acc > prob(prob_index)) THEN
    x % two_dim(i, j) % value = x % two_dim(i, j) % new_value
    x % two_dim(i, j) % successful_changes = x % two_dim(i, j) % &
        successful_changes + 1

    IF (x % two_dim(i, j) % successful_changes == x % two_dim(i, j) % &
        update_count) THEN
        parameters_achieved = parameters_achieved + 1
    END IF
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

The simple bird model with spatial dependence

Only sections that differ from the previous test case are described here.

New values for functional parameters depending on alpha:

```
! calculating new value(s) for p

DO i2 = 1, 21
    DO j2 = 1, 4
        p % two_dim(i2, j2) % new_value = EXP(alpha % one_dim(1) % &
            new_value*northerness % one_dim(i2) % value+q % two_dim(i2, j2) % &
```

```
            value)/(1+EXP(alpha % one_dim(1) % new_value*northerness % &
            one_dim(i2) % value+q % two_dim(i2, j2) % value))
    END DO
END DO
```

Acceptation code for x (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    x % two_dim(i, j) % value = x % two_dim(i, j) % new_value
    x % two_dim(i, j) % successful_changes = x % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating q % two_dim(spatial(i, k), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + x % two_dim(spatial(neighbour, h + 1), j) % &
                value
        END DO
        q % two_dim(neighbour, j) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i,  k +1)
        p % two_dim(neighbour, j) % value = EXP(alpha % one_dim(1) % &
            value*northerness % one_dim(neighbour) % value+q % &
            two_dim(neighbour, j) % value)/(1+EXP(alpha % one_dim(1) % &
            value*northerness % one_dim(neighbour) % value+q % &
            two_dim(neighbour, j) % value))
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for x (if the update strategy is random):

```
IF (p_acc > prob(prob_index)) THEN
    x % two_dim(i, j) % value = x % two_dim(i, j) % new_value

    x % two_dim(i, j) % successful_changes = x % two_dim(i, j) % &
        successful_changes + 1

    IF (x % two_dim(i, j) % successful_changes == x % two_dim(i, j) % &
        update_count) THEN
        parameters_achieved = parameters_achieved + 1
    END IF

    ! 1 to number of neighbours
```

```
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating q % two_dim(spatial(i, k), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + x % two_dim(spatial(neighbour, h + 1), j) % &
                value
        END DO
        q % two_dim(neighbour, j) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i,  k +1)
        p % two_dim(neighbour, j) % value = EXP(alpha % one_dim(1) % &
            value*northerness % one_dim(neighbour) % value+q % &
            two_dim(neighbour, j) % value)/(1+EXP(alpha % one_dim(1) % &
            value*northerness % one_dim(neighbour) % value+q % &
            two_dim(neighbour, j) % value))
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

The lip cancer model

The number of counties is set to 10.

New values for functional parameters depending on alpha:

```
! calculating new value(s) for mu

DO i2 = 1, 10
    mu % one_dim(i2) % new_value = EXP(LOG(expected % one_dim(i2) % &
        value)+alpha % one_dim(1) % new_value*x % one_dim(i2) % &
        value/10+sigma % one_dim(1) % value*b % one_dim(i2) % value)
END DO
```

Acceptation formula for alpha:

```
!  P(alpha') / P(alpha)

p_acc = 1
CALL user_normal_freq(0.0_dp, 100.0_dp, alpha % one_dim(1) % new_value, &
    new_frequency)
CALL user_normal_freq(0.0_dp, 100.0_dp, alpha % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( obs | alpha') / P( obs | alpha)

DO i=1, 10
    ifail=0
```

```
    IF (mu % one_dim(i) % new_value > 1.0D+6) THEN
        CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
            new_frequency, ifail)
    ELSE
        CALL G01BKF(mu % one_dim(i) % new_value, obs % one_dim(i) % value, &
            temp_real, temp_real, new_frequency, ifail)
    END IF
    IF (ifail /= 0) THEN
        WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
            &distribution, parameters: mu = ', mu % one_dim(i) % new_value
        STOP
    END IF
    ifail=0
    IF (mu % one_dim(i) % value > 1.0D+6) THEN
        CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
            frequency, ifail)
    ELSE
        CALL G01BKF(mu % one_dim(i) % value, obs % one_dim(i) % value, &
            temp_real, temp_real, frequency, ifail)
    END IF
    IF (ifail /= 0) THEN
        WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
            &distribution, parameters: mu = ', mu % one_dim(i) % value
        STOP
    END IF
    p_acc = p_acc * new_frequency / frequency
END DO

! q(alpha', alpha) / q(alpha, alpha')

transition_real = alpha % one_dim(1) % new_value - alpha % one_dim(1) % value
CALL user_normal_freq(0.0_dp, 10.0_dp, -transition_real, frequency)
CALL user_normal_freq(0.0_dp, 10.0_dp, transition_real, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for alpha:

```
IF (p_acc > prob(prob_index)) THEN
    alpha % one_dim(1) % value = alpha % one_dim(1) % new_value
    alpha % one_dim(1) % successful_changes = alpha % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 10
        mu % one_dim(i2) % value = mu % one_dim(i2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on sigma:

```
! calculating new value(s) for mu

DO i2 = 1, 10
    mu % one_dim(i2) % new_value = EXP(LOG(expected % one_dim(i2) % &
        value)+alpha % one_dim(1) % value*x % one_dim(i2) % value/10+sigma % &
        one_dim(1) % new_value*b % one_dim(i2) % value)
END DO
```

Acceptation formula for sigma:

```
!  P(sigma') / P(sigma)

p_acc = 1
CALL user_gamma_freq(1.0_dp, 1.0_dp, sigma % one_dim(1) % new_value, &
    new_frequency)
CALL user_gamma_freq(1.0_dp, 1.0_dp, sigma % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( obs | sigma') / P( obs | sigma)

DO i=1, 10
    ifail=0
    IF (mu % one_dim(i) % new_value > 1.0D+6) THEN
        CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
            new_frequency, ifail)
    ELSE
        CALL G01BKF(mu % one_dim(i) % new_value, obs % one_dim(i) % value, &
            temp_real, temp_real, new_frequency, ifail)
    END IF
    IF (ifail /= 0) THEN
        WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
            &distribution, parameters: mu = ', mu % one_dim(i) % new_value
        STOP
    END IF
    ifail=0
    IF (mu % one_dim(i) % value > 1.0D+6) THEN
        CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
            frequency, ifail)
    ELSE
        CALL G01BKF(mu % one_dim(i) % value, obs % one_dim(i) % value, &
            temp_real, temp_real, frequency, ifail)
    END IF
    IF (ifail /= 0) THEN
        WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
            &distribution, parameters: mu = ', mu % one_dim(i) % value
        STOP
    END IF
    p_acc = p_acc * new_frequency / frequency
END DO

! q(sigma', sigma) / q(sigma, sigma')

IF (sigma % one_dim(1) % value < 0.0010_dp .OR. sigma % one_dim(1) % value > &
```

```
    5.0_dp) THEN
    frequency = 0
ELSE
    frequency = 1.0 / (5.0_dp - (0.0010_dp))
END IF
IF (sigma % one_dim(1) % new_value < 0.0010_dp .OR. sigma % one_dim(1) % &
    new_value > 5.0_dp) THEN
    new_frequency = 0
ELSE
    new_frequency = 1.0 / (5.0_dp - (0.0010_dp))
END IF
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for sigma (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    sigma % one_dim(1) % value = sigma % one_dim(1) % new_value
    sigma % one_dim(1) % successful_changes = sigma % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 10
        mu % one_dim(i2) % value = mu % one_dim(i2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on b:

Note that the q:s and t:s of neighbour squares don't need to be updated at this phase. They are updated only if the new value for b is accepted.

```
! calculating new value(s) for mu

i2 = i
mu % one_dim(i2) % new_value = EXP(LOG(expected % one_dim(i2) % value)+alpha &
    % one_dim(1) % value*x % one_dim(i2) % value/10+sigma % one_dim(1) % &
    value*b % one_dim(i2) % new_value)
```

Acceptation formula for b:

```
!  P(b') / P(b)

p_acc = 1
CALL user_normal_freq(t % one_dim(i) % value, n % one_dim(i) % value, b % &
    one_dim(i) % new_value, new_frequency)
CALL user_normal_freq(t % one_dim(i) % value, n % one_dim(i) % value, b % &
    one_dim(i) % value, frequency)
```

```
p_acc = p_acc * new_frequency / frequency


! P( obs | b') / P( obs | b)

ifail=0
IF (mu % one_dim(i) % new_value > 1.0D+6) THEN
    CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
        new_frequency, ifail)
ELSE
    CALL G01BKF(mu % one_dim(i) % new_value, obs % one_dim(i) % value, &
        temp_real, temp_real, new_frequency, ifail)
END IF
IF (ifail /= 0) THEN
    WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
        &distribution, parameters: mu = ', mu % one_dim(i) % new_value
    STOP
END IF
ifail=0
IF (mu % one_dim(i) % value > 1.0D+6) THEN
    CALL G01BKF( 1.0D+6, obs % one_dim(i) % value, temp_real, temp_real, &
        frequency, ifail)
ELSE
    CALL G01BKF(mu % one_dim(i) % value, obs % one_dim(i) % value, temp_real, &
        temp_real, frequency, ifail)
END IF
IF (ifail /= 0) THEN
    WRITE (*, *) 'Error when trying to calculate frequency of the poisson &
        &distribution, parameters: mu = ', mu % one_dim(i) % value
    STOP
END IF
p_acc = p_acc * new_frequency / frequency

! q(b', b) / q(b, b')

IF (b % one_dim(i) % value < -10.0_dp .OR. b % one_dim(i) % value > 10.0_dp) &
    THEN
    frequency = 0
ELSE
    frequency = 1.0 / (10.0_dp - (-10.0_dp))
END IF
IF (b % one_dim(i) % new_value < -10.0_dp .OR. b % one_dim(i) % new_value > &
    10.0_dp) THEN
    new_frequency = 0
ELSE
    new_frequency = 1.0 / (10.0_dp - (-10.0_dp))
END IF
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for b (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    b % one_dim(i) % value = b % one_dim(i) % new_value
```

```
    b % one_dim(i) % successful_changes = b % one_dim(i) % successful_changes &
        + 1

    i2 = i
    mu % one_dim(i2) % value = mu % one_dim(i2) % new_value
    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_real = 0
        neighbour = spatial(i, k + 1)
        ! updating q % one_dim(spatial(i, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_real = sum_real + b % one_dim(spatial(neighbour, h + 1)) % &
                value
        END DO
        q % one_dim(neighbour) % value = sum_real
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i,  k +1)
        t % one_dim(neighbour) % value = q % one_dim(neighbour) % value/n % &
            one_dim(neighbour) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

A model with all kinds of stochastic dependencies

Apart from the model description the sizes of the entities are set by the testing program, since there is no data from which the sizes could be calculated.

Only acceptation codes are so non-trivial that they are described here. Nothing should be prited when calculating new values for functional parameters since there are none. Acceptation codes include updating only the current parameter.

Acceptation formula for a1:

```
!  P(a1') / P(a1)

p_acc = 1
CALL user_dist_1_freq(3, a1 % one_dim(1) % new_value, new_frequency)
CALL user_dist_1_freq(3, a1 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! q( a1', a1) / q(a1, a1')

CALL user_dist_prop_1_freq(1, 2, a1 % one_dim(1) % value, frequency)
CALL user_dist_prop_1_freq(1, 2, a1 % one_dim(1) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a2:

```
!  P(a2') / P(a2)

p_acc = 1
CALL user_dist_2_freq(4, a2 % one_dim(1) % new_value, new_frequency)
CALL user_dist_2_freq(4, a2 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( b2 | a2') / P( b2 | a2)

CALL user_dist_2-_freq(a2% one_dim(1) % new_value, b2 % one_dim(1), &
    new_frequency)
CALL user_dist_2-_freq(a2% one_dim(1) % value, b2 % one_dim(1), frequency)
p_acc = p_acc * new_frequency / frequency

! q( a2', a2) / q(a2, a2')

CALL user_dist_prop_2_freq(1, 2, a2 % one_dim(1) % value, frequency)
CALL user_dist_prop_2_freq(1, 2, a2 % one_dim(1) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b2:

```
!  P(b2') / P(b2)

p_acc = 1
CALL user_dist_2-_freq(a2 % one_dim(1) % value, b2 % one_dim(1) % new_value, &
    new_frequency)
CALL user_dist_2-_freq(a2 % one_dim(1) % value, b2 % one_dim(1) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q( b2', b2) / q(b2, b2')

CALL user_dist_prop_2_freq(1, 2, b2 % one_dim(1) % value, frequency)
CALL user_dist_prop_2_freq(1, 2, b2 % one_dim(1) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a3:

```
!  P(a3') / P(a3)

p_acc = 1
CALL user_dist_3_freq(5, a3 % one_dim(i) % new_value, new_frequency)
CALL user_dist_3_freq(5, a3 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! q( a3', a3) / q(a3, a3')

CALL user_dist_prop_3_freq(1, 2, a3 % one_dim(i) % value, frequency)
CALL user_dist_prop_3_freq(1, 2, a3 % one_dim(i) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a4:

```
!  P(a4') / P(a4)

p_acc = 1
CALL user_dist_4_freq(6, a4 % one_dim(1) % new_value, new_frequency)
CALL user_dist_4_freq(6, a4 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( b4 | a4') / P( b4 | a4)

DO i=1, 2
    CALL user_dist_4-_freq(a4 % one_dim(1) % new_value, b4 % one_dim(i) % &
        value, new_frequency)
    CALL user_dist_4-_freq(a4 % one_dim(1) % value, b4 % one_dim(i) % value, &
        frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q( a4', a4) / q(a4, a4')

CALL user_dist_prop_4_freq(1, 2, a4 % one_dim(1) % value, frequency)
CALL user_dist_prop_4_freq(1, 2, a4 % one_dim(1) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Where 2 is the size of e4.

Acceptation formula for b4:

```
!  P(b4') / P(b4)

p_acc = 1
CALL user_dist_4-_freq(a4 % one_dim(1) % value, b4 % one_dim(i) % new_value, &
    new_frequency)
CALL user_dist_4-_freq(a4 % one_dim(1) % value, b4 % one_dim(i) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency


! q( b4', b4) / q(b4, b4')

CALL user_dist_prop_4_freq(1, 2, b4 % one_dim(i) % value, frequency)
CALL user_dist_prop_4_freq(1, 2, b4 % one_dim(i) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a5 (note that the proposal strategy is random walk):

```
!  P(a5') / P(a5)

p_acc = 1
CALL user_dist_5_freq(7, a5 % one_dim(i) % new_value, new_frequency)
CALL user_dist_5_freq(7, a5 % one_dim(i) % value, frequency)
```

```
p_acc = p_acc * new_frequency / frequency


! P( b5 | a5') / P( b5 | a5)

CALL user_dist_5-_freq(a5 % one_dim(i) % new_value, b5 % one_dim(i) % value, &
    new_frequency)
CALL user_dist_5-_freq(a5 % one_dim(i) % value, b5 % one_dim(i) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q( a5', a5) / q(a5, a5')

transition_int = a5 % one_dim(i) % new_value - a5 % one_dim(i) % value
CALL user_dist_prop_5_freq(1, 2, -transition_int, frequency)
CALL user_dist_prop_5_freq(1, 2, transition_int, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b5:

```
!  P(b5') / P(b5)

p_acc = 1
CALL user_dist_5-_freq(a5 % one_dim(i) % value, b5 % one_dim(i) % new_value, &
    new_frequency)
CALL user_dist_5-_freq(a5 % one_dim(i) % value, b5 % one_dim(i) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency


! q( b5', b5) / q(b5, b5')

CALL user_dist_prop_5_freq(1, 2, b5 % one_dim(i) % value, frequency)
CALL user_dist_prop_5_freq(1, 2, b5 % one_dim(i) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a6:

```
!  P(a6') / P(a6)

p_acc = 1
CALL user_dist_6_freq(5, a6 % two_dim(i, j) % new_value, new_frequency)
CALL user_dist_6_freq(5, a6 % two_dim(i, j) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! q( a6', a6) / q(a6, a6')

CALL user_dist_prop_6_freq(1, 2, a6 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_6_freq(1, 2, a6 % two_dim(i, j) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a7:

```
!  P(a7') / P(a7)

p_acc = 1
CALL user_dist_7_freq(8, a7 % one_dim(1) % new_value, new_frequency)
CALL user_dist_7_freq(8, a7 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( b7 | a7') / P( b7 | a7)

DO i=1, 6
    DO j=1, 7
        CALL user_dist_7-_freq(a7 % one_dim(1) % new_value, b7 % two_dim(i, &
            j) % value, new_frequency)
        CALL user_dist_7-_freq(a7 % one_dim(1) % value, b7 % two_dim(i, j) % &
            value, frequency)
        p_acc = p_acc * new_frequency / frequency
    END DO
END DO

! q( a7', a7) / q(a7, a7')

CALL user_dist_prop_7_freq(1, 2, a7 % one_dim(1) % value, frequency)
CALL user_dist_prop_7_freq(1, 2, a7 % one_dim(1) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b7:

```
!  P(b7') / P(b7)

p_acc = 1
CALL user_dist_7-_freq(a7 % one_dim(1) % value, b7 % two_dim(i, j) % &
    new_value, new_frequency)
CALL user_dist_7-_freq(a7 % one_dim(1) % value, b7 % two_dim(i, j) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q( b7', b7) / q(b7, b7')

CALL user_dist_prop_7_freq(1, 2, b7 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_7_freq(1, 2, b7 % two_dim(i, j) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a8:

```
!  P(a8') / P(a8)

p_acc = 1
CALL user_dist_8_freq(9, a8 % one_dim(i) % new_value, new_frequency)
CALL user_dist_8_freq(9, a8 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency
```

```
! P( b8 | a8') / P( b8 | a8)

DO j=1, 9
    CALL user_dist_8-_freq(a8 % one_dim(i) % new_value, b8 % two_dim(i, j) % &
        value, new_frequency)
    CALL user_dist_8-_freq(a8 % one_dim(i) % value, b8 % two_dim(i, j) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q( a8', a8) / q(a8, a8')

CALL user_dist_prop_8_freq(1, 2, a8 % one_dim(i) % value, frequency)
CALL user_dist_prop_8_freq(1, 2, a8 % one_dim(i) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b8:

```
!  P(b8') / P(b8)

p_acc = 1
CALL user_dist_8-_freq(a8 % one_dim(i) % value, b8 % two_dim(i, j) % &
    new_value, new_frequency)
CALL user_dist_8-_freq(a8 % one_dim(i) % value, b8 % two_dim(i, j) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q( b8', b8) / q(b8, b8')

CALL user_dist_prop_8_freq(1, 2, b8 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_8_freq(1, 2, b8 % two_dim(i, j) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a9:

```
!  P(a9') / P(a9)

p_acc = 1
CALL user_dist_9_freq(10, a9 % one_dim(i) % new_value, new_frequency)
CALL user_dist_9_freq(10, a9 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( b9 | a9') / P( b9 | a9)

DO j=1, 10
    CALL user_dist_9-_freq(a9 % one_dim(i) % new_value, b9 % two_dim(j, i) % &
        value, new_frequency)
    CALL user_dist_9-_freq(a9 % one_dim(i) % value, b9 % two_dim(j, i) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO
```

```
! q( a9', a9) / q(a9, a9')

CALL user_dist_prop_9_freq(1, 2, a9 % one_dim(i) % value, frequency)
CALL user_dist_prop_9_freq(1, 2, a9 % one_dim(i) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b9:

```
!  P(b9') / P(b9)

p_acc = 1
CALL user_dist_9-_freq(a9 % one_dim(j) % value, b9 % two_dim(i, j) % &
    new_value, new_frequency)
CALL user_dist_9-_freq(a9 % one_dim(j) % value, b9 % two_dim(i, j) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q( b9', b9) / q(b9, b9')

CALL user_dist_prop_9_freq(1, 2, b9 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_9_freq(1, 2, b9 % two_dim(i, j) % new_value, new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for a10:

```
!  P(a10') / P(a10)

p_acc = 1
CALL user_dist_10_freq(10, a10 % two_dim(i, j) % new_value, new_frequency)
CALL user_dist_10_freq(10, a10 % two_dim(i, j) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( b10 | a10') / P( b10 | a10)

CALL user_dist_10-_freq(a10 % two_dim(i, j) % new_value, b10 % two_dim(i, j) &
    % value, new_frequency)
CALL user_dist_10-_freq(a10 % two_dim(i, j) % value, b10 % two_dim(i, j) % &
    value, frequency)
p_acc = p_acc * new_frequency / frequency

! q( a10', a10) / q(a10, a10')

CALL user_dist_prop_10_freq(1, 2, a10 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_10_freq(1, 2, a10 % two_dim(i, j) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation formula for b10:

```
!  P(b10') / P(b10)
```

```
p_acc = 1
CALL user_dist_10-_freq(a10 % two_dim(i, j) % value, b10 % two_dim(i, j) % &
    new_value, new_frequency)
CALL user_dist_10-_freq(a10 % two_dim(i, j) % value, b10 % two_dim(i, j) % &
    value, frequency)
p_acc = p_acc * new_frequency / frequency

! q( b10', b10) / q(b10, b10')

CALL user_dist_prop_10_freq(1, 2, b10 % two_dim(i, j) % value, frequency)
CALL user_dist_prop_10_freq(1, 2, b10 % two_dim(i, j) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

A model with all kinds of functional dependencies

New value codes are too trivial to be represented here. Only calculating new values for functional parameters, acceptation formulae and acceptation codes for parameters a1 - a26 are included.

Calculating new values for functional parameters depending on a1:

```
! calculating new value(s) for b1

b1 % one_dim(1) % new_value = a1 % one_dim(1) % new_value+1
```

Acceptation formula for a1:

```
!  P(a1') / P(a1)

p_acc = 1
CALL user_dist_1_freq(1.0_dp, a1 % one_dim(1) % new_value, new_frequency)
CALL user_dist_1_freq(1.0_dp, a1 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! q(a1', a1) / q(a1, a1')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a1 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a1 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a1 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a1 % one_dim(1) % value = a1 % one_dim(1) % new_value
    a1 % one_dim(1) % successful_changes = a1 % one_dim(1) % &
        successful_changes + 1

    b1 % one_dim(1) % value = b1 % one_dim(1) % new_value
```

```
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a2:

```
! calculating new value(s) for b2

b2 % one_dim(1) % new_value = a2 % one_dim(1) % new_value-2
```

Acceptation formula for a2:

```
!  P(a2') / P(a2)

p_acc = 1
CALL user_dist_2_freq(2.0_dp, a2 % one_dim(1) % new_value, new_frequency)
CALL user_dist_2_freq(2.0_dp, a2 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c2 | a2') / P( c2 | a2)

CALL user_dist_2-_freq(b2% one_dim(1) % new_value, c2 % one_dim(1), &
    new_frequency)
CALL user_dist_2-_freq(b2% one_dim(1) % value, c2 % one_dim(1), frequency)
p_acc = p_acc * new_frequency / frequency

! q(a2', a2) / q(a2, a2')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a2 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a2 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a2 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a2 % one_dim(1) % value = a2 % one_dim(1) % new_value
    a2 % one_dim(1) % successful_changes = a2 % one_dim(1) % &
        successful_changes + 1

    b2 % one_dim(1) % value = b2 % one_dim(1) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a4:

```
! calculating new value(s) for b4

i2 = i
b4 % one_dim(i2) % new_value = a4 % one_dim(i2) % new_value/4
```

Acceptation formula for a4:

```
!  P(a4') / P(a4)

p_acc = 1
CALL user_dist_4_freq(4.0_dp, a4 % one_dim(i) % new_value, new_frequency)
CALL user_dist_4_freq(4.0_dp, a4 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! q(a4', a4) / q(a4, a4')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a4 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a4 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a4 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a4 % one_dim(i) % value = a4 % one_dim(i) % new_value
    a4 % one_dim(i) % successful_changes = a4 % one_dim(i) % &
        successful_changes + 1

    i2 = i
    b4 % one_dim(i2) % value = b4 % one_dim(i2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a5:

```
! calculating new value(s) for b5

i2 = i
b5 % one_dim(i2) % new_value = EXP(a5 % one_dim(i2) % new_value)
```

Acceptation formula for a5:

```
!  P(a5') / P(a5)

p_acc = 1
CALL user_dist_2_freq(5.0_dp, a5 % one_dim(i) % new_value, new_frequency)
CALL user_dist_2_freq(5.0_dp, a5 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( c5 | a5') / P( c5 | a5)

CALL user_dist_5-_freq(b5 % one_dim(i) % new_value, c5 % one_dim(i) % value, &
    new_frequency)
CALL user_dist_5-_freq(b5 % one_dim(i) % value, c5 % one_dim(i) % value, &
    frequency)
p_acc = p_acc * new_frequency / frequency

! q(a5', a5) / q(a5, a5')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a5 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a5 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a5 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a5 % one_dim(i) % value = a5 % one_dim(i) % new_value
    a5 % one_dim(i) % successful_changes = a5 % one_dim(i) % &
        successful_changes + 1

    i2 = i
    b5 % one_dim(i2) % value = b5 % one_dim(i2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a7:

```
! calculating new value(s) for b7

i2 = i
j2 = j
b7 % two_dim(i2, j2) % new_value = a7 % two_dim(i2, j2) % new_value**2
```

Acceptation formula for a7:

```
!  P(a7') / P(a7)
```

```
p_acc = 1
CALL user_dist_7_freq(7.0_dp, a7 % two_dim(i, j) % new_value, new_frequency)
CALL user_dist_7_freq(7.0_dp, a7 % two_dim(i, j) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! q(a7', a7) / q(a7, a7')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a7 % two_dim(i, j) % value, &
    frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a7 % two_dim(i, j) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a7 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a7 % two_dim(i, j) % value = a7 % two_dim(i, j) % new_value
    a7 % two_dim(i, j) % successful_changes = a7 % two_dim(i, j) % &
        successful_changes + 1

    i2 = i
    j2 = j
    b7 % two_dim(i2, j2) % value = b7 % two_dim(i2, j2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a8:

```
! calculating new value(s) for b8

i2 = i
j2 = j
b8 % two_dim(i2, j2) % new_value = EXP(a8 % two_dim(i2, j2) % new_value)
```

Acceptation formula for a8:

```
!  P(a8') / P(a8)

p_acc = 1
CALL user_dist_2_freq(8.0_dp, a8 % two_dim(i, j) % new_value, new_frequency)
CALL user_dist_2_freq(8.0_dp, a8 % two_dim(i, j) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c8 | a8') / P( c8 | a8)
```

```
CALL user_dist_8-_freq(b8 % two_dim(i, j) % new_value, c8 % two_dim(i, j) % &
    value, new_frequency)
CALL user_dist_8-_freq(b8 % two_dim(i, j) % value, c8 % two_dim(i, j) % &
    value, frequency)
p_acc = p_acc * new_frequency / frequency

! q(a8', a8) / q(a8, a8')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a8 % two_dim(i, j) % value, &
    frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a8 % two_dim(i, j) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a8 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a8 % two_dim(i, j) % value = a8 % two_dim(i, j) % new_value
    a8 % two_dim(i, j) % successful_changes = a8 % two_dim(i, j) % &
        successful_changes + 1

    i2 = i
    j2 = j
    b8 % two_dim(i2, j2) % value = b8 % two_dim(i2, j2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a10:

```
! calculating new value(s) for b10

DO i2 = 1, 16
    b10 % one_dim(i2) % new_value = SIN(a10 % one_dim(1) % new_value)
END DO
```

Acceptation formula for a10:

```
!  P(a10') / P(a10)

p_acc = 1
CALL user_dist_10_freq(10.0_dp, a10 % one_dim(1) % new_value, new_frequency)
CALL user_dist_10_freq(10.0_dp, a10 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c10 | a10') / P( c10 | a10)
```

```
DO i=1, 16
    CALL user_dist_10-_freq(b10 % one_dim(i) % new_value, c10 % one_dim(i) % &
        value, new_frequency)
    CALL user_dist_10-_freq(b10 % one_dim(i) % value, c10 % one_dim(i) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a10', a10) / q(a10, a10')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a10 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a10 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a10 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a10 % one_dim(1) % value = a10 % one_dim(1) % new_value
    a10 % one_dim(1) % successful_changes = a10 % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 16
        b10 % one_dim(i2) % value = b10 % one_dim(i2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a12:

```
! calculating new value(s) for b12

b12 % one_dim(1) % new_value = 3-a12 % one_dim(1) % new_value
```

Acceptation formula for a12:

```
!  P(a12') / P(a12)

p_acc = 1
CALL user_dist_12_freq(12.0_dp, a12 % one_dim(1) % new_value, new_frequency)
CALL user_dist_12_freq(12.0_dp, a12 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c12 | a12') / P( c12 | a12)

DO i=1, 17
    CALL user_dist_12-_freq(b12 % one_dim(1) % new_value, c12 % one_dim(i) % &
```

```
        value, new_frequency)
    CALL user_dist_12-_freq(b12 % one_dim(1) % value, c12 % one_dim(i) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a12', a12) / q(a12, a12')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a12 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a12 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a12 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a12 % one_dim(1) % value = a12 % one_dim(1) % new_value
    a12 % one_dim(1) % successful_changes = a12 % one_dim(1) % &
        successful_changes + 1

    b12 % one_dim(1) % value = b12 % one_dim(1) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a14:

```
! calculating new value(s) for b14

i2 = i
b14 % one_dim(i2) % new_value = 18/a14 % one_dim(i2) % new_value
```

Acceptation formula for a14:

```
!  P(a14') / P(a14)

p_acc = 1
CALL user_dist_14_freq(14.0_dp, a14 % one_dim(i) % new_value, new_frequency)
CALL user_dist_14_freq(14.0_dp, a14 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( c14 | a14') / P( c14 | a14)

DO j=1, 19
    CALL user_dist_14-_freq(b14 % one_dim(i) % new_value, c14 % two_dim(i, j) &
        % value, new_frequency)
```

```
    CALL user_dist_14-_freq(b14 % one_dim(i) % value, c14 % two_dim(i, j) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a14', a14) / q(a14, a14')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a14 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a14 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a14 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a14 % one_dim(i) % value = a14 % one_dim(i) % new_value
    a14 % one_dim(i) % successful_changes = a14 % one_dim(i) % &
        successful_changes + 1

    i2 = i
    b14 % one_dim(i2) % value = b14 % one_dim(i2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a16:

```
! calculating new value(s) for b16

i2 = i
b16 % one_dim(i2) % new_value = 18/a16 % one_dim(i2) % new_value
```

Acceptation formula for a16:

```
!  P(a16') / P(a16)

p_acc = 1
CALL user_dist_16_freq(16.0_dp, a16 % one_dim(i) % new_value, new_frequency)
CALL user_dist_16_freq(16.0_dp, a16 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( c16 | a16') / P( c16 | a16)

DO j=1, 20
    CALL user_dist_16-_freq(b16 % one_dim(i) % new_value, c16 % two_dim(j, i) &
        % value, new_frequency)
```

```
    CALL user_dist_16-_freq(b16 % one_dim(i) % value, c16 % two_dim(j, i) % &
        value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a16', a16) / q(a16, a16')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a16 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a16 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a16 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a16 % one_dim(i) % value = a16 % one_dim(i) % new_value
    a16 % one_dim(i) % successful_changes = a16 % one_dim(i) % &
        successful_changes + 1

    i2 = i
    b16 % one_dim(i2) % value = b16 % one_dim(i2) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a18:

```
! calculating new value(s) for b18

i2 = i
DO j2 = 1, 23
    b18 % two_dim(i2, j2) % new_value = a18 % one_dim(i2) % new_value
END DO
```

Acceptation formula for a18:

```
!  P(a18') / P(a18)

p_acc = 1
CALL user_dist_18_freq(18.0_dp, a18 % one_dim(i) % new_value, new_frequency)
CALL user_dist_18_freq(18.0_dp, a18 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( c18 | a18') / P( c18 | a18)

DO j=1, 23
    CALL user_dist_18-_freq(b18 % two_dim(i, j) % new_value, c18 % two_dim(i, &
```

```
        j) % value, new_frequency)
    CALL user_dist_18-_freq(b18 % two_dim(i, j) % value, c18 % two_dim(i, j) &
        % value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a18', a18) / q(a18, a18')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a18 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a18 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a18 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a18 % one_dim(i) % value = a18 % one_dim(i) % new_value
    a18 % one_dim(i) % successful_changes = a18 % one_dim(i) % &
        successful_changes + 1

    i2 = i
    DO j2 = 1, 23
        b18 % two_dim(i2, j2) % value = b18 % two_dim(i2, j2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a20:

```
! calculating new value(s) for b20

j2 = i
DO i2 = 1, 24
    b20 % two_dim(i2, j2) % new_value = a20 % one_dim(j2) % new_value
END DO
```

Acceptation formula for a20:

```
!  P(a20') / P(a20)

p_acc = 1
CALL user_dist_20_freq(20.0_dp, a20 % one_dim(i) % new_value, new_frequency)
CALL user_dist_20_freq(20.0_dp, a20 % one_dim(i) % value, frequency)
p_acc = p_acc * new_frequency / frequency


! P( c20 | a20') / P( c20 | a20)
```

```
DO j=1, 24
    CALL user_dist_20-_freq(b20 % two_dim(j, i) % new_value, c20 % two_dim(j, &
        i) % value, new_frequency)
    CALL user_dist_20-_freq(b20 % two_dim(j, i) % value, c20 % two_dim(j, i) &
        % value, frequency)
    p_acc = p_acc * new_frequency / frequency
END DO

! q(a20', a20) / q(a20, a20')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a20 % one_dim(i) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a20 % one_dim(i) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a20 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a20 % one_dim(i) % value = a20 % one_dim(i) % new_value
    a20 % one_dim(i) % successful_changes = a20 % one_dim(i) % &
        successful_changes + 1

    j2 = i
    DO i2 = 1, 24
        b20 % two_dim(i2, j2) % value = b20 % two_dim(i2, j2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a22:

```
! calculating new value(s) for b22

b22 % one_dim(1) % new_value = 5*a22 % one_dim(1) % new_value
```

Acceptation formula for a22:

```
!  P(a22') / P(a22)

p_acc = 1
CALL user_dist_22_freq(22.0_dp, a22 % one_dim(1) % new_value, new_frequency)
CALL user_dist_22_freq(22.0_dp, a22 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c22 | a22') / P( c22 | a22)
```

```
DO i=1, 26
    DO j=1, 27
        CALL user_dist_22-_freq(b22 % one_dim(1) % new_value, c22 % &
            two_dim(i, j) % value, new_frequency)
        CALL user_dist_22-_freq(b22 % one_dim(1) % value, c22 % two_dim(i, j) &
            % value, frequency)
        p_acc = p_acc * new_frequency / frequency
    END DO
END DO

! q(a22', a22) / q(a22, a22')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a22 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a22 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a22 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a22 % one_dim(1) % value = a22 % one_dim(1) % new_value
    a22 % one_dim(1) % successful_changes = a22 % one_dim(1) % &
        successful_changes + 1

    b22 % one_dim(1) % value = b22 % one_dim(1) % new_value
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a24:

```
! calculating new value(s) for b24

DO i2 = 1, 28
    b24 % one_dim(i2) % new_value = a24 % one_dim(1) % new_value+1**2
END DO
```

Acceptation formula for a24:

```
!  P(a24') / P(a24)

p_acc = 1
CALL user_dist_24_freq(24.0_dp, a24 % one_dim(1) % new_value, new_frequency)
CALL user_dist_24_freq(24.0_dp, a24 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency

! P( c24 | a24') / P( c24 | a24)
```

```
DO i=1, 28
    DO j=1, 29
        CALL user_dist_24-_freq(b24 % one_dim(i) % new_value, c24 % &
            two_dim(i, j) % value, new_frequency)
        CALL user_dist_24-_freq(b24 % one_dim(i) % value, c24 % two_dim(i, j) &
            % value, frequency)
        p_acc = p_acc * new_frequency / frequency
    END DO
END DO

! q(a24', a24) / q(a24, a24')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a24 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a24 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a24 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a24 % one_dim(1) % value = a24 % one_dim(1) % new_value
    a24 % one_dim(1) % successful_changes = a24 % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 28
        b24 % one_dim(i2) % value = b24 % one_dim(i2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Calculating new values for functional parameters depending on a26:

```
! calculating new value(s) for b26

DO i2 = 1, 31
    b26 % one_dim(i2) % new_value = a26 % one_dim(1) % new_value+1**2
END DO
```

Acceptation formula for a26:

```
!  P(a26') / P(a26)

p_acc = 1
CALL user_dist_26_freq(26.0_dp, a26 % one_dim(1) % new_value, new_frequency)
CALL user_dist_26_freq(26.0_dp, a26 % one_dim(1) % value, frequency)
p_acc = p_acc * new_frequency / frequency
```

```
! P( c26 | a26') / P( c26 | a26)

DO i=1, 30
    DO j=1, 31
        CALL user_dist_26-_freq(b26 % one_dim(j) % new_value, c26 % &
            two_dim(i, j) % value, new_frequency)
        CALL user_dist_26-_freq(b26 % one_dim(j) % value, c26 % two_dim(i, j) &
            % value, frequency)
        p_acc = p_acc * new_frequency / frequency
    END DO
END DO

! q(a26', a26) / q(a26, a26')

CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a26 % one_dim(1) % value, frequency)
CALL user_dist_prop_freq(3.14_dp, 4.13_dp, a26 % one_dim(1) % new_value, &
    new_frequency)
p_acc = p_acc * frequency / new_frequency
```

Acceptation code for a26 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    a26 % one_dim(1) % value = a26 % one_dim(1) % new_value
    a26 % one_dim(1) % successful_changes = a26 % one_dim(1) % &
        successful_changes + 1

    DO i2 = 1, 31
        b26 % one_dim(i2) % value = b26 % one_dim(i2) % new_value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Models with all kinds of spatial dependencies

The results of all the models are presented together since all the variables have different names. Only acceptation codes with sequential update strategy are included in this section.

Acceptation code for dep01 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    dep01 % one_dim(i) % value = dep01 % one_dim(i) % new_value
    dep01 % one_dim(i) % successful_changes = dep01 % one_dim(i) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
```

```
        neighbour = spatial(i, k + 1)
        ! updating s01 % one_dim(spatial(i, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + dep01 % one_dim(spatial(neighbour, h + 1)) % &
                value
        END DO
        s01 % one_dim(neighbour) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for dep02 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    dep02 % two_dim(i, j) % value = dep02 % two_dim(i, j) % new_value
    dep02 % two_dim(i, j) % successful_changes = dep02 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating s02 % two_dim(spatial(i, k + 1), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + dep02 % two_dim(spatial(neighbour, h + 1), j) &
                % value
        END DO
        s02 % two_dim(neighbour, j) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for dep03 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    dep03 % two_dim(i, j) % value = dep03 % two_dim(i, j) % new_value
    dep03 % two_dim(i, j) % successful_changes = dep03 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(j, 1)
        sum_int = 0
        neighbour = spatial(j, k + 1)
        ! updating s03 % two_dim(i, spatial(j, k + 1))
```

```
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + dep03 % two_dim(i, spatial(neighbour, h + 1)) &
                % value
        END DO
        s03 % two_dim(i, neighbour) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depsc01 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depsc01 % one_dim(i) % value = depsc01 % one_dim(i) % new_value
    depsc01 % one_dim(i) % successful_changes = depsc01 % one_dim(i) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating sc01 % one_dim(spatial(i, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depsc01 % one_dim(spatial(neighbour, h + 1)) &
                % value
        END DO
        sc01 % one_dim(neighbour) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depcc01 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depcc01 % one_dim(i) % value = depcc01 % one_dim(i) % new_value
    depcc01 % one_dim(i) % successful_changes = depcc01 % one_dim(i) % &
        successful_changes + 1

END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Note that the value of a count variable never changes. Cases like this are not presented any more.

Acceptation code for depsc02 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depsc02 % two_dim(i, j) % value = depsc02 % two_dim(i, j) % new_value
    depsc02 % two_dim(i, j) % successful_changes = depsc02 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating sc02 % two_dim(spatial(i, k + 1), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depsc02 % two_dim(spatial(neighbour, h + 1), &
                j) % value
        END DO
        sc02 % two_dim(neighbour, j) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depsc03 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depsc03 % two_dim(i, j) % value = depsc03 % two_dim(i, j) % new_value
    depsc03 % two_dim(i, j) % successful_changes = depsc03 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(j, 1)
        sum_int = 0
        neighbour = spatial(j, k + 1)
        ! updating sc03 % two_dim(i, spatial(j, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depsc03 % two_dim(i, spatial(neighbour, h + &
                1)) % value
        END DO
        sc03 % two_dim(i, neighbour) % value = sum_int
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depd01 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depd01 % one_dim(i) % value = depd01 % one_dim(i) % new_value
    depd01 % one_dim(i) % successful_changes = depd01 % one_dim(i) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating sd01 % one_dim(spatial(i, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depd01 % one_dim(spatial(neighbour, h + 1)) % &
                value
        END DO
        sd01 % one_dim(neighbour) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i, k + 1)
        avg1 % one_dim(neighbour) % value = sd01 % one_dim(neighbour) % &
            value/cd01 % one_dim(neighbour) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depd02 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depd02 % two_dim(i, j) % value = depd02 % two_dim(i, j) % new_value
    depd02 % two_dim(i, j) % successful_changes = depd02 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating sd02 % two_dim(spatial(i, k + 1), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depd02 % two_dim(spatial(neighbour, h + 1), &
                j) % value
        END DO
        sd02 % two_dim(neighbour, j) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i, k + 1)
        avg2 % two_dim(neighbour, j) % value = REAL(sd02 % two_dim(neighbour, &
            j) % value)/cd02 % two_dim(neighbour, j) % value
    END DO
```

```
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depd03 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depd03 % two_dim(i, j) % value = depd03 % two_dim(i, j) % new_value
    depd03 % two_dim(i, j) % successful_changes = depd03 % two_dim(i, j) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(j, 1)
        sum_int = 0
        neighbour = spatial(j, k + 1)
        ! updating sd03 % two_dim(i, spatial(j, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depd03 % two_dim(i, spatial(neighbour, h + &
                1)) % value
        END DO
        sd03 % two_dim(i, neighbour) % value = sum_int
    END DO
    DO k = 1, spatial(j, 1)
        neighbour = spatial(j, k + 1)
        avg3 % two_dim(i, neighbour) % value = REAL(sd03 % two_dim(i, &
            neighbour) % value)/cd03 % two_dim(i, neighbour) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depscc01 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depscc01 % one_dim(i) % value = depscc01 % one_dim(i) % new_value
    depscc01 % one_dim(i) % successful_changes = depscc01 % one_dim(i) % &
        successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating scc01 % one_dim(spatial(i, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depscc01 % one_dim(spatial(neighbour, h + 1)) &
                % value
```

```
        END DO
        scc01 % one_dim(neighbour) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i,  k +1)
        comb1 % one_dim(neighbour) % value = scc01 % one_dim(neighbour) % &
            value+scc02 % one_dim(neighbour) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depscc02 likewise.

Acceptation code for depscc03 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depscc03 % two_dim(i, j) % value = depscc03 % two_dim(i, j) % new_value
    depscc03 % two_dim(i, j) % successful_changes = depscc03 % two_dim(i, j) &
        % successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(i, 1)
        sum_int = 0
        neighbour = spatial(i, k + 1)
        ! updating scc03 % two_dim(spatial(i, k + 1), j)
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depscc03 % two_dim(spatial(neighbour, h + 1), &
                j) % value
        END DO
        scc03 % two_dim(neighbour, j) % value = sum_int
    END DO
    DO k = 1, spatial(i, 1)
        neighbour = spatial(i,  k +1)
        comb2 % two_dim(neighbour, j) % value = scc03 % two_dim(neighbour, j) &
            % value+scc04 % two_dim(neighbour, j) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depscc04 likewise.

Acceptation code for depscc05 (if the update strategy is sequential):

```
IF (p_acc > prob(prob_index)) THEN
    depscc05 % two_dim(i, j) % value = depscc05 % two_dim(i, j) % new_value
```

```
    depscc05 % two_dim(i, j) % successful_changes = depscc05 % two_dim(i, j) &
        % successful_changes + 1

    ! 1 to number of neighbours
    DO k = 1, spatial(j, 1)
        sum_int = 0
        neighbour = spatial(j, k + 1)
        ! updating scc05 % two_dim(i, spatial(j, k + 1))
        DO h = 1, spatial(neighbour, 1)
            sum_int = sum_int + depscc05 % two_dim(i, spatial(neighbour, h + &
                1)) % value
        END DO
        scc05 % two_dim(i, neighbour) % value = sum_int
    END DO
    DO k = 1, spatial(j, 1)
        neighbour = spatial(j,  k +1)
        comb3 % two_dim(i, neighbour) % value = scc05 % two_dim(i, neighbour) &
            % value+scc06 % two_dim(i, neighbour) % value
    END DO
END IF
prob_index = prob_index + 1
IF (prob_index > SIZE(prob) .OR. prob_index < 1) THEN
    CALL G05FAF(0.0_dp, 1.0_dp, SIZE(prob), prob)
    prob_index = 1
END IF
```

Acceptation code for depscc06 likewise.