Generators for Pseudo-Random Number Distributions

Aleksandar Donev

January 2000

Contents

1 Module RandomNumbers .................................................. 1
  1.1 Uniform Random Integer Number Generator ....................... 2
  1.1.1 Seeds for the Generators ...................................... 2
  1.1.2 Random or Unpredictable Seed Initialization .................. 2
  1.1.3 Generator of Uniformly-Distributed Integers .................. 5
  1.2 Array Generators .................................................. 8
  1.2.1 Procedure RandomBits .......................................... 9
  1.2.2 Explicit-shape array arguments ................................ 11
  1.2.3 Assumed-shape array arguments ................................. 12
  1.3 Uniform Real Number Generator .................................. 14
  1.3.1 Scalar Generators ............................................. 15
  1.3.2 Explicit-shape array generators ............................... 15
  1.3.3 Assumed-shape generators ..................................... 16
  1.4 Normal-Distribution Generator .................................. 17
  1.4.1 Scalar generator ............................................... 18
  1.4.2 Array generators .............................................. 19

2 Main program .......................................................... 19

3 Formatting rules for HPF/F90 files ................................ 23
1  Module RandomNumbers

This module has functions for generating pseudorandom numbers from several distributions, including uniform and gaussian (normal) variates. The macros developed in this web file can be used to extend this module for any other generator, so long as there is a scalar Fortran 90 generator, in an almost trivial way. There are functions for generating both integer (where appropriate) and single- or double-precision reals. There are also functions for generating either single numbers or arrays of various ranks. These have all been overloaded with generic interfaces and are thus very easy to use. The basic uniform integer generators are 32 and 64 bit shift-based generators borrowed from Allan Miller's Fortran 90 website.

The module is intended to be portable to architectures that support 32 and 64 bit integers, regardless of what the default word size is. The module Precision is used, and the single and double precision kinds for reals and integers, as well as the kinds i<sub>32</sub> and i<sub>64</sub> for 32 and 64 bit integers are used carefully. The decisions on which precision to use is mostly controlled with the macros:

NB tells how many bits to use in the generator when generating real numbers (both single and double precision in this version of this file). This should almost always be set to the number of bits in the base integer unless you wanna wait for days for the generator to finish.

SP controls how many bits are used in the generator for single-precision integers. This should be set to the actual BIT_SIZE(0_sp) to avoid problems.

DP is the same as SP but for double precision integers. It should be set either to BIT_SIZE(0_dp) (better) or to SP (to speed the generator at the cost of some conversions).

The following are for a Pentium machine:

"RandomDistributions.f90" 1.0.0.1 ≡
@m NB 32
@m SP 32
@m DP 64

MODULE RandomNumbers
  USE Precision
  IMPLICIT NONE
PRIVATE
PUBLIC :: RandomUniform, RandomNormal, RandomSeeds, UnpredictableSeeds, RandomBits
  ( GlobalVariables 1.1.1.1 )
  ( GenericInterfaces 1.2.1.3 )
CONTAINS
  ( InitializeSeeds 1.1.2.1 )
  ( UniformInteger 1.1.3.5 )
  ( ArraysOfUniformIntegers 1.2.1.2 )
  ( UniformReal 1.3.1 )
  ( ArraysOfUniformReals 1.3.2 )
  ( NormalReal 1.4.1 )
  ( ArraysOfNormalReals 1.4.2 )
END MODULE RandomNumbers
At some places we generate code that we need to discard:

```
"RandomDistributions.f90" 1.0.0.2 ≡
  @0 UnusedCode.f90
  (UnusedCode 1.2.1.1)
```

### 1.1 Uniform Random Integer Number Generator

The basis of this module is the generator of uniformly distributed pseudo-random integers. Here I use L’Ecuyer’s random number generator, downloaded from Alan Miller’s website http://www.ozemail.com.au/milleraj (2000). These are shift-based generator which generate integers with either 32 or 64 bits random, written in very portable Fortran 90. They are of rather good quality and pretty fast too, but may need to be supplemented for special applications, such as Monte Carlo:

#### 1.1.1 Seeds for the Generators

The 32-bit generator requires 4 integer seeds, the 64-bit one 5 seeds. These are **PRIVATE** global variables in the module and are given initial default values. Also global variables are the normalization constants for conversion from the pseudo-random integer to real numbers:

```
"RandomDistributions.f90" 1.1.1.1 ≡
  @IM NS SIFELSE (NB, 32, 4, 5)  // Number of seeds
  (GlobalVariables 1.1.1.1) ≡
  INTEGER (KIND = i_32), DIMENSION (4), SAVE :: seeds32 = (/ 153587801_l32, -759022222_l32,
                      1288503317_l32, 1718083407_l32 /)  // Initial seeds
  INTEGER (KIND = i_64), DIMENSION (5), SAVE :: seeds64 = (/ 153587801_l64, -759022222_l64,
                      1288503317_l64, -1718083407_l64, -123456789_l64 /)  // Initial seeds
  REAL (KIND = r_sp), PARAMETER :: normalization_r_sp = 1.0_rsp /HUGE(0&j_0&NB)
  REAL (KIND = r_dp), PARAMETER :: normalization_r_dp = 1.0_dp /HUGE(0&j_0&NB)
```

This code is used in section 1.0.0.1.

#### 1.1.2 Random or Unpredictable Seed Initialization

The integer generators require 4/5 integer seeds for initialization, which should be passed to the routines `InitializeSeeds32` or `InitializeSeeds64`. I also supply a routine `RandomSeeds` which can be called with a single integer seed to generate the 4/5 random seeds using the intrinsic `RANDOM_NUMBER`, which is initialized using a simple linear-congruence generator. If this is not OK, then use the routine `UnpredictableSeeds` which uses time information to assign unpredictable values to the seeds.
The following routines are called with specific seeds. They are private, since the details of seed initialization are platform-dependent:

```
subroutine InitializeSeeds32(i_seeds) // Modelled after Allan Miller's
    integer (kind = i32), dimension (4), intent (in) :: i_seeds // Input seeds-32 bit integers
    integer, dimension (4) :: seeds32
    if (iand(seeds32, -2_i32) == 0)
        seeds32 = i_seeds - 1023_i32
    if (iand(seeds32, -8_i32) == 0)
        seeds32 = i_seeds - 1023_i32
    if (iand(seeds32, -16_i32) == 0)
        seeds32 = i_seeds - 1023_i32
    if (iand(seeds32, -32_i32) == 0)
        seeds32 = i_seeds - 1023_i32
end subroutine InitializeSeeds32

subroutine InitializeSeeds64(i_seeds) // Modelled after Allan Miller's
    integer (kind = i64), dimension (5), intent (in) :: i_seeds // Input seeds-64 bit integers
    integer, dimension (5) :: seeds64
    if (iand(seeds64, -2_i64) == 0)
        seeds64 = i_seeds - 8388607_i64
    if (iand(seeds64, -512_i64) == 0)
        seeds64 = i_seeds - 8388607_i64
    if (iand(seeds64, -4096_i64) == 0)
        seeds64 = i_seeds - 8388607_i64
    if (iand(seeds64, -131072_i64) == 0)
        seeds64 = i_seeds - 8388607_i64
    if (iand(seeds64, -8388608_i64) == 0)
        seeds64 = i_seeds - 8388607_i64
end subroutine InitializeSeeds64
```

See also sections 1.1.2.2 and 1.1.2.3.

This code is used in section 1.0.0.1.
Generators for Pseudo-Random Number Distributions

This routine only requires a single seed and uses the intrinsic RANDOM_NUMBER to initialize the seeds (for both the 32- and the 64-bit generators). It is PUBLIC, since its interface is platform-independent, though the implementation will depend on the processor. Here I use a simple serial linear-congruence generator from Numerical Recipes to initialize the internal generator and then use the intrinsic to generate the seeds for my new generator. I guess you can say we can just ignore the intrinsic and use the linear-congruence generator, but on a parallel platform, for example, the intrinsic is safer to use.

⟨InitializeSeeds 1.1.2.1⟩ +≡

SUBROUTINE RandomSeeds(seed)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: seed      // Must be default integer
  INTEGER :: i   // A counter
  INTEGER :: n_seeds  // How many seeds?
  INTEGER, DIMENSION(:,), ALLOCATABLE :: intrinsic_seeds
  /* Unfortunately the intrinsic generators only works with REAL arguments, so we need some
   conversions and temporaries: */
  INTEGER, DIMENSION(5) :: i_seeds
  REAL, DIMENSION(5) :: r_seeds   // Real seeds

  CALL RANDOM_SEED(SIZE = n_seeds)
  ALLOCATE (intrinsic_seeds(n_seeds))
  IF (n_seeds > 0) THEN
    intrinsic_seeds_1 = ABS(seed)  // Must be positive
    DO i = 2, n_seeds
      intrinsic_seeds_i = MOD(8121 * intrinsic_seeds_{i-1} + 28411, 134456)  // From 6.4 in Chapman
    END DO
  END IF

  CALL RANDOM_SEED(PUT = intrinsic_seeds)
  CALL RANDOM_NUMBER(r_seeds)  // Use the intrinsic
  i_seeds = INT(HUGE(1_i32) * r_seeds)
  CALL InitializeSeeds32(INT(i_seeds(1:4), i_32))
  CALL InitializeSeeds64(INT(i_seeds, i_64))

END SUBROUTINE RandomSeeds
This routine uses clock information to generate an unpredictable number to initialize the seeds via a call to 
\textit{RandomSeeds} with a time-dependent seed (like C's \textit{randomize}). It is \textit{public}, and is fully platform-independent. I prefer to use this one when working to initialize the seeds.

\texttt{(InitializeSeeds 1.1.2.1) \equiv}

\begin{verbatim}
SUBROUTINE UnpredictableSeeds()
    IMPLICIT NONE
    INTEGER :: unpredictable_seed       // Seed for call to RandomSeeds
    INTEGER :: times(8)                 // The current date and time-must be default kind
    CALL DATE_AND_TIME(VALUES = times)
    /* I use a made-up random expression to make a seed from the time information. Feel free to
       change this if you know better */
    unpredictable_seed = times_0 * (times_2 * times_3 + times_5) + times_7 * times_5 + times_8 * times_6
    /* Maximum value is 83626 */
    CALL RandomSeeds(unpredictable_seed)
END SUBROUTINE UnpredictableSeeds
\end{verbatim}

1.1.3 Generator of Uniformly-Distributed Integers

This subroutines generate a single random integer with 32/64 random bits. They are the basis of this module and known to have good properties and are extremely fast because they are based on bit shifts (Reference: L'Ecuyer, P. (1999) "Tables of maximally equidistributed combined LFSR generators", Math. of Comput., 68, p.p. 261-269). The cycle length for the 32-bit generator is claimed to be about $2^{113}$ or about $10^{34}$. 
These are macro that generate a single 32/64-bit random integer. I make these macros in order to be able to inline them in some of the routines to avoid calling overhead.

"RandomDistributions.f90" 1.1.3.1

@m UNIFORMINTEGER32(number,...)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds32_1, 6), seeds32_1), -13)
  seeds32_1 = IOR(ISSHFT(IAND(seeds32_1, -2_i32), 18), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds32_2, 2), seeds32_2), -27)
  seeds32_2 = IOR(ISSHFT(IAND(seeds32_2, -8_i32), 2), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds32_3, 13), seeds32_3), -21)
  seeds32_3 = IOR(ISSHFT(IAND(seeds32_3, -16_i32), 7), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds32_4, 3), seeds32_4), -12)
  seeds32_4 = IOR(ISSHFT(IAND(seeds32_4, -128_i32), 13), temp_uniform)
  number = IOR(IOR(IOR(seeds32_1, seeds32_2), seeds32_3), seeds32_4)
  #. // This can execute additional commands if needed

@m UNIFORMINTEGER64(number,...)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds64_1, 1), seeds64_1), -53)
  seeds64_1 = IOR(ISSHFT(IAND(seeds64_1, -2_i64), 10), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds64_2, 24), seeds64_2), -50)
  seeds64_2 = IOR(ISSHFT(IAND(seeds64_2, -512_i64), 5), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds64_3, 3), seeds64_3), -23)
  seeds64_3 = IOR(ISSHFT(IAND(seeds64_3, -4096_i64), 29), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds64_4, 5), seeds64_4), -24)
  seeds64_4 = IOR(ISSHFT(IAND(seeds64_4, -131072_i64), 23), temp_uniform)
  temp_uniform = ISHFT(IOR(ISSHFT(seeds64_5, 3), seeds64_5), -33)
  seeds64_5 = IOR(ISSHFT(IAND(seeds64_5, -8388608_i64), 8), temp_uniform)
  number = IOR(IOR(IOR(seeds64_1, seeds64_2), seeds64_3), seeds64_4, seeds64_5)
  #. // This can execute additional commands if needed

@m UNIFORMINTEGERNB(number,-NB)
  UniformInteger##_NB(number) // A shortcut-workaround

Now I use these two basic macros to make many routines. Here is a shortcut for the required temporary declaration for the above generators (_NB is the number of bits in the generator):

"RandomDistributions.f90" 1.1.3.2

@m UNIFORMINTEGERTEMP(_NB)
  INTEGER(KIND = i__NB): temp_uniform // Local temporary needed by UniformInteger
The base generator \texttt{UniformInteger} generates an integer in the full range of values for the given type. The generically overloaded routine \texttt{RandomUniform} contained in this module accepts an optional argument \texttt{range = (/ a, b /)} which specifies the interval \([a, b]\) in which the returned integers should be located (the length of this interval should be small compared to \texttt{HUGE(1&i_NB)}). If not provided, no normalization is performed on the numbers returned by \texttt{UniformInteger}. The normalization is done using purely integer arithmetic to speed it up, at the risk of overflows and such:

Here is the declaration of the range for the integer routines:

\begin{verbatim}
"RandomDistributions.f90" 1.1.3.4 ≡
@m _DeclareINTEGERRange(_kind)
  integer (_kind = _kind), dimension (2), intent (in), optional :: range
@m _NormalizationINTEGERTOINTEGER(variable, _kind, range, _NB, ...)
  if (present(range)) then  // Perform normalization
    variable = abs(variable) / (huge(1&i_NB) / (range_2 - range_1 + 1)) + range_1
    // This avoids real arithmetic
  end if
@m _UniformINTEGERRange(number, _kind, range, _NB, ...)
  _UniformInteger##_NB(number)
@m _NormalizationINTEGERTOINTEGER(number, _kind, range, _NB)
\end{verbatim}

Now we actually make subroutines that will generate a normalized scalar integer, using the above macro \texttt{UniformIntegerRange} to actually do the work, and we include versions for both single and double precision integers in the production code:

\begin{verbatim}
"RandomDistributions.f90" 1.1.3.5 ≡
@m _UniformINTEGERPROCEDURE(_kind, _NB)
  subroutine UniformScalar##_kind(number, range)
    implicit none
    integer (_kind = _kind), intent (out) :: number
    _DeclareINTEGERRange(_kind)
    _UniformINTEGERTemp(_NB)  // Declare the temporary
    _UniformINTEGERRange(number, _kind, range, _NB)
    // Generate an integer with 32 bits random
  end subroutine  // UniformScalar for integers
(UniformInteger 1.1.3.5) ≡
@m _UniformINTEGERPROCEDURE(_.sp, _sp)  // Single-word integers using sr-bit generator
@m _UniformINTEGERPROCEDURE(_.dp, _dp)  // Double-word integers using dp-bit generator
\end{verbatim}

See also section 1.1.3.6.

This code is used in section 1.0.0.1.
The normalization performed in the above routines is an added overhead branch or calculation. I provide a generic overloaded routine `RandomBits` which simply generates a single scalar or an array of integers using the shift-generator without any normalization. These are pretty fast and can be used for maximum efficiency. Again, I include both single and double precision versions:

```
"RandomDistributions.f90" 1.1.3.6 ⇔
@mx RANDOMBItScalar(_NB, _kind)
  subroutine RandomBitScalar(_& _kind (number)
    implicit none
    integer (kind = _kind), intent (out) :: number
    _UniformIntegerTemp(_NB)     // Declare the temporary
    _UniformIntegerNB(number, _NB) // Generate a default integer using shift generator
  end subroutine   // RandomBitScalar

(UniformInteger 1.1.3.5) ⇔
  _RandomBitScalar(sp, isp)
  _RandomBitScalar(dp, idp)
```

### 1.2 Array Generators

Now I make routines for generating arrays of integers. Here we have many conflicts between efficiency and simplicity, and I try my best not to go overboard with either one. The first set of routines, use an assumed-size array for efficiency and in order to avoid certain overheads, and have `UniformInteger` inlined. At least under Linux LF95 this is more efficient. Then I give a set of routines which use assumed-size arrays. These are all implemented for several ranks and are overloaded with a generic interface.

The important thing to notice is that the macros given below can be used to alter things such as inlining or assumed-size/shape choices very easily.
1.2.1 Procedure RandomBits

The following macro creates a routine that generates an array of random values. It uses an assumed-size argument, and the produced routine can accept additional arguments, such as a range or distribution parameters. The scalar generator can also take additional parameters which are just passed through in this case. This one will not work if there are no AdditionalArguments, but I am not sure how to fix this in a simple and elegant way:

"RandomDistributions.f90" 1.2.1 ≡
@m _GenerateRandomArray(_type, kind, AdditionalArguments, AdditionalDeclarations, ScalarDeclarations, ScalarGenerator, ≠ )

SUBROUTINE RandomBitArray(kind, n_elements, AdditionalArguments)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n_elements
>Type(kind = kind), DIMENSION(n_elements), INTENT(out) :: array
AdditionalDeclarations  // Additional arguments
INTEGER :: element  // A counter and temporary
!ScalarDeclarations(*)

DO element = 1, n_elements
!ScalarGenerator(array_element, #)  // Call the scalar-generator macro
END DO

END SUBROUTINE  // RandomBitArray

Now I implement a version in which UniformScalar is inlined. Now, if we include range here than the loop in RandomBitArray would contain a branch, which although most compilers should be able to handle, would in general degrade performance. Therefore we do not use this version in the production code:

(UnusedCode 1.2.1.1) ≡
 GenerateRandomArray(INTEGER, i_sp, range, DeclareIntegerRange(i_sp),
 UniformIntegerTemp(i_sp), UniformIntegerRange, i_sp, range, sp)
 GenerateRandomArray(INTEGER, i_dp, range, DeclareIntegerRange(i_dp),
 UniformIntegerTemp(i_dp), UniformIntegerRange, i_dp, range, dp)

This code is used in section 1.0.0.2.
For efficiency reasons, the routine RandomBitArray for integers does not process range (even though it accepts it) and therefore performs no normalization. It will be used later on, but it is **private**, so nobody will know of this small trick. It also inlines the scalar generator, so it is pretty efficient. Since we must have an additional argument, I include a *method* argument, which may later be used to select the desired method if more are available. It is **optional** and so it does not have to be included (but the call will have more overhead!). So here is the version that I adopt as efficient enough:

"RandomDistributions.f90" 1.2.1.2

```fortran
@m _DeclareMethod integer, intent(in), optional :: method

(_ArraysOfUniformIntegers 1.2.1.2) ≡

  _generateRandomArray(integer, isp, method, _DeclareMethod,
                     _UniformIntegerTemp, _UniformIntegerNB, sp)

  _generateRandomArray(integer, idp, method, _DeclareMethod,
                     _UniformIntegerTemp, _UniformIntegerNB, dp)
```

See also sections 1.2.3.2 and 1.2.3.3.

This code is used in section 1.0.0.1.

I will also make these fast no-normalization routines publicly available overloaded generically, simply because they are fast and efficient and the user can use them for other purposes, such as extending this module. They simply generate single values or arrays of random single or double precision integers using the shift-generators.

(_GenericInterfaces 1.2.1.3) ≡

```fortran
interface RandomBits
  module procedure RandomBitScalar_isp
  module procedure RandomBitScalar_idp
  module procedure RandomBitArray_isp
  module procedure RandomBitArray_idp
end interface
```

See also sections 1.3.3.2 and 1.4.2.1.

This code is used in section 1.0.0.1.
1.2.2 Explicit-shape array arguments

Now the tricky part comes when we try to write easy-to-use generic routines with assumed-shape or explicit-shape arrays (both are guaranteed to be contiguous). I give two versions, one simple, which should be, but may not be, efficient for contiguous arguments, and one complicated which should work well for non-contiguous array sections.

The macro `GENERATERANDOMARRAYSCONTIGUOUS` creates a subroutine that accepts an explicit-shape array argument of a given rank and then calls an array which accepts a rank-1 assumed-size argument. This version should be rather efficient for contiguous arguments. The plan here was to actually make the argument assumed-shape, in which case the compiler should be smart enough not to generate large overheads if the actual argument is continuous, but it seems that LF95 generates unnecessary copy-in overhead, so I decided to alter this a bit. In a sense, these routines simply become wrappers so that we generically overload this routine. I allow for an `AdditionalProcessing` processing to avoid normalizations inside the loops:

```f90
"RandomDistributions.f90" 1.2.2
@m GENERATERANDOMARRAYSCONTIGUOUS(_type, _kind, _rank, _distribution, AdditionalArguments, AdditionalDeclarations, ArrayGenerator, AdditionalProcessing, ≠ )

SUBROUTINE _distribution&ArrayContiguous&rank&_kind(array, _VARSEQUENCE(n_extent, 1, _rank), AdditionalArguments)

IMPLICIT NONE
INTEGER, INTENT(IN):: _VARSEQUENCE(n_extent, 1, _rank)
_TYPE(_kind = _kind), DIMENSION(_VARSEQUENCE(n_extent, 1, _rank)), INTENT_OUT:: array
// Assumed-shape

AdditionalDeclarations

CALL #!ArrayGenerator(array, SIZE(array))
#!AdditionalProcessing(array, #.)

END SUBROUTINE // RandomDistributionArrayContiguous
```
1.2.3 Assumed-shape array arguments

Now I give a routine which uses an assumed-shape array argument and nested do loops to traverse it and call a scalar generator for each element of the array. It is sort of like a replacement for elemental (which can not be used for non pure routines. It should in most likelihood work faster for non-contiguous arrays with not too large strides. But it’s really a very formidable macro construction. Here one can too inline the scalar generator, but things get complicated when one does that. For example, when an optional range argument is included in the uniform generators, conditional normalization is performed. The conditional should really be taken out of the loop nest by splitting the loop into two duplicate versions. But I choose not to go overboard...

This routine uses nested do loops to traverse the assumed-shape arrays, and these are generated using rather entangled calls to the internal macros _NestedLoopStart and _NestedLoopEnd:

"RandomDistributions.f90" 1.23 ≡
@m _GENERATERANDOMARRAYSLOOPS(_TYPE, Kind, _rank, Distribution, AdditionalArguments, AdditionalDeclarations, ScalarGenerator, AdditionalProcessing, ≠ )
SUBROUTINE _distribution&ArrayLoops&rank&Kind(array, AdditionalArguments)
IMPLICIT NONE
_TYPE(Kind = Kind), DIMENSION(FULLEXTENT(_rank)), INTENT(OUT) :: array
AdditionalDeclarations
INTEGER :: _VARSEQUENCE(i, 1, _rank)
_NestedLoopStart (i, array, _rank)
!ScalarGenerator(array(_VARSEQUENCE(i, 1, _rank)), #)  // This can be a macro
_NestedLoopEnd (_rank)
!AdditionalProcessing(array, #)
END SUBROUTINE  // RandomDistributionArrayLoops

The typical (albeit somewhat inefficient) usage of the above macro will be to have as a the scalar generator a simple call to the subroutine that works on scalars. This macro is to be used as the scalar generator in this case:

"RandomDistributions.f90" 1.23.1 ≡
@m _CALL _UNIFORMSCALAR(number, Kind, ...)
call UniformScalar##kind(numiber, #)
Now we use the above template to generate routines for array arguments of various ranks. We do this for arrays of rank 1-3, just so we don’t create too much code:

"RandomDistributions.f90" 1.2.3.2
@m MAX_RANK 3
<ArraysOfUniformIntegers 1.2.1.2> +≡
&do (RANK, 1, MAX_RANK)
{
  GENERATERANDOMARRAYSCONTIGUOUS(INTEGER, i_sp, RANK, Uniform, range,
    DECLAREINTEGERRANGE(i_sp), RandomBitArray.i_sp, NORMALIZATIONINTEGERTOINTEGER,
    i_sp, range, sp)
  GENERATERANDOMARRAYSCONTIGUOUS(INTEGER, i_dp, RANK, Uniform, range,
    DECLAREINTEGERRANGE(i_dp), RandomBitArray.i_dp, NORMALIZATIONINTEGERTOINTEGER,
    i_dp, range, dp)
}

In the production module I do not really inline the scalar generator, but to avoid the branch overhead associated with the call to UniformScalar and the normalization, here instead I use RandomBits as a scalar generator and perform the normalization in a second pass, using the same macro as above (thanks to array syntax!). Of course, this has the disadvantage of doing two load/stores per value. Just replace GENERATERANDOMBITSCALAR with CALL_UNIFORMSCALAR if you want the other version:

"RandomDistributions.f90" 1.2.3.3
@m CALL_RANDOMBITSCALAR(number, kind, range, NB)
    call RandomBitScalar.@kind(number)  // We call the generic routine in this case
<ArraysOfUniformIntegers 1.2.1.2> +≡
&do (RANK, 1, MAX_RANK)
{
  GENERATERANDOMARRAYSLOOPs(INTEGER, i_sp, RANK, Uniform, range,
    DECLAREINTEGERRANGE(i_sp), CALL_RANDOMBITSCALAR,
    NORMALIZATIONINTEGERTOINTEGER, i_sp, range, sp)
  GENERATERANDOMARRAYSLOOPs(INTEGER, i_dp, RANK, Uniform, range,
    DECLAREINTEGERRANGE(i_dp), CALL_RANDOMBITSCALAR,
    NORMALIZATIONINTEGERTOINTEGER, i_dp, range, dp)
}
Now we make the generic interfaces for these integer functions. Since the interface is different for the routines with explicit-shape and assumed-shape arrays (the array dimensions are passed as well for explicit-shape), we include both versions in the generic routine:

\[
\langle \text{InterfaceUniformInteger 1.2.3.4} \rangle \equiv
\]

\[
\text{MODULE PROCEDURE UniformScalar_i_sp}
\]

\[
\text{MODULE PROCEDURE UniformScalar_i_dp}
\]

\[
\text{MODULE PROCEDURE UniformArrayContiguous_@RANK @i_j_sp}
\]

\[
\text{MODULE PROCEDURE UniformArrayContiguous_@RANK @i_j_dp}
\]

\[
\text{MODULE PROCEDURE UniformArrayLoops_@RANK @i_j_sp}
\]

\[
\text{MODULE PROCEDURE UniformArrayLoops_@RANK @i_j_dp}
\]

This code is used in section 1.3.3.2.

### 1.3 Uniform Real Number Generator

I will use the integer number generator to generate real numbers via normalization. Here is the macro to perform this normalization:

"RandomDistributions.f90" 1.3 \equiv

\[
\text{@m NORMAIZATIONINTEGERToREAL(variable, \_kind, range, i\_variable, ...)}
\]

\[
\text{IF (PRESENT(range)) THEN} \quad // \text{Perform normalization}
\]

\[
\text{variable} = 0.5\#\_kind \times ((\text{range}_2 - \text{range}_1) \times \text{REAL(i\_variable, \_kind)} \times \text{normalization}\#\_kind + (\text{range}_1 + \text{range}_2))
\]

\[
\text{ELSE}
\]

\[
\text{variable} = 0.5\#\_kind \times (\text{REAL(i\_variable, \_kind)} \times \text{normalization}\#\_kind + 1.0\#\_kind)
\]

\[
\text{END IF}
\]

The routines for generating a real random number are now very easy to compose using all the macros we have developed so far. The design follows closely the integer example:

"RandomDistributions.f90" 1.3.0.1 \equiv

\[
\text{@m UNIFORMREALRANGE(\_kind, number, range, i\_number, NB, ...)}
\]

\[
\text{UniformInteger}\#\#NB(i\_number)
\]

\[
\text{NORMAIZATIONINTEGERToREAL(number, \_kind, range, i\_number, NB)
\]

\[
\text{@m UNIFORMREALTEMP(i\_temp, NB)
\]

\[
\text{UNIFORMINTEGERTEMP(NB)}
\]

\[
\text{INTEGER (kind = i\#\_NB)} :: i\_temp\quad // \text{Local temporaries needed by UNIFORMReal}
\]

\[
\text{@m DECLAREREALRANGE(\_kind)}
\]

\[
\text{REAL (kind = \_kind), \ DIMENSION (2), \ INTENT (IN), \ OPTIONAL :: range}\quad // \text{Default integer range}
\]
1.3.1 Scalar Generators

Here is the actual subroutine that will generate a single or double precision real scalar:

"RandomDistributions.f90" 1.3.1 ≡
@m _UniformRealProcedure(_kind, NB, ...)

subroutine UniformScalar_##_kind(number, range)
 implicit NONE
REAL(kind = _kind), intent(out) :: number
_DeclareRealRange(_kind)
_UniformRealTemp(_number, _NB)
_UniformRealRange(_kind, number, range, _number, _NB)
#  // This can execute additional commands if needed
end subroutine  // UniformScalar for real numbers

<UniformReal 1.3.1> ≡
_UniformRealProcedure(r_sp, NB)  // Use default integers in the generator
_UniformRealProcedure(r_dp, NB)

This code is used in section 1.0.0.1.

1.3.2 Explicit-shape array generators

Just as for integers, here I include versions for explicit-shape arrays. But unlike the integer case, I do not inline the scalar generator or try to do anything about the branch in the normalization. First we create the routines for generating a one-dimensional assumed-size array of random real numbers (feel free to optimize further):

<ArraysOfUniformReals 1.3.2> ≡
"GenerateRandomArray(real, r_sp, range, _DeclareRealRange(r_sp), _Dummy, _CallUniformScalar, r_sp, range)
GenerateRandomArray(real, r_dp, range, _DeclareRealRange(r_dp), _Dummy, _CallUniformScalar, r_dp, range)

See also sections 1.3.2.1 and 1.3.3.

This code is used in section 1.0.0.1.

And now we can use the above to make the explicit-shape routines:

<ArraysOfUniformReals 1.3.2> +≡
 ado(RANK, 1, _MAX_RANK)
 { GenerateRandomArraysContiguous(real, r_sp, RANK, Uniform, range,
 _DeclareRealRange(r_sp), RandomBitArray_r_sp, _Dummy, r_sp, range, _NB)
 GenerateRandomArraysContiguous(real, r_dp, RANK, Uniform, range,
 _DeclareRealRange(r_dp), RandomBitArray_r_dp, _Dummy, r_dp, range, _NB) }
1.3.3 Assumed-shape generators

Again, following the integer approach, it is trivial to include versions for assumed-shape array arguments that use a nested loop nest to traverse the array and call the scalar generator.

\[
\{\text{ArraysOfUniformReals 1.3.2}\} \equiv \\
\begin{array}{l}
\text{MODULE PROCEDURE UniformScalar}_r \sp \\
\text{MODULE PROCEDURE UniformScalar}_d \\
\end{array}
\$DO (RANK, 1, MAX_RANK)\\n\begin{array}{l}
\text{GENERATERANDOMARRAYSLOOPS(REAL, r}_sp, \text{ RANK}, \text{ Uniform, range,} \\
\quad \text{DECLAREREALRANGE}(r}_sp, \text{ CALL UNIFORMSCALAR, } \text{Dummy, r}_sp, \text{ range)}\\n\text{GENERATERANDOMARRAYSLOOPS(REAL, r}_dp, \text{ RANK, Uniform, range,} \\
\quad \text{DECLAREREALRANGE}(r}_dp, \text{ CALL UNIFORMSCALAR, } \text{Dummy, r}_dp, \text{ range)}
\end{array}
\]

Finally, I add these to the generic interface of RandomUniform:

\[
\{\text{InterfaceUniformReal 1.3.3.1}\} \equiv \\
\begin{array}{l}
\text{MODULE PROCEDURE UniformArrayContiguous}_r \sp \\
\text{MODULE PROCEDURE UniformArrayContiguous}_d \\
\text{MODULE PROCEDURE UniformArrayLoops}_r \sp \\
\text{MODULE PROCEDURE UniformArrayLoops}_d
\end{array}
\$DO (RANK, 1, MAX_RANK)\\n\begin{array}{l}
\text{\text{INTERFACE RandomUniform} \\
\quad \text{\{ InterfaceUniformInteger 1.2.3.4\}} \\
\quad \text{\{ InterfaceUniformReal 1.3.3.1\}}
\end{array}
\]

END INTERFACE

This code is used in section 1.3.3.2.
1.4 Normal-Distribution Generator

Simply because I use these frequently in my codes, I include routines for generating random real numbers drawn from a normal distribution,

\[ p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right) \]

This is very closely modelled after Allan Miller’s code, which uses the ratio of uniforms method of A.J. Kinderman and J.F. Monahan augmented with quadratic bounding curves. The mean \( x_0 \) and the standard deviation \( \sigma \) are controlled by an optional argument \( \text{mean}_\text{std} = [x_0, \sigma] \) in the routines included under the generic \textit{RandomNormal}:

"RandomDistributions.f90“ 1.4 ≡

@m _DECLAREREALMEANSTD(jkind, ...)

REAL (kind = jkind), DIMENSION (2), INTENT (IN), OPTIONAL :: mean_std
1.4.1 Scalar generator

Here is the scalar generator, along with two specific instances for single and double precision reals which we include in the module. I should say that the additional code was only for single precision values, so some of the parameters here are not given to sufficient precision:

"RandomDistributions.f90" 1.4.1 ≡
@m _NORMALREALPROCEDURE( _kind, ...)
SUBROUTINE NormalScalar_@@_kind(number, mean_std)
  IMPLICIT NONE
  REAL( KIND = _kind ), INTENT (OUT) :: number
  _DECLARE_REALMEANSTD(_kind) // mean_std = ( _mean, std ) /)
  REAL(_kind), PARAMETER :: s = 0.449871*_kind, t = -0.386595*_kind,
  a = 0.19600*_kind, b = 0.25472*_kind, r1 = 0.27597*_kind, r2 = 0.27846*_kind,
  r = 1.7156*_kind
  REAL(_kind) :: u, v, x, y, q
  DO
    CALL UniformScalar_@@_kind(u) // P = (u,v) is in the unit square
    CALL UniformScalar_@@_kind(v)
    v = r*(v - 0.5*_kind)
    x = u - s
    y = ABS(u) - t
    q = x^2 + y*(a*y - b*x)
    IF (q < r1)
      EXIT // Accept P if inside inner ellipse
    IF (q > r2)
      CYCLE // Reject P if outside outer ellipse
    IF (v^2 < -4.0*_kind*log(u)*(u^2))
      EXIT // Accept P if inside acceptance region
  END DO
  number = v / u // A normal deviate centered at 0 with std of 1
IF (PRESENT(mean_std)) THEN // Re-normalize to new mean and std.
  number = mean_std(2)*number + mean_std(1)
END IF
END SUBROUTINE // NormalScalar for real numbers

/NormalReal 1.4.1 ≡
  _NORMALREALPROCEDURE(r_dp)
  _NORMALREALPROCEDURE(r_dp)

This code is used in section 1.0.0.1.
1.4.2 Array generators

Just as in the case for uniform reals, we create routines that work on array arguments. Here I do not include versions for explicit-shape arrays, since this routine is slow anyway, so any overheads associated with assumed-shape arrays are not visible. The normal routines are also overloaded with a generic interface:

```
"RandomDistributions.f90" 1.4.2 ≡
    @m  _CALL_NORMALSCALAR(number, _kind, ...)
    CALL NormalScalar##_kind(number, #.)

<ArraysOfNormalReals 1.4.2> ≡
  do (RANK, 1, MAX_RANK)
  {
    GENERATERANDOMARRAYS(loops(REAL, r_sp, RANK, Normal, mean_std, 
    DECLAREREALMEANSTD(r_sp), _CALL_NORMALSCALAR, _DUMMY, r_sp, mean_std)
    GENERATERANDOMARRAYS(loops(REAL, r_dp, RANK, Normal, mean_std, 
    DECLAREREALMEANSTD(r_dp), _CALL_NORMALSCALAR, _DUMMY, r_dp, mean_std)
  }

This code is used in section 1.0.0.1.

Now we make the generic interfaces for these real functions. Notice that I choose only the nested-do assumed-shape version for the generic routine:

<GenericInterfaces 1.2.13> ≡
  INTERFACE RandomNormal
    MODULE PROCEDURE NormalScalar_r_sp
    MODULE PROCEDURE NormalScalar_r_dp
  d0 (RANK, 1, MAX_RANK)
  {
    MODULE PROCEDURE NormalArrayLoops_r& RANK & r_sp
    MODULE PROCEDURE NormalArrayLoops_r& RANK & r_dp
  }
END INTERFACE

2 Main program

This program tests the above module. It uses the StopWatch module from NIST to time the execution of certain routines. It always runs the array routines with range present for normal routines (and writes these to a file named "test.dat" along with uniform generates), but for the other tests you can choose the type (i or r) and precision (sp or dp) here:

"RandomDistributions.f90" 2 ≡
  @m  _TYPE r
  @m  _PR sp
Here are timing results for single-precision (word-size) integers and reals on a Linux Pentium machine running Red Hat:

- **Integers:**
  
  Times printed by StopWatch:
  
  RandomBits or random_number:
  
  \[
  \begin{array}{ccc}
  \text{cpu} & \text{user} & \text{sys} \\
  0.85 & 0.83 & 0.02 \\
  \\
  \text{RandomUniform-contiguous rank 1:} & \text{cpu} & \text{sys} \\
  0.81 & 0.81 & 0.00 \\
  \\
  \text{RandomUniform-contiguous rank 2:} & \text{cpu} & \text{sys} \\
  0.88 & 0.85 & 0.03 \\
  \\
  \text{RandomUniform-loops rank 1:} & \text{cpu} & \text{sys} \\
  1.44 & 1.44 & 0.00 \\
  \\
  \text{RandomUniform-loops rank 2:} & \text{cpu} & \text{sys} \\
  1.41 & 1.41 & 0.00 \\
  \\
  \text{RandomUniform-with range, rank 1:} & \text{cpu} & \text{sys} \\
  2.59 & 2.58 & 0.01 \\
  \\
  \text{RandomNormal-with mean_std:} & \text{cpu} & \text{sys} \\
  8.16 & 8.12 & 0.04 \\
  \end{array}
  \]

- **Reals:**
  
  Times printed by StopWatch:
  
  RandomBits or random_number:
  
  \[
  \begin{array}{ccc}
  \text{cpu} & \text{user} & \text{sys} \\
  0.51 & 0.48 & 0.03 \\
  \\
  \text{RandomUniform-contiguous rank 1:} & \text{cpu} & \text{sys} \\
  2.00 & 1.99 & 0.01 \\
  \\
  \text{RandomUniform-contiguous rank 2:} & \text{cpu} & \text{sys} \\
  2.09 & 2.09 & 0.00 \\
  \\
  \text{RandomUniform-loops rank 1:} & \text{cpu} & \text{sys} \\
  2.02 & 2.01 & 0.01 \\
  \\
  \text{RandomUniform-loops rank 2:} & \text{cpu} & \text{sys} \\
  1.98 & 1.98 & 0.00 \\
  \\
  \text{RandomUniform-with range, rank 1:} & \text{cpu} & \text{sys} \\
  2.37 & 2.37 & 0.00 \\
  \\
  \text{RandomNormal-with mean_std:} & \text{cpu} & \text{sys} \\
  8.14 & 8.11 & 0.03 \\
  \end{array}
  \]
And here is the promised main program:

```
"RandomDistributions.f90" 2.2 ==

PROGRAM Random_Test
USE Precision, only: dp, wp, dp_, wp
USE RandomNumbers
USE StopWatch
IMPLICIT NONE

INTEGER, PARAMETER :: i_wp = i_0&_PR, r_wp = r_0&_PR  // Working precisions
INTEGER, PARAMETER :: n_points = 1000000  // Number of points to generate
INTEGER, PARAMETER :: n_reps = 10  // Number of repetitions of timing loop
INTEGER :: point, reps  // Counters
INTEGER (KIND = i_wp) :: i_number  // A generated integer number
INTEGER (KIND = i_wp), DIMENSION (n_points) :: i_array_1  // An array of random integers
INTEGER (KIND = i_wp), DIMENSION (n_points / 10, 10) :: i_array_2  // Rank 2 integer
REAL (KIND = r_wp) :: r_number  // A generated real number
REAL (KIND = r_wp), DIMENSION (n_points) :: r_array_1  // An array of random reals
REAL (KIND = r_wp), DIMENSION (n_points / 10, 10) :: r_array_2  // Rank 2 real
REAL (KIND = r_wp), DIMENSION (n_points) :: n_array  // Normal variates

 TYPE (WATCHTYPE), DIMENSION (7) :: watch

CALL CREATE_WATCH ( watch, name = (/ "RandomBits or random_number " , &
"RandomUniform-contiguous rank 1 " , &
"RandomUniform-contiguous rank 2 " , &
"RandomUniform-loops rank 1 " , &
"RandomUniform-loops rank 2 " , &
"RandomUniform-with range, rank 1 " , &
"RandomNormal-with mean_std " )

CALL UnpredictableSeeds()

DO reps = 1, n_reps
  CALL START_WATCH (watch(1))
  IFELSE (_type, i, CALL RandomBits(_type@wp_array_1, n_points), CALL RANDOM_NUMBER(_type@wp_array_1) )
  CALL STOP_WATCH (watch(1))
  CALL START_WATCH (watch (2))
  CALL RandomUniform(_type@wp_array_1, n_points)
  CALL STOP_WATCH (watch(2))
  CALL START_WATCH (watch(3))
  CALL RandomUniform(_type@wp_array_2, n_points / 10, 10)
  CALL STOP_WATCH (watch(3))
  CALL START_WATCH (watch(4))
  CALL RandomUniform(_type@wp_array_1)
  CALL STOP_WATCH (watch(4))
  CALL START_WATCH (watch(5))
  CALL RandomUniform(_type@wp_array_2)
  CALL STOP_WATCH (watch(5))
  CALL START_WATCH (watch(6))
  IFELSE (_type, i, CALL RandomUniform(i_array_1, range=/3.i_wp,5.i_wp/) , CALL RandomUniform(r_array_1, range=/3.0_r_wp,5.0_r_wp/) )
```
CALL STOP_WATCH (watch(6))
CALL START_WATCH (watch(7))
CALL RandomNormal(n_array, mean_std = (/ 10.0_wp, 2.0_wp /))
CALL STOP_WATCH (watch(7))

END DO
CALL PRINT_WATCH (watch)

DO point = 1, min(10000, n_points)
   WRITE (UNIT = 10, FMT = *) _TYPE@&_array@ (point), n_array(point)
END DO

END PROGRAM Random_Test

[HFPF2Formatting.hweb]
3 Formatting rules for HPF/F90 files

[HPP2Formatting.hweb] These are just same auxiliary formatting rules and useful macros I use from time to time.

@m _GENERICINTERFACE(generic_name, ...)
    INTERFACE generic_name
    MODULE PROCEDURE #.
    END INTERFACE generic_name
@m _DECLARE_IWORD(...)
    INTEGER :: #.
@m _DECLARE_IWP(...)
    INTEGER (KIND = i_wp) :: #.
@m _DECLARE_R_WP(...)
    REAL (KIND = r_wp) :: #.
@m _DECLARE_R_SP(...)
    REAL (KIND = r_sp) :: #.
@m _DECLARE_R_DP(...)
    REAL (KIND = r_dp) :: #.
@m _FULLEXTENT(_rank) : $DO (DIM, 2, _rank) { , }$
@m _VARIABLESEQUENCE(_variable, _start, _end)
    _variable##_start$DO (DIM, $EVAL (_start + 1), _end) { , _variable@DIM }$
@m _NESTEDLOOPSTART(_variable, _array, _rank)
    $DO (DIM, _rank, 1, -1) { DO _variable@DIM = LBOUND(_array, DIM), UBOUND(_array, DIM) }$
@m _NESTEDLOOPEND(_rank)$DO (DIM, 1, _rank) { END DO }$
@m _DUMMY(...