

Open Scientific Library Reference Manual

Jason R. Blevins

*Department of Economics
Duke University*

Jason R. Blevins
Department of Economics
Duke University
213 Social Sciences Building
Box 90097
Durham NC 27708-0097
USA
<http://jblevins.org/>

This document was typeset using L^AT_EX.

Copyright © 2006-2009 Jason Blevins.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<http://libosl.org/>

Contents

Contents	iii
1 Introduction	1
1.1 Routines Available in OSL	1
1.2 No Warranty	1
2 The Open Scientific Library	2
2.1 Library Structure	2
2.2 Dependencies	2
2.3 Conventions	2
2.4 Generic Interfaces	3
2.5 Common Functional Interfaces	6
2.6 Error Handling	7
2.7 Thread Safety	7
3 OSL Core	8
3.1 Precision Constants	8
3.2 Mathematical Constants	10
3.3 Formatted Output	10
3.4 Miscellaneous Mathematical Functions	11
3.5 IEEE Floating Point Procedures	11
3.6 Vector Operations	11
3.7 Low-Level Operations	12
4 Pseudo-Random Number Generation	13
4.1 Initialization and Seeding	14
4.2 Random Number Generators	14
4.3 Sampling From a Random Number Generator	14
4.4 Sampling From Specific Distributions	15
5 Probability Distributions	18
5.1 The Normal Distribution	18
5.2 The Uniform Distribution	19
5.3 The Exponential Distribution	21
5.4 The Gumbel Distribution	22
6 Statistics	24
6.1 Sample Statistics	24
6.2 Markov-Chain Approximations to AR(1) Processes	27
7 Simulated Annealing	28

8	Numerical Differentiation	32
8.1	Finite-Difference Gradient	32
8.2	Finite-Difference Jacobian	33
8.3	Finite-Difference Hessian	34
9	Special Functions	36
9.1	Factorial and Related Functions	36
10	Permutations	39
10.1	Limitations	40
11	Sorting	41
11.1	Quicksort	41
11.2	Shell Sort	41
A	GNU Free Documentation License	43
A.1	Applicability and Definitions	43
A.2	Verbatim Copying	45
A.3	Copying in Quantity	45
A.4	Modifications	45
A.5	Combining Documents	47
A.6	Collections of Documents	47
A.7	Aggregation with Independent Works	47
A.8	Translation	48
A.9	Termination	48
A.10	Future Revisions of This License	48
A.11	Relicensing	49
A.12	Addendum: How to use this License for your documents	49
	Bibliography	50

Chapter 1

Introduction

The Open Scientific Library (OSL) is a collection of open-source routines for scientific computing, with a particular emphasis on statistics and econometrics. The routines are written in standards-compliant Fortran 95 with a modern interface geared towards both scientific researchers and software developers.

The target end user is the applied researcher with a moderately-sized, dense problem. The algorithms contained in OSL are intended to be both usable and understandable. Efficiency is important, but achieving the utmost level of performance is secondary to transparency, portability and usability.

1.1 Routines Available in OSL

The library covers many areas of scientific and numerical computing, including:

Random number generation	Probability distributions
Statistics	Histograms
Optimization	Markov Chain Monte Carlo
Simulated Annealing	Sorting
Numerical differentiation	Chebyshev polynomials
Solving linear and nonlinear systems	Vector and matrix operations
Interpolation	IEEE floating-point
Special functions	Code profiling

1.2 No Warranty

The software described in this manual has no warranty, it is provided “as is”. It is your responsibility to validate the behavior of the routines and their accuracy using the source code provided, or to purchase support and warranties from commercial redistributors.

Chapter 2

The Open Scientific Library

2.1 Library Structure

The OSL is structured in a very modular fashion. There is a core module which provides kind parameters for controlling precision, basic mathematical functions, and an interface for generating pseudo-random numbers. Built on top of this core are a number of modules geared towards more specific tasks. Each module depends on the core and possibly other modules. When possible, individual module procedures are further separated into distinct source files for even more targeted reuse.

This structure caters both to users who prefer to compile and link to the complete library as well as those who would rather download individual modules or procedures as needed in source form, possibly storing the files alongside the rest of their source code. Thus, the OSL is intended to be distributed both in pre-packaged, binary form and in a pick-and-choose fashion.

2.2 Dependencies

2.3 Conventions

Argument lists In most cases, inputs are listed first, followed by outputs, and then optional parameters.

Module	Dependencies
osl_core	none
osl_dist	osl_core
osl_rng	osl_core
osl_sort	osl_core
osl_stat	osl_core, osl_dist
osl_special	osl_core
osl_permutation	osl_core, osl_special
osl_combination	osl_core, osl_special
osl_diff	osl_core
osl_siman	osl_core, osl_rng
osl_func	osl_core

Table 2.1: Module Dependencies

Naming conventions For consistency, all modules relating to objects are follow standard naming conventions. For example, for a module named `osl_foo`, the relevant user-defined type is `osl_foo_t` and the method names all begin with `osl_foo` as in `osl_foo_init`, `osl_foo_print`, `osl_foo_free`, etc.

2.4 Generic Interfaces

To allow OSL routines to be used with object-oriented code, a standard generic interface must be developed. Consider the following `model` object, which could represent a complex statistical or physical which has some objective function which needs to be minimized or maximized, taking the data encapsulated in the `model` object as given (represented here by `alpha`). When evaluated at some value x , the objective function might even need to modify some data stored in the `model` while carrying out the necessary calculations to return the requested value.

```

module model_mod
  implicit none

  type :: model_t
    real :: alpha
  end type model_t

contains

  ! Objective function with a minimum at alpha
  function model_objective(self, x) result(y)
    class(model_t), intent(inout) :: self
    real, intent(in) :: x
    real :: y
    y = (x - self%alpha)**2
  end function model_objective

end module model_mod

```

The goal is to write a generic optimization module with procedures which can optimize functions like `model_objective`, but the developer of such a module cannot possibly know *a priori* each and every type that it will be used with (for example, our `model_t` type).

Consider an optimization module with the following structure:

```

module opt_mod
  implicit none

contains

  subroutine optimize(func, x)
    interface
      subroutine func(x, y)
        real, intent(in) :: x
        real, intent(out) :: y
      end subroutine func
    end interface
    real, intent(inout) :: x
    ! optimize func and return the optimum in x
  end subroutine optimize

end module opt_mod

```

The above `optimize` function is general in the sense that it can be used with any function that satisfies the `func` interface, but unfortunately this does not include object-oriented procedures such as our `model_objective`. We need a generic way to pass the object, or a pointer to the object, to the `optimize` function, which can then pass it to a generic `func` function each time it is evaluated.

Fortran 95

C programmers use typeless `void` pointers in such situations which can be casted to the appropriate type when needed. As the OSL written in *standard* Fortran 95, in order to implement generic, type-independent algorithms, it must do so within the constraints of the standard. This is moderately difficult in Fortran 95, and will be less so in Fortran 2003. Fortran 95 has no generic pointer type,¹ but the `transfer` intrinsic can achieve the same results.

The `transfer` function introduced in Fortran 90 can be used to move data of one type through procedures and variables that were expecting data of some other type. This is the sense in which `transfer` approximates the behavior of void pointers in C. Essentially, it copies the bits in memory representing a variable source to a scalar or array of the same type as a given `mol`d. The syntax for `transfer` is

```
result = transfer(source, mol[, size])
```

where `source` and `mol`d are scalars or arrays of any type and `size` is an optional scalar of type `integer`. The `result` is of the same type as `mol`d. If `size` is given, `result` is a rank-one array of length `size`. If `size` is omitted but `mol`d is an array, then `result` is an array just large enough to represent the source. Finally, if `size` is omitted and `mol`d is a scalar, then `result` is a scalar.

Therefore, an array of any type will suffice to pass the data through the `optimize` procedure to the objective function. All OSL routines use an array of type `integer`. A generic `func` interface is then

```
subroutine func(x, y, ctx)
  real, intent(in) :: x
  real, intent(out) :: y
  integer, dimension(:), intent(in), optional :: ctx
end subroutine func
```

and the corresponding `optimize` signature is

```
subroutine optimize(func, x, ctx)
```

We call the new optional argument `ctx` a context argument since the context of the object will be encoded within it. In practice, the most flexible way to provide the context is by using `transfer` to pass a pointer to the original object by creating a new type containing only a pointer:

```
type :: model_ptr_t
  type(model_t), pointer :: p
end type model_ptr_t
```

The corresponding objective function wrapper would be as follows:

```
subroutine model_objective_wrapper(x, y, ctx)
  real, intent(in) :: x
  real, intent(out) :: y
```

¹Most Fortran 95 compilers support Cray pointers, which can achieve the desired generality in practice, but Cray pointers are not defined in the Fortran 95 standard.

```

integer, dimension(:), intent(in), optional :: ctx
type(model_ptr_t) :: ptr

ptr = transfer(ctx, ptr)
call model_objective(ptr%p, x, y)
end subroutine model_objective_wrapper

```

Calling `optimize` would involve using `transfer` to encode the pointer:

```

type(model_t) :: model
type(model_ptr_t) :: ptr
ptr%p => model
integer, dimension(:) :: ctx_mold
call optimize(model_objective_wrapper, x, transfer(ptr, ctx_mold))

```

assuming `ctx_mold` is already defined, for example, as

```

integer, dimension(:), allocatable :: ctx_mold

```

OSL provides a global parameter `osl_ctx` for use as a context mold.

Fortran 2003

In Fortran 2003, new features such as unlimited polymorphic (`class(*)`) objects or C pointers (variables of `type(c_ptr)`) can be used to implement generic data structures in a more natural way. Given the same `model_mod` module from before, we will briefly preview one possible generic Fortran 2003 interface using C pointers. A context argument will again be used, but now instead of using an array of type `integer` as the medium of transport, we use a C pointer, with type `c_ptr` which is defined in the `iso_c_binding` module. The analogous generic `func` interface using C pointers would be

```

subroutine func(x, y, ctx)
  import :: c_ptr
  real, intent(in) :: x
  real, intent(out) :: y
  type(c_ptr), intent(in), optional :: ctx
end subroutine func

```

and the wrapper function would be

```

subroutine model_objective_wrapper(x, y, ctx)
  use iso_c_binding
  real, intent(in) :: x
  real, intent(out) :: y
  type(c_ptr), intent(in), optional :: ctx
  type(model_t), pointer :: modelp

  modelp = c_f_pointer(ctx)
  call model_objective(modelp, x, y)
end subroutine model_objective_wrapper

```

Calling `optimize` would involve using `c_loc` to obtain the location in memory of a `model_t` variable:

```

type(model_t) :: model
call optimize(model_objective_wrapper, x, c_loc(model))

```

The Fortran 95 and Fortran 2003 approaches are very similar conceptually. Note that we no longer require the intermediate `model_ptr_t` type or the `transfer` intrinsic, but we now use the `iso_c_binding` module and the `c_f_pointer` and `c_loc` procedures. In both cases,

some sort of wrapper is required unless the `model` object is written with the above generic interface in mind.

2.5 Common Functional Interfaces

Many OSL procedures operate on Fortran procedures which represent functions of some type. For example, the optimization routines either minimize or maximize a given function and the numeric differentiation routines calculate numeric gradients, Jacobians, or Hessians of given functions. These target functions are passed as arguments to the OSL functions that operate on them and as such we must define standard interfaces for them.

The interfaces defined by OSL are based on the dimensions of the domain and range of the function and whether the function is expected to provide derivatives or not. All functional arguments are assumed to be subroutines. The interfaces are `func`, `func_n`, `func_nd`, and `func_nm` and are defined below. For generality, and specifically for interoperability with object-oriented code, each interface specifies an optional context argument `ctx`. In all cases, `ctx` is a rank-one assumed-shape `integer` array (of the default type). With the use of the `transfer` intrinsic function, this construct can mimic the behavior of `void *` pointers in C in standard Fortran 95. For more details on the use of `transfer` in this regard, see, for example, [Blevins \(2009\)](#).

The `func` interface applies to real-valued functions of one variable $f : \mathbb{R} \rightarrow \mathbb{R}$, `func_n` is for real-valued functions of many variables $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and `func_nm` is for real vector-valued functions of many variables $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. These interfaces are defined as follows:

```
interface
  subroutine func(x, y, ctx)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), intent(in) :: x
    real(wp), intent(out) :: y
    integer, dimension(:), intent(in), optional :: ctx
  end subroutine func
end interface
```

```
interface
  subroutine func_n(x, y, ctx)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), dimension(:), intent(in) :: x
    real(wp), intent(out) :: y
    integer, dimension(:), intent(in), optional :: ctx
  end subroutine func_n
end interface
```

```
interface
  subroutine func_nm(x, y, ctx)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), dimension(:), intent(in) :: x
    real(wp), dimension(:), intent(out) :: y
    integer, dimension(:), intent(in), optional :: ctx
  end subroutine func_nm
end interface
```

For routines which require derivatives, `func_nd` describes a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that provides a gradient when requested and `func_nnd` describes a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ that provides a Jacobian when requested:

```

interface
  subroutine func_nd(x, y, df, ctx)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), dimension(:), intent(in) :: x
    real(wp), intent(out) :: y
    real(wp), dimension(size(x)), optional, intent(out) :: df
    integer, dimension(:), intent(in), optional :: ctx
  end subroutine func_nd
end interface

interface
  subroutine func_nnd(x, y, d, ctx)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), dimension(:), intent(in) :: x
    real(wp), dimension(size(x)), intent(out) :: y
    real(wp), dimension(size(x),size(x)), optional, intent(out) :: d
    integer, dimension(:), intent(in), optional :: ctx
  end subroutine func_nnd
end interface

```

2.6 Error Handling

Functions and subroutines provide an optional `status` argument in which status and error codes are returned. A non-zero status code indicates an error and a zero indicates success.

2.7 Thread Safety

The library is thread safe. Technically speaking there are a few minor exceptions regarding global variables that determine the overall library behavior (e.g., error handling). These situations should be fairly obvious and such variables should not be modified by different threads.

Chapter 3

OSL Core

All OSL modules depend on the `osl_core` module which contains kind parameters for controlling the precision of real variables, mathematical constants, and various numerical functions.

3.1 Precision Constants

Real Variables

Although OSL provides kind parameter constants for both single (`osl_sp`) and double (`osl_dp`) precision real variables, the use of “working precision” (`osl_wp`) is encouraged because it makes changing precision easier, allowing for experimentation in the present and flexibility in the future. The recommended way to declare real variables is using the `osl_wp` kind parameter for working precision via the renaming operator after the use statement as follows:

Example: Working Precision

```
program working_precision
  use osl_core, wp => osl_wp
  implicit none

  real(wp) :: x = 1.0_wp
  print *, 'The value of x is', x
end program working_precision
```

Output:

```
The value of x is 1.0000000000000000
```

`osl_wp` is a public integer constant defined in `osl_core` and is made available to programs and modules that declare `use osl_core`. The `wp => osl_wp` statement *renames* the `osl_wp` constant so that it is available as `wp`. This convention allows for compatibility with all OSL functions, which expect real variables to be of kind `osl_wp`, and it allows one to easily change the precision at any time simply by redefining the integer `wp`.

The OSL provides the following kind constants for single and double precision:

```
integer, parameter :: osl_sp = selected_real_kind(6, 37)
integer, parameter :: osl_dp = selected_real_kind(15, 307)
```

Presently, `osl_wp` is defined as an alias for double precision:

```
integer, parameter :: osl_wp = osl_dp
```

If one day quadruple precision becomes standard, converting any programs written using `real(wp)` variables is simply a matter of writing:

```
integer, parameter :: wp = selected_real_kind(33, 4931)
```

Alternatively, if one day OSL includes a quad-precision kind constant:

```
use osl_core, wp => osl_qp
```

Thus, code written using `wp` is easily portable to other precisions.

The properties of real variables can be determined using Fortran's `huge`, `tiny`, `range`, and `precision` intrinsics.

Example: Real Variables

```
program reals
use osl_core, wp => osl_wp
implicit none

real(wp) :: big = huge(1.0_wp)
real(wp) :: small = tiny(1.0_wp)

print 100, 'huge:      ', big
print 100, 'tiny:      ', small
print 100, 'range:     ', range(big)
print 100, 'precision:', precision(big)

100 format (a10,1x,g12.5)
end program reals
```

Output:

```
huge:      0.17977+309
tiny:      0.22251-307
range:           307
precision:      15
```

Integer Variables

OSL also provides integer kind parameters for declaring integer variables with a minimum number of bytes. The following constants are defined which correspond to the kind parameters for 1-, 2-, 4-, and 8-byte integers respectively:

```
integer, parameter :: osl_i1b = selected_int_kind(2)
integer, parameter :: osl_i2b = selected_int_kind(4)
integer, parameter :: osl_i4b = selected_int_kind(9)
integer, parameter :: osl_i8b = selected_int_kind(18)
```

Equivalently, these are 4-, 16-, 32-, and 64-bit integers. Like their real counterparts, these constants can be renamed in the use statement as follows:

```
use osl_core, i8b => osl_i8b
```

Constant	Value
osl_e	e
osl_pi	π
osl_pi_2	$\pi/2$
osl_pi_4	$\pi/4$
osl_2pi	2π
osl_1_pi	$1/\pi$
osl_2_pi	$2/\pi$
osl_sqrt2	$\sqrt{2}$
osl_sqrt1_2	$\sqrt{1/2}$
osl_sqrt3	$\sqrt{3}$
osl_sqrtpi	$\sqrt{\pi}$
osl_euler	Euler's constant (γ)

Table 3.1: Mathematical Constants

3.2 Mathematical Constants

The `osl_core` module contains predefined constants, or parameters in Fortran parlance, for several common mathematical constants such as π . These values are listed in Table 3.1 and are defined up to double precision.

3.3 Formatted Output

The `osl_core` module provides several functions for generating uniformly formatted output. In particular, `osl_print` serves as an interface for printing a short message, such as a variable name, followed by a scalar, vector, or matrix of type `integer` or `real(wp)`. Through the use of Fortran's interfaces, the same subroutine call is used for any of these cases as illustrated by the following examples.

Example: Formatted output via `osl_print`

```

program print
  use osl_core, wp => osl_wp
  implicit none

  integer, parameter      :: n = 2
  integer                 :: i = 7
  integer, dimension(n)  :: ivec = 7
  integer, dimension(n,n) :: imat = 7
  real(wp)                :: r = 7.0_wp
  real(wp), dimension(n) :: rvec = 7.0_wp
  real(wp), dimension(n,n) :: rmat = 7.0_wp

  call osl_print('osl_print examples:')
  call osl_print('i = ', i)
  call osl_print('ivec = ', ivec)
  call osl_print('imat = ', imat)

  call osl_print('r = ', r)
  call osl_print('rvec = ', rvec)
  call osl_print('rmat = ', rmat)
end program print

```

Output:

```

osl_print examples:
i =          7
ivec =
          7          7
imat =
          7          7
          7          7
r =          7.0000
rvec =
          7.0000          7.0000
rmat =
          7.0000          7.0000
          7.0000          7.0000

```

3.4 Miscellaneous Mathematical Functions

The Signum Function

Procedure: elemental **function** `osl_sgn(x)`**Arguments:**

- `x` - a **real**(wp) or **integer** scalar.

Return value: -1 if `x` is negative and 1 otherwise.

3.5 IEEE Floating Point Procedures

The `osl_core` module provides elemental functions for testing whether IEEE floating point numbers are finite or NaN (not-a-number). Many compiler vendors provide similar functions, but until Fortran 2003, no standard intrinsic functions were available. There are two relevant functions.

Procedure: elemental **function** `osl_is_nan(x)`**Arguments:**

- `x` - a scalar **real**(wp) variable.

Return value: `.true.` if `x` is equal to NaN and `.false.` otherwise.**Procedure:** elemental **function** `osl_is_finite(x)`**Arguments:**

- `x` - a scalar **real**(wp) variable.

Return value: `.true.` if `x` is finite (not NaN or Inf) and `.false.` otherwise.

Note: These functions will be deprecated in the future in favor of the Fortran 2003 `ieee_arithmetic` intrinsic module which contains functions `ieee_is_finite` and `ieee_is_nan`.

3.6 Vector Operations

Linearly Spaced Vectors

The `osl_linspace` function returns `n` linearly-spaced vectors between (and including) two vectors `a` and `b`.

Example: Linearly spaced vectors

```

program linspace
  use osl_core, wp => osl_wp
  implicit none

  integer, parameter :: n = 10
  integer, parameter :: dim = 2
  real(wp), dimension(dim) :: a = (/ 1, 4 /)
  real(wp), dimension(dim) :: b = (/ -3, 7 /)
  real(wp), dimension(n, dim) :: v

  v = osl_linspace(a, b ,n)
  call osl_print('v = ', v)
end program linspace

```

Output:

```

v =
      1.0000      4.0000
      0.55556    4.3333
      0.11111    4.6667
     -0.33333    5.0000
     -0.77778    5.3333
     -1.2222    5.6667
     -1.6667    6.0000
     -2.1111    6.3333
     -2.5556    6.6667
     -3.0000    7.0000

```

```

interface osl_linspace
  module procedure osl_linspace_swp
  module procedure osl_linspace_vwp
end interface

function osl_linspace_swp(a, b, n)
  real(wp), intent(in) :: a, b
  integer, intent(in) :: n
  real(wp), dimension(n) :: osl_linspace
end function osl_linspace_swp

function osl_linspace_vwp(a, b, n)
  real(wp), dimension(:), intent(in) :: a, b
  integer, intent(in) :: n
  real(wp), dimension(n,size(a)) :: osl_linspace_vwp
end function osl_linspace_vwp

```

3.7 Low-Level Operations

OSL provides several “low-level” operations for scalar, vector, and matrix variables of type `real(wp)` and `integer` (of the default kind). The subroutine `osl_swap(a,b)` exchanges the values of `a` and `b`. Calling `osl_reverse(a)` results in the values of `a` being reversed.

Chapter 4

Pseudo-Random Number Generation

The `osl_rng` module provides an object-oriented multi-distribution pseudo-random number generation interface. The module defines the `osl_rng_t` type which acts as an object and stores, among other things, the initial seed and the current state of the random number generator. With this object-oriented interface, pseudo-random numbers can be drawn from several distributions using the same entropy source. Furthermore, multiple independent random number generators can be created as needed.

Example: Random Number Generation

```
program rng
  use osl_core, only: wp => osl_wp
  use osl_rng
  implicit none

  type(osl_rng_t) :: r
  integer, parameter :: seed = 274
  real(wp) :: u

  ! Initialize and seed RNG
  call osl_rng_init(r, SEED=seed)

  ! Draw from Uniform(0,1)
  u = osl_uniform_rnd(r)
  print *, u

  ! Draw from Normal(0,1)
  u = osl_normal_rnd(r)
  print *, u

  ! Free memory
  call osl_rng_free(r)
end program rng
```

Output:

```
0.47777819994320014
-0.77016531390507914
```

4.1 Initialization and Seeding

Each `osl_rng_t` instance must be initialized by calling `osl_rng_init`.

```
subroutine osl_rng_init(self, type, seed)
  type(osl_rng_t), intent(inout) :: self
  integer, optional, intent(in) :: type
  integer, optional, intent(in) :: seed
end subroutine osl_rng_init
```

`type` is an integer which specifies an underlying uniform random number generator. Currently only the `mzran` generator is supported (type `OSL_RNG_MZRAN`). If the seed is not specified, the state is set to the default seed for the specific generator chosen.

A generic interface is provided for seeding generators through a subroutine called `osl_rng_seed` which accepts a single `integer` seed:

```
subroutine osl_rng_seed(self, seed)
  type(osl_rng_t), intent(inout) :: self
  integer, intent(in) :: seed
end subroutine osl_rng_seed
```

Random number generators of type `osl_rng_t` must also be freed by calling `osl_rng_free` once they are no longer needed to prevent memory leaks.

```
subroutine osl_rng_free(self)
  type(osl_rng_t), intent(inout) :: self
end subroutine osl_rng_free
```

4.2 Random Number Generators

mzran

The `OSL_RNG_MZRAN` generator is based the `mzran` pseudo-random number generator of Marsaglia and Zaman (1994). `mzran` uses a four-dimensional state vector and each such `osl_rng_t` instance begins with the state set to the default seed:

$$s = (521288629, 362436069, 16163801, 1131199299).$$

4.3 Sampling From a Random Number Generator

Uniform random draws of several types may be sampled from random number generators in OSL, either as integers or real numbers. These functions are considered part of the underlying uniform integer generator and thus have the `osl_rng` prefix. Functions for obtaining random draws from other distributions have the corresponding distribution prefix (e.g., `osl_normal`) and are discussed in Section 4.4.

```
function osl_rng_get(self)
  type(osl_rng_t), intent(inout) :: self
  integer(osl_i4b) :: osl_rng_get
end function osl_rng_get
```

This function returns a random 32-bit (four-byte) integer within the range of the underlying generator.

```
function osl_rng_uniform(self)
  type(osl_rng_t), intent(inout) :: self
  real(osl_wp) :: osl_rng_uniform
end function osl_rng_uniform
```

This function returns a random uniform real number in the $[0,1]$ interval.

```
function osl_rng_uniform_int(self, a, b)
  type(osl_rng_t), intent(inout) :: self
  integer, intent(in) :: a
  integer, intent(in) :: b
  integer :: osl_rng_uniform_int
end function osl_rng_uniform_int
```

This function returns a uniform integer from a to b , inclusive. It scales down random integers drawn from `osl_rng_get` and thus should not be used for drawing integers from the full range of the generator, which should be drawn by calling `osl_rng_get` directly.

Example: Sampling From a Random Number Generator

```
program rng_sampling
  use osl_core, wp => osl_wp
  use osl_rng
  implicit none

  type(osl_rng_t) :: rng
  real(wp) :: u
  integer :: i

  ! initialize a generator of the default type
  call osl_rng_init(rng)

  ! draw a random 32-bit integer
  i = osl_rng_get(rng)
  print *, i

  ! draw a uniform real number between 0 and 1
  u = osl_rng_uniform(rng)
  print *, u

  ! draw a integer between 1 and 10
  i = osl_rng_uniform_int(rng, 1, 10)
  print *, i

  ! free the generator
  call osl_rng_free(rng)
end program rng_sampling
```

Output:

```
-1721637130
0.79823845366438206
10
```

4.4 Sampling From Specific Distributions

Uniform Distribution

`osl_uniform_rnd` is an interface for generating draws from the continuous $\text{Uniform}(a,b)$ distribution. Without any additional parameters, it generates draws from $\text{Uniform}(0,1)$:

```

function osl_uniform_rnd(rng, a, b)
  type(osl_rng_t), intent(inout) :: rng
  real(wp), optional, intent(in) :: a
  real(wp), optional, intent(in) :: b
  real(wp) :: osl_uniform_rnd
end function osl_uniform_rnd

```

Normal Distribution

Draws from the standard Normal distribution are generated using the Marsaglia Polar method. The algorithm generates these draws in pairs so one of them is stored and returned at the next call.

```

function osl_normal_rnd(rng)
  type(osl_rng_t), intent(inout) :: rng
  real(wp) :: osl_normal_rnd
end function osl_normal_rnd

```

Exponential Distribution

Draws from the exponential distribution with parameter λ can be generated using `osl_exponential_rnd`:

```

function osl_exponential_rnd(rng, lambda)
  type(osl_rng_t), intent(inout) :: rng
  real(wp), intent(in) :: lambda
  real(wp) :: osl_exponential_rnd
end function osl_exponential_rnd

```

Gumbel Distribution

Draws from the Gumbel, or type I extreme value, distribution can be generated using `osl_gumbel_rnd`. These draws are generated using an inverse cdf transformation. The parameters μ and β are optional with default values `0.0_wp` and `1.0_wp` respectively.

```

function osl_gumbel_rnd(self, mu, beta)
  type(osl_rng_t), intent(inout) :: self
  real(wp), optional, intent(in) :: mu
  real(wp), optional, intent(in) :: beta
  real(wp) :: osl_gumbel_rnd
end function osl_gumbel_rnd

```

General Discrete Distributions

The subroutine `osl_rng_discrete` draws a sample from a general discrete probability distribution on the integers $\{1, 2, \dots, n\}$ with weights given by p . Enough draws are returned to fill the given vector x . The weights need not sum to one because they will be normalized internally. This procedure uses the alias method of Walker (1977). See also Kronmal and Peterson (1979).

Note that this subroutine can also be used to sample vectors, say, from the rows of a matrix M . Among other things this is useful for resampling observations from a data matrix during a Bootstrap procedure (Horowitz, 2001). The sample returned in x can be used as row indices to sample random rows from M :

```
p = 1.0_wp / dble(n)
call osl_rng_discrete(rng, p, x)
sample = M(x, :)
```

Chapter 5

Probability Distributions

The module `osl_dist` provides functions related to probability distributions. This includes evaluating the cumulative distribution function (cdf), probability density function (pdf), and quantile function (inverse cdf) of several common distributions. These functions are named, with a distribution-specific tag such as `normal` followed by, respectively, the suffixes `_pdf`, `_cdf`, and `_inv`. Routines for drawing random numbers from such distributions are contained in the `osl_rng` module (Chapter 4) and follow a similar naming convention (with suffix `_rnd`).

Distribution	PDF	CDF	Quantile Function	Section
Normal	<code>osl_normal_pdf</code>	<code>osl_normal_cdf</code>	<code>osl_normal_inv</code>	5.1
Uniform	<code>osl_uniform_pdf</code>	<code>osl_uniform_cdf</code>	<code>osl_uniform_inv</code>	5.2
Exponential	<code>osl_exponential_pdf</code>	<code>osl_exponential_cdf</code>	<code>osl_exponential_inv</code>	5.3
Gumbel	<code>osl_gumbel_pdf</code>	<code>osl_gumbel_cdf</code>	<code>osl_gumbel_inv</code>	5.4

Table 5.1: Probability distributions and suffixes

This module depends only on the `osl_core` module.

5.1 The Normal Distribution

The Normal distribution with mean μ and standard deviation σ has pdf

$$\phi(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

The cdf, $\Phi(x; \mu, \sigma)$ has no known closed form. The standard Normal distribution has mean zero and standard deviation one.

`osl_normal_pdf` returns the value of the pdf of the Normal distribution with mean `mu` and standard deviation `sigma`. Similarly `osl_normal_cdf` and `osl_normal_inv` evaluate the cdf and quantile functions respectively. For each of these functions, `mu` and `sigma` are optional. The default default parameters `mu = 0` and `sigma = 1` correspond to those of the standard Normal distribution.

```
real(wp) function osl_normal_pdf(x, mu, sigma)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: sigma
end function osl_normal_pdf
```

```

real(wp) function osl_normal_cdf(x, mu, sigma)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: sigma
end function osl_normal_cdf

real(wp) function osl_normal_inv(p, mu, sigma)
  real(wp), intent(in) :: p
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: sigma
end function osl_normal_inv

```

Example: Normal Distribution

```

program normal
  use osl_core, wp => osl_wp
  use osl_dist, only: osl_normal_pdf, osl_normal_cdf, osl_normal_inv

  ! Quantiles of interest
  real(wp), parameter, dimension(5) :: tau = &
    (/ 0.025_wp, 0.05_wp, 0.5_wp, 0.95_wp, 0.975_wp /)

  integer :: i
  real(wp) :: z

  ! Print quantiles, pdf, and cdf values
  print '(4a8)', 'quantile', 'z', 'pdf', 'cdf'
  do i = 1, size(tau)
    z = osl_normal_inv(tau(i))
    print '(4f8.3)', tau(i), z, osl_normal_pdf(z), osl_normal_cdf(z)
  end do
end program normal

```

Output:

quantile	z	pdf	cdf
0.025	-1.960	0.058	0.025
0.050	-1.645	0.103	0.050
0.500	0.000	0.399	0.500
0.950	1.645	0.103	0.950
0.975	1.960	0.058	0.975

5.2 The Uniform Distribution

The uniform distribution on the interval $[a, b]$ has pdf

$$f(x; a, b) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b, \\ 0 & \text{otherwise,} \end{cases}$$

and cdf

$$F(x; a, b) = \begin{cases} 0 & \text{if } x < a, \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b, \\ 1 & \text{if } x > b. \end{cases}$$

The following functions evaluate the pdf, cdf, and quantile functions, respectively. In each case, the parameters *a* and *b* are optional with default values `0.0_wp` and `1.0_wp` respectively, corresponding to the standard Uniform(0,1) distribution.

```
real(wp) function osl_uniform_pdf(x, a, b)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: a
  real(wp), intent(in), optional :: b
end function osl_uniform_pdf
```

```
real(wp) function osl_uniform_cdf(x, a, b)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: a
  real(wp), intent(in), optional :: b
end function osl_uniform_cdf
```

```
real(wp) function osl_uniform_inv(p, a, b)
  real(wp), intent(in) :: p
  real(wp), intent(in), optional :: a
  real(wp), intent(in), optional :: b
end function osl_uniform_inv
```

Example: Uniform Distribution

```
program uniform
  use osl_core, wp => osl_wp
  use osl_dist, only: osl_uniform_pdf, osl_uniform_cdf, osl_uniform_inv

  ! Quantiles of interest
  real(wp), parameter, dimension(5) :: tau = &
    (/ 0.025_wp, 0.05_wp, 0.5_wp, 0.95_wp, 0.975_wp /)

  integer :: i
  real(wp) :: z

  ! Print quantiles, pdf, and cdf values
  print '(4a8)', 'quantile', 'z', 'pdf', 'cdf'
  do i = 1, size(tau)
    z = osl_uniform_inv(tau(i))
    print '(4f8.3)', tau(i), z, osl_uniform_pdf(z), osl_uniform_cdf(z)
  end do
end program uniform
```

Output:

quantile	z	pdf	cdf
0.025	0.025	1.000	0.025
0.050	0.050	1.000	0.050
0.500	0.500	1.000	0.500
0.950	0.950	1.000	0.950
0.975	0.975	1.000	0.975

5.3 The Exponential Distribution

The exponential distribution has a single parameter—the rate parameter λ —and has pdf

$$f(x; \lambda) = \begin{cases} \lambda \exp(-\lambda x) & \text{if } x \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

and cdf

$$F(x; \lambda) = \begin{cases} 1 - \exp(-\lambda x) & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The pdf, cdf, and inverse cdf functions can be evaluated using the following functions. The parameter is λ required in each.

```
real(wp) function osl_exponential_pdf(x, lambda)
  real(wp), intent(in) :: x
  real(wp), intent(in) :: lambda
end function osl_exponential_pdf
```

```
real(wp) function osl_exponential_cdf(x, lambda)
  real(wp), intent(in) :: x
  real(wp), intent(in) :: lambda
end function osl_exponential_cdf
```

```
real(wp) function osl_exponential_inv(p, lambda)
  real(wp), intent(in) :: p
  real(wp), intent(in) :: lambda
end function osl_exponential_inv
```

Example: Exponential Distribution

```
program exponential
  use osl_core, wp => osl_wp
  use osl_dist

  ! Parameter
  real(wp), parameter :: lambda = 0.5_wp

  ! Quantiles of interest
  real(wp), parameter, dimension(5) :: tau = &
    (/ 0.025_wp, 0.05_wp, 0.5_wp, 0.95_wp, 0.975_wp /)

  integer :: i
  real(wp) :: z

  ! Print quantiles, pdf, and cdf values
  print '(4a8)', 'quantile', 'z', 'pdf', 'cdf'
  do i = 1, size(tau)
    z = osl_exponential_inv(tau(i), lambda)
    print '(4f8.3)', tau(i), z, &
      osl_exponential_pdf(z, lambda), &
      osl_exponential_cdf(z, lambda)
  end do
end program exponential
```

Output:

quantile	z	pdf	cdf
0.025	0.051	0.487	0.025
0.050	0.103	0.475	0.050
0.500	1.386	0.250	0.500
0.950	5.991	0.025	0.950
0.975	7.378	0.013	0.975

5.4 The Gumbel Distribution

The Gumbel distribution, or type I extreme value distribution, has two parameters μ and β and has pdf

$$f(x; \mu, \beta) = \frac{1}{\beta} \exp\left(-\frac{x - \mu}{\beta}\right) \exp\left[-\exp\left(-\frac{x - \mu}{\beta}\right)\right]$$

and cdf

$$F(x; \mu, \beta) = \exp[-\exp(-(x - \mu)/\beta)].$$

The inverse cdf, or quantile function, for $p \in (0, 1)$ is

$$F^{-1}(p; \mu, \beta) = \mu - \beta \ln[-\ln p].$$

The pdf, cdf, and inverse cdf functions can be evaluated using the following functions.

```
real(wp) function osl_gumbel_pdf(x, mu, beta)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: beta
end function osl_gumbel_pdf
```

```
real(wp) function osl_gumbel_cdf(x, mu, beta)
  real(wp), intent(in) :: x
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: beta
end function osl_gumbel_cdf
```

```
real(wp) function osl_gumbel_inv(p, mu, beta)
  real(wp), intent(in) :: p
  real(wp), intent(in), optional :: mu
  real(wp), intent(in), optional :: beta
end function osl_gumbel_inv
```

Example: Gumbel Distribution

```
program gumbel
  use osl_core, wp => osl_wp
  use osl_dist

  ! Quantiles of interest
  real(wp), parameter, dimension(5) :: tau = &
    (/ 0.025_wp, 0.05_wp, 0.5_wp, 0.95_wp, 0.975_wp /)
```

```
integer :: i
real(wp) :: z

! Print quantiles, pdf, and cdf values
print '(4a8)', 'quantile', 'z', 'pdf', 'cdf'
do i = 1, size(tau)
  z = osl_gumbel_inv(tau(i))
  print '(4f8.3)', tau(i), z, osl_gumbel_pdf(z), osl_gumbel_cdf(z)
end do
end program gumbel
```

Output:

quantile	z	pdf	cdf
0.025	-1.305	0.092	0.025
0.050	-1.097	0.150	0.050
0.500	0.367	0.347	0.500
0.950	2.970	0.049	0.950
0.975	3.676	0.025	0.975

Chapter 6

Statistics

The `osl_stat` module provides functions related to statistics, besides those relating directly to statistical distributions already provided by `osl_dist`.

Among the more specialized routines are the `osl_tauschen` subroutine (section 6.2) which implements the method developed by [Tauschen \(1986\)](#) for constructing finite-state Markov chain approximations to AR(1) processes.

6.1 Sample Statistics

Sample Mean

The `osl_mean` function computes the mean of a vector or the mean of the rows of a matrix.

Example: Sample Mean

```
program mean
  use osl_core, wp => osl_wp
  use osl_stat
  use osl_rng

  type(osl_rng_t) :: rng
  integer, parameter :: T = 5
  real(wp), dimension(T) :: vec
  real(wp), dimension(T,2) :: mat
  integer :: i

  call osl_rng_init(rng)
  do i = 1, T
    vec(i) = osl_uniform_rnd(rng)
    mat(i,1) = 0.5_wp + osl_normal_rnd(rng)
    mat(i,2) = -1.0_wp + osl_normal_rnd(rng)
  end do
  call osl_rng_free(rng)

  call osl_print('vec:', vec)
  call osl_print('mean(vec):', osl_mean(vec))

  call osl_print('mat:', mat)
  call osl_print('mean(mat):', osl_mean(mat))
end program mean
```

Output:

```

vec:
  0.99150E-01    0.23148    0.88828    0.90027    0.21780
mean(vec):      0.46740
mat:
  -1.1749    -0.82838
  0.94341    0.75324
  1.4379    -0.32099
  1.4346    -1.0468
  0.54118    1.1925
mean(mat):
  0.63643    -0.50079E-01

```

Sample Variance

The variance of a sample can be calculated using `osl_var`. This function accepts either a real vector argument x of length n , representing n observations on a scalar random variable X , or a real matrix argument Y of dimension $n \times k$ representing a sample of n observations on a random vector of dimension k . Either version of the function also accepts an optional mean which will be used to more efficiently calculate the variance if provided. Otherwise, the mean will be calculated internally.

In the case of a scalar random variable X , `osl_var` returns the sample variance

$$s_X = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

where \bar{x} denotes the sample mean. Similarly, in the case of a vector random variable Y , `osl_var` returns the sample variance matrix

$$S_Y = \frac{1}{n-1} \sum_{i=1}^n (Y_i - \bar{Y})(Y_i - \bar{Y})^T.$$

Example: Sample Variance

```

program var
  use osl_core, wp => osl_wp
  use osl_stat
  use osl_rng

  type(osl_rng_t) :: rng
  integer, parameter :: T = 100, N = 2
  real(wp), dimension(T) :: x
  real(wp), dimension(T,N) :: Y
  real(wp) :: var_x
  real(wp), dimension(N) :: mean_Y
  real(wp), dimension(N,N) :: var_Y
  integer :: i, j

  call osl_rng_init(rng)
  do i = 1, T
    x(i) = osl_normal_rnd(rng)
    do j = 1, N
      Y(i,j) = osl_normal_rnd(rng)
    end do
  end do

```

```

end do
call osl_rng_free(rng)

var_x = osl_var(x)
print '("var(x):", g17.5)', var_x

mean_Y = osl_mean(Y)
var_Y = osl_var(Y, MEAN=mean_Y)
print '("var(Y):", 2g17.5,/,7x,2g17.5)', var_Y
end program var

```

Output:

```

var(x):      0.75548
var(Y):      0.62558          0.91996E-01
              0.91996E-01          1.0971

```

Sample Covariance

The behavior of the `osl_cov` function is analogous to that of `osl_var` in that it can operate on samples on either scalar or vector random variables and in either case, the means may be provided for efficiency purposes.

In the case of samples on two random vectors X and Y , `osl_cov` returns the sample covariance

$$S_{XY} = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})^T.$$

Example: Sample Covariance

```

program cov
use osl_core, wp => osl_wp
use osl_stat
use osl_rng

type(osl_rng_t) :: rng
integer, parameter :: T = 100, N = 2
real(wp), dimension(T,N) :: x, y
real(wp), dimension(N,N) :: cov_xy
real(wp) :: cov_x1y1
integer :: i, j

call osl_rng_init(rng)
do i = 1, T
  do j = 1, N
    x(i,j) = osl_normal_rnd(rng)
    y(i,j) = 0.5_wp * x(i,1) + 0.5_wp * osl_normal_rnd(rng)
  end do
end do
call osl_rng_free(rng)

cov_x1y1 = osl_cov(x(:,1), y(:,1))
print '("cov(x1, y1):", 2g17.5,/,12x,2g17.5)', cov_x1y1

```

```

cov_xy = osl_cov(x, y)
print ('cov(x, y): ",2g17.5,/,12x,2g17.5)', cov_xy
end program cov

```

Output:

```

cov(x1, y1):      0.37941
cov(x, y):        0.37941      -0.61157E-02
                  0.31667      -0.65161E-01

```

6.2 Markov-Chain Approximations to AR(1) Processes

The `osl_tauschen` subroutine constructs a discrete Markov chain approximation to a given AR(1) process using the method of Tauchen (1986). This code is based that in the online appendix of Aruoba, Fernández-Villaverde, and Rubio-Ramírez (2006).

Example: Tauchen's Procedure

```

program tauschen
  use osl_core, wp => osl_wp
  use osl_stat, only: osl_tauschen
  implicit none

  integer, parameter :: n = 5
  real(wp), parameter :: rho = 0.7_wp
  real(wp), parameter :: sigma = 0.25_wp
  real(wp), parameter :: cover = 0.5_wp
  real(wp), dimension(n) :: z
  real(wp), dimension(n, n) :: pi

  call osl_tauschen(rho, sigma, n, cover, z, pi)
  call osl_print('States =', z)
  call osl_print('Transition matrix = ', pi)
end program tauschen

```

Output:

```

States =
  -0.17504      -0.87518E-01      0.00000E+00      0.87518E-01      0.17504
Transition matrix =
  0.48604      0.13761      0.12337      0.97978E-01      0.15500
  0.38972      0.13819      0.13488      0.11661      0.22060
  0.29976      0.13077      0.13895      0.13077      0.29976
  0.22060      0.11661      0.13488      0.13819      0.38972
  0.15500      0.97978E-01      0.12337      0.13761      0.48604

```

Chapter 7

Simulated Annealing

The module `osl_siman` provides an implementation of the Simulated Annealing algorithm of [Corana, Marchesi, Martini, and Ridella \(1987\)](#) for derivative-free global minimization of a multidimensional function. For a detailed description of the algorithm and its suggested usage, see [Goffe, Ferrier, and Rogers \(1992\)](#). Additionally, [Goffe, Ferrier, and Rogers \(1994\)](#) discuss its application to problems in econometrics.

An object of type `osl_siman_t` stores all of the input parameters and results. First, the object must be initialized with a call to `osl_siman_init`. Although the starting value `x` and the initial temperature `temp` are the only required arguments, this subroutine also accepts many optional arguments (denoted `...`) which are described in detail below. The `osl_siman_init` interface is as follows:

```
subroutine osl_siman_init(self, x, temp, ...)
  type(osl_siman_t), intent(inout) :: self
  real(wp), dimension(:), intent(in) :: x
  real(wp), intent(in) :: temp
end subroutine osl_siman_init
```

After initialization, any parameters of the `osl_siman_t` object may be modified by calling `osl_siman_set` which accepts the same arguments as `osl_siman_init` with the exception that `x` and `temp` are also optional:

```
subroutine osl_siman_set(self, ...)
  type(osl_siman_t), intent(inout) :: self
end subroutine osl_siman_set
```

The `osl_siman_opt` subroutine carries out the actual optimization. The only *required* arguments are an initialized `osl_siman_t` object and a function to be optimized which adheres to the `func_n` interface. All other parameters, such as bounds, convergence criteria, etc. should be set prior to calling `osl_siman_opt` either via `osl_siman_init` or `osl_siman_set`. The optimal value of `x` and the functional value at the optimum may optionally be returned via the optional `x_opt` and `f_opt` keyword arguments. The Fortran interface is:

```
subroutine osl_siman_opt(self, func_n, x_opt, f_opt)
  type(osl_siman_t), intent(inout) :: self
  real(wp), dimension(self%dim), intent(out), optional :: x_opt
  real(wp), intent(out), optional :: f_opt

interface
  subroutine func_n(x, y)
    use osl_core, wp => osl_wp
    implicit none
    real(wp), dimension(:), intent(in) :: x
```

```

    real(wp), intent(out) :: y
  end subroutine func_n
end interface
end subroutine osl_siman_opt

```

Here, `self%dim` is the dimension of `x`.

The optional arguments to `osl_siman_init` and `osl_siman_set` and their default values are given below.

- **real(wp) :: temp = 10.0_wp**
The initial temperature. This is a function-dependent value and requires some experimentation. [Goffe et al. \(1994\)](#) for advice.
- **logical :: max = .false.**
Indicates whether the function should be maximized or minimized. A value of `.true.` indicates that the function should be maximized. Otherwise, it will be minimized.
- **real(wp), dimension(size(x,1)) :: step**
Step length vector. It should cover the region of interest around the starting value `x`. On iteration i , from the point x^i , the next trial point is selected from the interval between $x^i - step^i$ and $x^i + step^i$. Since `step` is adjusted so that about half of all points are accepted, the input value is not very important (i.e. if the value is off, `step` is adjusted to the correct value).
- **logical, dimension(size(x,1)) :: fix**
Logical indicators for components of `x` to hold fixed.
- **real(wp), dimension(size(x,1)) :: lb**
Lower bound on the set of possible `x` values. If the algorithm chooses a trial value of `x` such that $x_k < lb_k$ or $x_k > ub_k$, for component k , a point within the bounds for that component is randomly chosen. For effective unconstrained optimization, simply leave `lb` and `ub` at their default values.
- **real(wp), dimension(size(x,1)) :: ub**
Upper bound on the set of possible `x` values.
- **real(wp) :: alpha = 0.85_wp**
The rate of temperature reduction: $T_t = T_0 \alpha^t$. [Corana et al. \(1987\)](#) suggest using $\alpha = 0.85$.
- **integer :: period = 100**
Number of iterations performed between temperature reductions. After `period` cycles of length `dwell` over `dim` elements (i.e., `period*dwell*dim` function evaluations), the temperature t is changed by the factor `alpha`. The value suggested by [Corana et al. \(1987\)](#) is $\max\{100, 5 \times \text{dim}\}$. See [Goffe et al. \(1994\)](#) for further advice.
- **integer :: dwell = 20**
Number of cycles. After `dwell * dim` function evaluations, each element of `step` is adjusted so that approximately half of all function evaluations are accepted. The suggested value is 20.
- **real(wp) :: step_adj = 2.0_wp**
Step length adjustment factor. The suggested value is 2.0.

- **integer** :: seed = 274
Seed for the internal random number generator.
- **real**(wp) :: eps = osl_eps_wp
Error tolerance used in the stopping criteria. If the function values from the previous num_eps temperatures differ from the known optimum by less than eps *and* the final function value at the current temperature differs from the current known optimum by less than eps, the algorithm terminates. See also num_eps.
- **integer** :: num_eps = 4
Number of final function values used to decide upon termination. Suggested value is 4. See also eps.
- **integer** :: max_eval = 100000
The maximum number of function evaluations. If it is exceeded, the algorithm terminates.
- **integer** :: verb = 0
Verbosity level:
 0. Nothing is printed.
 1. Prints the initial inputs, the status before each temperature reduction, and the final results. There are dim*dwell*period function evaluations between temperature reductions.
 2. Prints everything at level 1 and, each time the step length is changed (after dim*dwell function evaluations), the new step length step, the current optimal x, and the current trial x.
 3. Prints everything at level 2 and each function evaluation, whether it was accepted or rejected, and whether it was a new optima.

Example: Simulated Annealing

```

program siman
  use osl_core, wp => osl_wp
  use osl_siman
  use osl_func
  implicit none

  integer, parameter :: n = 2
  real(wp), parameter, dimension(n) :: x_start = (/ 2.354471_wp, -0.319186_wp /)
  real(wp), dimension(n) :: x_opt
  real(wp) :: f_opt
  real(wp), parameter :: temp = 5.0_wp
  type(osl_siman_t) :: sa

  call osl_print_header('Simulated Annealing Example')

  call osl_print('Searching for global minimum: F(0.864, 1.23) = 16.0817')
  call osl_print('Starting at local minimum: F(2.354, -0.319) = 20.9805')

  call osl_siman_init(sa, x_start, temp, &
    MAX=.false., VERB=0, &
    EPS=1.0e-6_wp, NUM_EPS=4, MAX_EVAL=100000, &

```

```

        ALPHA=0.5_wp, PERIOD=5, DWELL=20)

call osl_print_subheader('Unconstrained')
call osl_siman_opt(sa, osl_func_judge, X_OPT=x_opt, F_OPT=f_opt)
call osl_print('x_opt:', x_opt)
call osl_print('f_opt:', f_opt)

call osl_print_subheader('Fixed first component')
call osl_siman_set(sa, FIX=(/ .true., .false. /))
call osl_siman_opt(sa, osl_func_judge, X_OPT=x_opt, F_OPT=f_opt)
call osl_print('x_opt:', x_opt)
call osl_print('f_opt:', f_opt)

call osl_siman_free(sa)
end program siman

```

Output:

```

Simulated Annealing Example
=====
Searching for global minimum: F(0.864, 1.23) = 16.0817
Starting at local minimum: F(2.354, -0.319) = 20.9805

Unconstrained
-----
x_opt:
      0.86479      1.2357
f_opt:      -16.082

Fixed first component
-----
x_opt:
      2.3545      -0.98851
f_opt:      -20.898

```

Chapter 8

Numerical Differentiation

The `osl_diff` module contains routines for calculating finite-difference approximations to various derivatives of functions. For real-valued functions of many real variables (functions satisfying the `func_n` interface), `osl_gradient` calculates central-, forward-, and backward-difference approximations to the gradient. For real-vector-valued functions of many real variables (functions satisfying the `func_nm` interface), `osl_jacobian` calculates a central-difference approximation to the Jacobian. Finally, real-valued functions of many real variables (functions satisfying the `func_n` interface), `osl_hessian` calculates a central-difference approximation to the Hessian.

In each case, the stepsize h is chosen according to the rule of thumb of [Miranda and Fackler \(2002\)](#). For first derivatives, $h = \sqrt{\epsilon_f} \max\{|x|, 1\}$ for one-sided approximations and $h = \sqrt[3]{\epsilon_f} \max\{|x|, 1\}$ for two-sided approximations, where ϵ_f is the order of the error in the evaluation of the function. For second-derivatives, $h = \sqrt[4]{\epsilon_m} \max\{|x|, 1\}$. Furthermore, these routines prevent error introduced by the stepsize h by choosing h so that the difference between $x - h$ and $x + h$ is exactly representable for central-difference approximations (and similarly for forward- and backward-difference approximations).

8.1 Finite-Difference Gradient

The routine `osl_gradient` calculates either a central-, forward-, or backward-difference approximation to the gradient of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ represented by a subroutine `func_n`. The direction of the difference is specified through the optional `method` argument and can be one of the following constants: `osl_diff_central` (default), `osl_diff_forward`, or `osl_diff_backward`.

```
subroutine osl_gradient(func_n, x, grad, method, epsf, ctx)
  interface
    subroutine func_n(x, y, ctx)
      use osl_core, wp => osl_wp
      implicit none
      real(wp), dimension(:), intent(in) :: x
      real(wp), intent(out) :: y
      integer, dimension(:), intent(in), optional :: ctx
    end subroutine func_n
  end interface
  real(wp), dimension(:), intent(in) :: x
  real(wp), dimension(size(x)), intent(out) :: grad
  integer, intent(in), optional :: method
  real(wp), intent(in), optional :: epsf
  integer, dimension(:), intent(in), optional :: ctx
```

```
end subroutine osl_gradient
```

Example: Finite Difference Gradient

```
program gradient
  use osl_core, wp => osl_wp
  use osl_diff, only: osl_gradient
  use osl_func, only: osl_func_paraboloid, osl_func_paraboloid_nd
  implicit none

  integer, parameter :: n = 2
  real(wp), dimension(n) :: x, df, df_ans
  real(wp) :: f

  x = 2.0_wp
  call osl_func_paraboloid_nd(x, f, df_ans)
  call osl_gradient(osl_func_paraboloid, x, df)

  call osl_print('x:', x)
  call osl_print('Numeric gradient:', df)
  call osl_print('Analytic gradient:', df_ans)
end program gradient
```

Output:

```
x:
      2.0000      2.0000
Numeric gradient:
      16.000     -216.00
Analytic gradient:
      16.000     -216.00
```

8.2 Finite-Difference Jacobian

The subroutine `osl_jacobian` computes a central-difference approximation to the Jacobian of a real-valued function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ represented by a subroutine `func_nm`.

```
subroutine osl_jacobian(func_nm, x, jac, epsf, ctx)
  interface
    subroutine func_nm(x, y, ctx)
      use osl_core, wp => osl_wp
      implicit none
      real(wp), dimension(:), intent(in) :: x
      real(wp), dimension(:), intent(out) :: y
      integer, dimension(:), intent(in), optional :: ctx
    end subroutine func_nm
  end interface
  real(wp), dimension(size(jac,2)), intent(in) :: x
  real(wp), dimension(:, :), intent(out) :: jac
  real(wp), intent(in), optional :: epsf
  integer, dimension(:), intent(in), optional :: ctx
end subroutine osl_jacobian
```

Example: Finite Difference Jacobian

```

program jacobian
  use osl_core, wp => osl_wp
  use osl_diff, only: osl_jacobian
  use osl_func, only: osl_func_nm
  implicit none

  integer, parameter :: n = 2
  real(wp), dimension(n) :: x
  real(wp), dimension(n,n) :: J, J_ans

  x = 2.0_wp                                ! point of evaluation
  J_ans(1,:) = (/ 4.0, 0.0 /)                ! actual Jacobian
  J_ans(2,:) = (/ 0.0, 4.0 /)

  print *, 'f(x,y) = < x^2, y^2 >'
  print *, 'df(x,y)/dx = 2x, df(x,y)/dy = 2y'

  call osl_jacobian(osl_func_nm, x, J)

  call osl_print('x:', x)
  call osl_print('Numeric Jacobian at x:', J)
  call osl_print('Analytic Jacobian at x:', J_ans)
end program jacobian

```

Output:

```

f(x,y) = < x^2, y^2 >
df(x,y)/dx = 2x, df(x,y)/dy = 2y
x:
      2.0000      2.0000
Numeric Jacobian at x:
      4.0000      0.00000E+00
      0.00000E+00      4.0000
Analytic Jacobian at x:
      4.0000      0.00000E+00
      0.00000E+00      4.0000

```

8.3 Finite-Difference Hessian

The `osl_hessian` routine calculates a central-difference approximation to the Hessian of a real-valued function $f: \mathbb{R}^n \rightarrow \mathbb{R}$, represented by a subroutine `func_n`, using the algorithm described by (Miranda and Fackler, 2002, section 5.6).

```

subroutine osl_hessian(func_n, x, hess, epsf, ctx)
  interface
    subroutine func_n(x, y, ctx)
      use osl_core, wp => osl_wp
      implicit none
      real(wp), dimension(:), intent(in) :: x
      real(wp), intent(out) :: y
      integer, dimension(:), intent(in), optional :: ctx
    end subroutine func_n
  end interface
  real(wp), dimension(:), intent(in) :: x
  real(wp), dimension(size(x),size(x)), intent(out) :: hess

```

```

    real(wp), optional, intent(in) :: epsf
    integer, dimension(:), intent(in), optional :: ctx
end subroutine osl_hessian

```

Example: Finite Difference Hessian

```

program hessian
  use osl_core, wp => osl_wp
  use osl_diff, only: osl_hessian
  use osl_func, only: osl_func_paraboloid, osl_func_paraboloid_nd
  implicit none

  integer, parameter :: n = 2
  real(wp), dimension(n) :: x
  real(wp), dimension(n,n) :: H, H_ans

  x = (/ 2.0_wp, 2.0_wp /)
  H_ans(1,:) = (/ 20.0_wp, 0.0_wp /)
  H_ans(2,:) = (/ 0.0_wp, 40.0_wp /)

  call osl_hessian(osl_func_paraboloid, x, H)

  call osl_print('x:', x)
  call osl_print('Numeric Hessian:', H)
  call osl_print('Analytic Hessian:', H_ans)
end program hessian

```

Output:

```

x:
      2.0000      2.0000
Numeric Hessian:
      20.000      0.00000E+00
      0.00000E+00      40.000
Analytic Hessian:
      20.000      0.00000E+00
      0.00000E+00      40.000

```

Chapter 9

Special Functions

This chapter describes the implementation of various special functions.

9.1 Factorial and Related Functions

Factorial

The factorial of a non-negative integer n , denoted $n!$, is equal to the product of all positive integers less than or equal to n :

$$n! \equiv \prod_{k=1}^n k = 1 \cdot 2 \cdot \dots \cdot n.$$

Factorials arise frequently in combinatorics. For example, $n!$ is the number of distinct permutations of n elements. The factorial of zero, $0!$, is defined as 1.

In OSL, factorials are obtained via the `osl_factorial` function. Rather than calculating factorials by definition, this function is actually implemented as a table lookup because factorials for $n > 170$ are too large to be represented even as 32-bit floating point numbers. The `osl_factorial` function returns a `real(wp)` value because otherwise, if it returned an `integer`, it would be limited to $n \leq 12$, since $13! = 6,227,020,800$ is well out of range of a 32-bit `integer` (in general, but especially so in Fortran, where all `integers` are signed).

The tradeoff with returning floating point values is that factorials for $n > 18$ are not exact. The benefit is that factorials as large as $n = 170$ can be evaluated. An optional `err` argument is provided to approximate the error. If n is negative, the `status` flag returns a domain error `OSL_EDOM`, and if n is larger than 170, `status` will equal `OSL_EOVRFLW`, indicating an overflow error. Otherwise, `status` is set to `OSL_SUCCESS`.

```
function osl_factorial(n, err, status)
  integer, intent(in) :: n
  real(wp), intent(out), optional :: err
  integer, intent(out), optional :: status
  real(wp) :: osl_factorial
end function osl_factorial
```

Example: Factorial

```
program factorial
  use osl_core, wp => osl_wp
  use osl_special, only: osl_factorial
  implicit none
```

```

integer :: i

do i = 1, 12
  print 'i2, "!", i2)', i, int(osl_factorial(i))
end do
end program factorial

```

Output:

```

1!      1
2!      2
3!      6
4!     24
5!    120
6!    720
7!   5040
8!  40320
9!  362880
10! 3628800
11! 39916800
12! 479001600

```

Binomial Coefficients

The `osl_choose` function calculates the binomial coefficient $\binom{n}{k}$, defined as

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = \frac{n!}{k!(n-k)!} \quad \text{if } 0 \leq k \leq n$$

and

$$\binom{n}{k} = 0 \quad \text{if } k < 0 \text{ or } k > n.$$

```

integer function osl_choose(n, k)
  integer, intent(in) :: n
  integer, intent(in) :: k
end function osl_choose

```

Example: Pascal's Triangle

```

program choose
  use osl_core, only: wp => osl_wp
  use osl_special, only: osl_choose
  implicit none

  integer, parameter :: n_max = 7
  integer :: n, k

  do n = 0, n_max
    do k = 1, (n_max - n) / 2
      write(*, '(4x)', advance='no')
    end do
  end do

```

```
if (mod(n_max - n, 2) == 1) then
  write(*, '(2x)', advance='no')
end if
do k = 0, n
  write(*, '(i4)', advance='no') osl_choose(n,k)
end do
write(*, '(a)') ''
end do
end program choose
```

Output:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
```

Chapter 10

Permutations

In OSL, a permutation of size n is represented by an integer vector of dimension N containing the values $1, 2, \dots, n$ with each value appearing exactly once. There are $n!$ possible permutations of size n and thus an ordering, or ranking, of permutations is a bijective mapping from the set of permutations to the integers $1, 2, \dots, n!$.

A very natural ordering arises when permutations are arranged in lexicographic order. A permutation π is said to precede σ in lexicographic order if, for some i , $\pi_{1:i} = \sigma_{1:i}$ and $\pi_{i+1} < \sigma_{i+1}$. The permutation routines available in the GNU Scientific Library (Galassi, Davies, Theiler, Gough, Jungman, Alken, Booth, and Rossi, 2009) return permutations in lexicographic order.

Instead, OSL employs the Johnson-Trotter algorithm, which lists all permutations in an order such that each permutation differs from the preceding one only by a single transposition of adjacent elements. A proof of the validity of the algorithm is given by Stanton and White (1986).

Example: Permutations

```
program permutation
  use osl_core, wp => osl_wp
  use osl_permutation
  implicit none

  integer, parameter :: n = 3
  integer, dimension(n) :: pi
  integer :: status, rank

  call osl_permutation_identity(pi)

  do
    ! Print the current permutation and rank
    rank = osl_permutation_rank(pi)
    write(*, '(i2,":",4(1x,i1))') rank, pi

    ! Obtain the next (exiting on last)
    call osl_permutation_next(pi, status)
    if (status == OSL_FAIL) exit
  end do
end program permutation
```

Output:

```
1: 1 2 3
```

```
2: 1 3 2
3: 3 1 2
4: 3 2 1
5: 2 3 1
6: 2 1 3
```

10.1 Limitations

At present, the `osl_permutation` module is limited to permutations with $n \leq 170$. This bound arises because the number of possible permutations for $n > 170$ is larger than the range the of 32-bit integers used to enumerate them.

Chapter 11

Sorting

The `osl_sort` module contains routines for sorting arrays of numbers. It depends only on the `osl_core` module.

11.1 Quicksort

The subroutine `osl_quicksort` implements the Quicksort algorithm of [Hoare \(1961\)](#) which sorts an array of real numbers into ascending order. See also Section 8.2 of [Press, Teukolsky, Vetterling, and Flannery \(1992\)](#).

Example: Quicksort algorithm

```
program quicksort
  use osl_core, only: wp => osl_wp
  use osl_sort, only: osl_quicksort
  implicit none

  integer, parameter :: n_sort = 5
  real(wp), dimension(n_sort) :: v
  integer :: i

  do i = 1, n_sort
    call random_number(v(i))
  end do
  print '(A10, 10g9.2)', 'Unsorted:', v

  call osl_quicksort(v)
  print '(A10, 10g9.2)', 'Sorted:', v
end program quicksort
```

Output:

```
Unsorted:  1.0    0.57    0.97    0.75    0.37
Sorted:  0.37    0.57    0.75    0.97    1.0
```

11.2 Shell Sort

The subroutine `osl_shellsort` implements the sorting algorithm of ?. `osl_shellsort` accepts a vector `v` and sorts it in place. On output, the elements of `v` are placed in ascending order.

Example: Shell Sort

```
program shellsort
  use osl_core, only: wp => osl_wp
  use osl_sort, only: osl_shellsort
  implicit none

  integer, parameter :: n_sort = 5
  real(wp), dimension(n_sort) :: v
  integer :: i

  do i = 1, n_sort
    call random_number(v(i))
  end do
  print '(a10, 10g9.2)', 'Unsorted:', v

  call osl_shellsort(v)
  print '(a10, 10g9.2)', 'Sorted:', v
end program shellsort
```

Output:

```
Unsorted:  1.0    0.57    0.97    0.75    0.37
Sorted:  0.37    0.57    0.75    0.97    1.0
```

Appendix A

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

A.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

A.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software

Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

A.11 Relicensing

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

A.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

- Aruoba, S. B., J. Fernández-Villaverde, and J. F. Rubio-Ramírez (2006). Comparing solution methods for dynamic equilibrium economies. *Journal of Economic Dynamics and Control* 30, 2477–2508.
- Blevins, J. R. (2009). A generic linked list implementation in fortran 95. Working paper, Duke University.
- Corana, A., M. Marchesi, C. Martini, and S. Ridella (1987). Minimizing multimodal functions of continuous variables with the simulated annealing algorithm. *ACM Trans. Mathematical Software* 31, 262–280.
- Galassi, M., J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi (2009). *GNU Scientific Library: Reference Manual* (3 ed.).
- Goffe, W. L., G. D. Ferrier, and J. Rogers (1992). Simulated annealing: an initial application in econometrics. *Computational Economics* 5, 1572–9974.
- Goffe, W. L., G. D. Ferrier, and J. Rogers (1994). Global optimization of statistical functions with simulated annealing. *Journal of Econometrics* 60, 65–99.
- Hoare, C. A. R. (1961). Quicksort: Algorithm 64. *Communications of the ACM* 4, 321–322.
- Horowitz, J. L. (2001). The bootstrap. In J. J. Heckman and E. Leamer (Eds.), *Handbook of Econometrics*, Volume 5, Amsterdam. North Holland.
- Kronmal, R. A. and A. V. Peterson, Jr. (1979). On the alias method for generating random variables from a discrete distribution. *The American Statistician* 33, 214–218.
- Marsaglia, G. and A. Zaman (1994). Some portable very-long-period random number generators. *Computers in Physics* 8, 117–121.
- Miranda, M. J. and P. L. Fackler (2002). *Applied Computational Economics and Finance*. MIT Press.
- Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. New York: Cambridge University Press.
- Stanton, D. and D. White (1986). *Constructive Combinatorics*. Springer.
- Tauchen, G. (1986). Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters* 20(2), 177–81.
- Walker, A. J. (1977). An efficient method for generating discrete random variables with general distributions. *ACM Trans. Mathematical Software* 3, 253–256.