

VAMP, Version 1.0: Vegas AMPlified: Anisotropy, Multi-channel sampling and Parallelization

Thorsten Ohl¹²

Darmstadt University of Technology
Schloßgartenstr. 9
D-64289 Darmstadt
Germany

IKDA 98/??
hep-ph/yymmnnn
October 1999

DRAFT: April 21, 2010

¹e-mail: ohl@hep.tu-darmstadt.de

²Supported by Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie, Germany.

Abstract

We present an new implementation of the classic Vegas algorithm for adaptive multi-dimensional Monte Carlo integration in Fortran95. This implementation improves the performance for a large class of integrands, supporting stratified sampling in higher dimensions through automatic identification of the directions of largest variation. This implementation also supports multi channel sampling with individual adaptive grids. Sampling can be performed in parallel on workstation clusters and other parallel hardware.

Revision Control

prelude.nw 314 2010-04-17 20:32:33Z ohl
divisions.nw 314 2010-04-17 20:32:33Z ohl
vamp.nw 317 2010-04-18 00:31:03Z ohl
vampi.nw 314 2010-04-17 20:32:33Z ohl
vamp_test.nw 314 2010-04-17 20:32:33Z ohl
vamp_test0.nw 314 2010-04-17 20:32:33Z ohl
application.nw 314 2010-04-17 20:32:33Z ohl
kinds.nw 314 2010-04-17 20:32:33Z ohl
constants.nw 314 2010-04-17 20:32:33Z ohl
exceptions.nw 314 2010-04-17 20:32:33Z ohl
specfun.nw 314 2010-04-17 20:32:33Z ohl
vamp_stat.nw 314 2010-04-17 20:32:33Z ohl
histograms.nw 314 2010-04-17 20:32:33Z ohl
utils.nw 314 2010-04-17 20:32:33Z ohl
linalg.nw 314 2010-04-17 20:32:33Z ohl
products.nw 314 2010-04-17 20:32:33Z ohl
kinematics.nw 314 2010-04-17 20:32:33Z ohl
coordinates.nw 314 2010-04-17 20:32:33Z ohl
mpi90.nw 314 2010-04-17 20:32:33Z ohl
pmep.nw 314 2010-04-17 20:32:33Z ohl
postlude.nw 314 2010-04-17 20:32:33Z ohl

CONTENTS

1	INTRODUCTION	1
2	ALGORITHMS	3
2.1	<i>Importance Sampling</i>	4
2.2	<i>Stratified Sampling</i>	5
2.3	<i>Vegas</i>	6
2.3.1	<i>Vegas' Inflexibility</i>	6
2.3.2	<i>Vegas' Dark Side</i>	7
2.4	<i>Multi Channel Sampling</i>	7
2.5	<i>Revolving</i>	8
2.6	<i>Parallelization</i>	8
2.6.1	<i>Multilinear Structure of the Sampling Algorithm</i>	8
2.6.2	<i>State and Message Passing</i>	13
2.6.3	<i>Random Numbers</i>	13
2.6.4	<i>Practice</i>	14
3	DESIGN TRADE OFFS	19
3.1	<i>Programming Language</i>	20
4	USAGE	21
4.1	<i>Basic Usage</i>	21
4.1.1	<i>Basic Example</i>	22
4.2	<i>Advanced Usage</i>	24
4.2.1	<i>Types</i>	25
4.2.2	<i>Shared Arguments</i>	25
4.2.3	<i>Single Channel Procedures</i>	27
4.2.4	<i>Inout/Output and Marshling</i>	28
4.2.5	<i>Multi Channel Procedures</i>	31
4.2.6	<i>Event Generation</i>	34
4.2.7	<i>Parallelization</i>	34
4.2.8	<i>Diagnostics</i>	35

4.2.9	<i>Other Procedures</i>	36
4.2.10	<i>(Currently) Undocumented Procedures</i>	36
5	IMPLEMENTATION	37
5.1	<i>The Abstract Datatype division</i>	37
5.1.1	<i>Creation, Manipulation & Injection</i>	38
5.1.2	<i>Grid Refinement</i>	44
5.1.3	<i>Probability Density</i>	48
5.1.4	<i>Quadrupole</i>	49
5.1.5	<i>Forking and Joining</i>	49
5.1.6	<i>Inquiry</i>	56
5.1.7	<i>Diagnostics</i>	57
5.1.8	<i>I/O</i>	60
5.1.9	<i>Marshaling</i>	66
5.1.10	<i>Boring Copying and Deleting of Objects</i>	68
5.2	<i>The Abstract Datatype vamp_grid</i>	70
5.2.1	<i>Initialization</i>	76
5.2.2	<i>Sampling</i>	84
5.2.3	<i>Forking and Joining</i>	94
5.2.4	<i>Parallel Execution</i>	100
5.2.5	<i>Diagnostics</i>	103
5.2.6	<i>Multi Channel</i>	109
5.2.7	<i>Mapping</i>	123
5.2.8	<i>Event Generation</i>	132
5.2.9	<i>Convenience Routines</i>	137
5.2.10	<i>I/O</i>	140
5.2.11	<i>Marshaling</i>	154
5.2.12	<i>Boring Copying and Deleting of Objects</i>	160
5.3	<i>Interface to MPI</i>	163
5.3.1	<i>Parallel Execution</i>	164
5.3.2	<i>Event Generation</i>	178
5.3.3	<i>I/O</i>	180
5.3.4	<i>Communicating Grids</i>	183
6	SELF TEST	188
6.1	<i>No Mapping Mode</i>	188
6.1.1	<i>Serial Test</i>	188
6.1.2	<i>Parallel Test</i>	198
6.1.3	<i>Output</i>	200
6.2	<i>Mapped Mode</i>	200
6.2.1	<i>Serial Test</i>	200

6.2.2	<i>Parallel Test</i>	216
6.2.3	<i>Output</i>	217
7	APPLICATION	218
7.1	<i>Cross section</i>	218
A	CONSTANTS	241
A.1	<i>Kinds</i>	241
A.2	<i>Mathematical and Physical Constants</i>	241
B	ERRORS AND EXCEPTIONS	243
C	THE ART OF RANDOM NUMBERS	247
C.1	<i>Application Program Interface</i>	247
C.2	<i>Low Level Routines</i>	250
C.2.1	<i>Generation of 30-bit Random Numbers</i>	250
C.2.2	<i>Initialization of 30-bit Random Numbers</i>	251
C.2.3	<i>Generation of 52-bit Random Numbers</i>	255
C.2.4	<i>Initialization of 52-bit Random Numbers</i>	256
C.3	<i>The State</i>	258
C.3.1	<i>Creation</i>	259
C.3.2	<i>Destruction</i>	260
C.3.3	<i>Copying</i>	261
C.3.4	<i>Flushing</i>	262
C.3.5	<i>Input and Output</i>	262
C.3.6	<i>Marshaling and Unmarshaling</i>	267
C.4	<i>High Level Routines</i>	270
C.4.1	<i>Single Random Numbers</i>	272
C.4.2	<i>Arrays of Random Numbers</i>	273
C.4.3	<i>Procedures With Explicit <code>tao_random_state</code></i>	275
C.4.4	<i>Static Procedures</i>	276
C.4.5	<i>Generic Procedures</i>	277
C.4.6	<i>Luxury</i>	278
C.5	<i>Testing</i>	280
C.5.1	<i>30-bit</i>	280
C.5.2	<i>52-bit</i>	282
C.5.3	<i>Test Program</i>	283
D	SPECIAL FUNCTIONS	284
D.1	<i>Test</i>	286
E	STATISTICS	289

F	HISTOGRAMMING	292
G	MISCELLANEOUS UTILITIES	302
	<i>G.1 Memory Management</i>	302
	<i>G.2 Sorting</i>	305
	<i>G.3 Mathematics</i>	308
	<i>G.4 I/O</i>	310
H	LINEAR ALGEBRA	311
	<i>H.1 LU Decomposition</i>	311
	<i>H.2 Determinant</i>	314
	<i>H.3 Diagonalization</i>	314
	<i>H.4 Test</i>	319
I	PRODUCTS	321
J	KINEMATICS	322
	<i>J.1 Lorentz Transformations</i>	322
	<i>J.2 Massive Phase Space</i>	324
	<i>J.3 Massive 3-Particle Phase Space Revisited</i>	327
	<i>J.4 Massless n-Particle Phase Space: RAMBO</i>	330
	<i>J.5 Tests</i>	331
K	COORDINATES	334
	<i>K.1 Angular Spherical Coordinates</i>	334
	<i>K.2 Trigonometric Spherical Coordinates</i>	337
	<i>K.3 Surface of a Sphere</i>	340
L	IDIOMATIC FORTRAN90 INTERFACE FOR MPI	341
	<i>L.1 Basics</i>	341
	<i>L.2 Point to Point</i>	344
	<i>L.3 Collective Communication</i>	350
M	POOR MAN'S ELEMENTAL PROCEDURES	355
	<i>M.1 m4 Macros</i>	355
	<i>M.2 Miscellaneous Utilities</i>	356
	<i>M.3 Errors and Exceptions</i>	358
	<i>M.4 The Abstract Datatype division</i>	358
	<i>M.5 The Abstract Datatype vamp_grid</i>	366

N	IDEAS	371
N.1	<i>Toolbox for Interactive Optimization</i>	371
N.2	<i>Partially Non-Factorized Importance Sampling</i>	371
N.3	<i>Correlated Importance Sampling (?)</i>	371
N.4	<i>Align Coordinate System (i.e. the grid) with Singularities (or the hot region)</i>	372
N.5	<i>Automagic Multi Channel</i>	372
O	CROSS REFERENCES	373
O.1	<i>Identifiers</i>	373
O.2	<i>Refinements</i>	393

Program Summary:

- **Title of program:** VAMP, Version 1.0 (October 1999)
- **Program obtainable** by anonymous `ftp` from the host `crunch.ikp.physik.th-darmstadt.de` in the directory `pub/ohl/vamp`.
- **Licensing provisions:** Free software under the GNU General Public License.
- **Programming language used:** Fortran95 [8] (Fortran90 [7] and F [13] versions available as well)
- **Number of program lines in distributed program, including test data, etc.:** ≈ 4300 (excluding comments)
- **Computer/Operating System:** Any with a Fortran95 (or Fortran90 or F) programming environment.
- **Memory required to execute with typical data:** Negligible on the scale of typical applications calling the library.
- **Typical running time:** A small fraction (typically a few percent) of the running time of applications calling the library.
- **Purpose of program:**
- **Nature of physical problem:**
- **Method of solution:**
- **Keywords:** adaptive integration, event generation, parallel processing

—1—

INTRODUCTION

We present a reimplementation of the classic Vegas [1, 2] algorithm for adaptive multi-dimensional integration in Fortran95 [8, 12]¹. The purpose of this reimplementation is two-fold: for pedagogical reasons it is useful to employ Fortran95 features (in particular the array language) together with literate programming [4] for expressing the algorithm more concisely and more transparently. On the other hand we use a Fortran95 abstract type to separate the state from the functions. This allows multiple instances of Vegas with different adaptations to run in parallel and it paves the road for a more parallelizable implementation.

The variable names are more in line with [1] than with [2] or with [16, 17, 18], which is almost identical to [2].

Copyleft

Mention the GNU General Public License (maybe we can switch to the GNU Library General Public License)

```
1 <Copyleft notice 1>≡
  ! Copyright (C) 1998 by Thorsten Ohl <ohl@hep.tu-darmstadt.de>
  !
  ! VAMP is free software; you can redistribute it and/or modify it
  ! under the terms of the GNU General Public License as published by
  ! the Free Software Foundation; either version 2, or (at your option)
  ! any later version.
  !
  ! VAMP is distributed in the hope that it will be useful, but
  ! WITHOUT ANY WARRANTY; without even the implied warranty of
```

¹Fully functional versions conforming to preceding Fortran standard [7], High Performance Fortran (HPF) [9, 10, 14], and to the Fortran90 subset F [13] are available as well. A translation to the obsolete FORTRAN77 standard [6] is possible in principle, but extremely tedious and error prone if the full functionality shall be preserved.

! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
!!

Mention that the tangled sources are not the preferred form of distribution:

2 *<Copyleft notice 1>+≡*
! This version of the source code of vamp has no comments and
! can be hard to understand, modify, and improve. You should have
! received a copy of the literate 'noweb' sources of vamp that
! contain the documentation in full detail.
!!

—2— ALGORITHMS

 The notation has to be synchronized with [3]!

We establish some notation to allow a concise discussion. Notation:

$$\text{expectation: } E(f) = \frac{1}{|\mathcal{D}|} \int_{\mathcal{D}} dx f(x) \quad (2.1a)$$

$$\text{variance: } V(f) = E(f^2) - (E(f))^2 \quad (2.1b)$$

$$\text{estimate of expectation (average): } \langle X|f \rangle = \frac{1}{|X|} \sum_{x \in X} f(x) \quad (2.1c)$$

$$\text{estimate of variance: } \sigma_X^2(f) = \frac{1}{|X| - 1} (\langle X|f^2 \rangle - \langle X|f \rangle^2) \quad (2.1d)$$

Where $|X|$ is the size of the point set and $|\mathcal{D}| = \int_{\mathcal{D}} dx$ the size of the integration region. If $\mathcal{E}(\langle f \rangle)$ denotes the ensemble average of $\langle X|f \rangle$ over random point sets X with $|X| = N$, we have for expectation and variance

$$\mathcal{E}(\langle f \rangle) = E(f) \quad (2.2a)$$

$$\mathcal{E}(\sigma^2(f)) = V(f) \quad (2.2b)$$

and the ensemble variance of the expectation is also given by the variance

$$\mathcal{V}(\langle f \rangle) = \frac{1}{N} V(f) \quad (2.2c)$$

Therefore, it can be estimated from $\sigma_X^2(f)$. Below, we will also use the notation \mathcal{E}_g for the ensemble average over random point sets X_g with probability distribution g . We will write $E_g(f) = E(fg)$ as well.

2.1 Importance Sampling

If, instead of uniformly distributed points X , we use points X_g distributed according to a probability density g , we can easily keep the expectation constant

$$\mathcal{E}_g(\langle f \rangle) = E_g \left(\frac{f}{g} \right) = E(f) \quad (2.3)$$

while the variance transforms non-trivially

$$\mathcal{V}_g(\langle f \rangle) = \frac{1}{N} V_g \left(\frac{f}{g} \right) = \frac{1}{N} \left(E_g \left(\frac{f^2}{g^2} \right) - \left(E_g \left(\frac{f}{g} \right) \right)^2 \right) \quad (2.4)$$

and the error is minimized when f/g is constant, i.e. g is a good approximation of f . The non-trivial problem is to find a g that can be generated efficiently and is a good approximation at the same time.

One of the more popular approaches is to use a mapping ϕ of the integration domain

$$\begin{aligned} \phi : \mathcal{D} &\rightarrow \Delta \\ x &\mapsto \xi = \phi(x) \end{aligned} \quad (2.5)$$

In the new coordinates, the distribution is multiplied by the Jacobian of the inverse map ϕ^{-1} :

$$\int_{\mathcal{D}} dx f(\phi(x)) = \int_{\Delta} d\xi J_{\phi^{-1}}(\xi) f(\xi) \quad (2.6)$$

A familiar example is given by the map

$$\begin{aligned} \phi : [0, 1] &\rightarrow \mathbf{R} \\ x &\mapsto \xi = x^0 + a \cdot \tan \left(\left(x - \frac{1}{2} \right) \pi \right) \end{aligned} \quad (2.7)$$

with the inverse $\phi^{-1}(\xi) = \text{atan}((\xi - x_0)/a)/\pi + 1/2$ and the corresponding Jacobian reproducing a resonance

$$J_{\phi^{-1}}(\xi) = \frac{d\phi^{-1}(\xi)}{d\xi} = \frac{a}{\pi} \frac{1}{(\xi - x^0)^2 + a^2} \quad (2.8)$$

Obviously, this works only for a few special distributions. Fortunately, we can combine several of these mappings to build efficient integration algorithms, as will be explained in section 2.4 below. Another approach is to construct the approximation numerically, by appropriate binning of the integration domain (cf. [1, 2, 19]). The most popular technique for this will be discussed below in section 2.3.

2.2 Stratified Sampling

The technique of importance sampling concentrates the sampling points in the region where the contribution to the integrand is largest. Alternatively we can also concentrate the sampling points in the region where the contribution to the variance is largest.

If we divide the sampling region \mathcal{D} into n disjoint subregions \mathcal{D}^i

$$\mathcal{D} = \bigcup_{i=1}^n \mathcal{D}^i, \quad \mathcal{D}^i \cap \mathcal{D}^j = \emptyset \quad (i \neq j) \quad (2.9)$$

a new estimator is

 Bzzzt! Wrong. These multi-channel formulae are incorrect for partitionings and must be fixed.

$$\overline{\langle X|f \rangle} = \sum_{i=1}^n \frac{N_i}{N} \langle X_{\theta_i} | f \rangle \quad (2.10)$$

where

$$\theta_i(x) = \begin{cases} 1 & \text{for } x \in \mathcal{D}^i \\ 0 & \text{for } x \notin \mathcal{D}^i \end{cases} \quad (2.11)$$

and

$$\sum_{i=1}^n N_i = N \quad (2.12)$$

since the expectation is linear

$$\mathcal{E}(\overline{\langle f \rangle}) = \sum_{i=1}^n \frac{N_i}{N} \mathcal{E}_{\theta_i}(\langle f \rangle) = \sum_{i=1}^n \frac{N_i}{N} E_{\theta_i}(f) = \sum_{i=1}^n \frac{N_i}{N} E(f\theta_i) = E(f) \quad (2.13)$$

On the other hand, the variance of the estimator $\overline{\langle X|f \rangle}$ is

$$\mathcal{V}(\overline{\langle f \rangle}) = \sum_{i=1}^n \frac{N_i}{N} \mathcal{V}_{\theta_i}(\langle f \rangle) \quad (2.14)$$

This is minimized for

$$N_i \propto \sqrt{V(f \cdot \theta_{\mathcal{D}^i})} \quad (2.15)$$

as a simple variation of $\mathcal{V}(\overline{\langle f \rangle})$ shows.

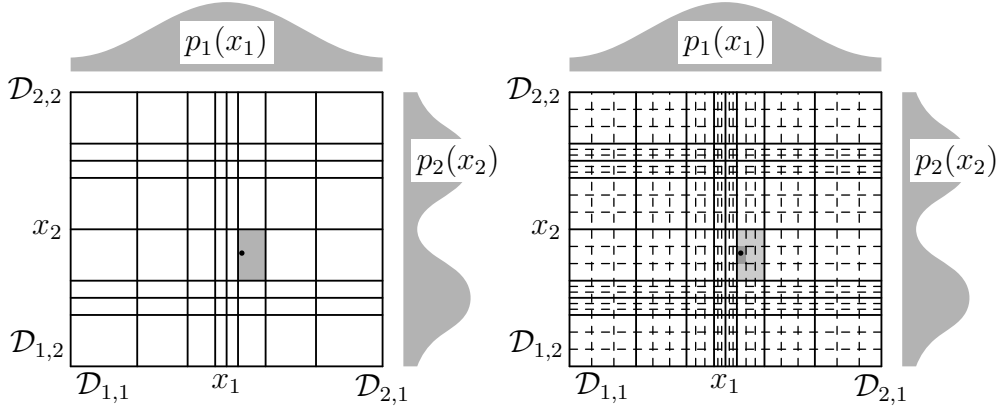


Figure 2.1: **vegas** grid structure for non-stratified sampling (left) and for genuinely stratified sampling (right), which is used in low dimensions. N.B.: the grid and the weight functions $p_{1,2}$ are only in qualitative agreement.



Figure 2.2: One-dimensional illustration of the **vegas** grid structure for pseudo stratified sampling, which is used in high dimensions.

2.3 Vegas

Under construction!

2.3.1 Vegas' Inflexibility

The classic implementation of the Vegas algorithm [1, 2] treats all dimensions alike. This constraint allows a very concise FORTRAN77-style coding of the algorithm, but there is no theoretical reason for having the same number of divisions in each direction. On the contrary, under these circumstances, even a dimension in which the integrand is rather smooth will contribute to the exponential blow-up of cells for stratified sampling. It is obviously beneficial to use a finer grid in those directions in which the fluctuations are stronger, while a coarser grid will suffice in the other directions.

One small step along this line is implemented in Version 5.0 of the package **BASES/SPRING** [19], where one set of “wild” variables is separated from “smooth” variables [20].

The present reimplementaion of the Vegas algorithm allows the application to choose the number of divisions in each direction freely. The routines that reshape the grid accept an integer array with the number of divisions as an optional argument `num_div`. It is easy to construct examples in which the careful use of this feature reduces the variance significantly.

Currently, no attempt is made for automatic optimization of the number of divisions. One reasonable approach is to monitor Vegas’ grid adjustments and to increase the number of division in those directions where Vegas’ keeps adjusting because of fluctuations. For each direction, a numerical measure of these fluctuations is given by the spread in the m_i . The total number of cells can be kept constant by reducing the number of divisions in the other directions appropriately. Thus

$$n_{\text{div},j} \rightarrow \frac{Q_j n_{\text{div},j}}{\left(\prod_j Q_j\right)^{1/n_{\text{dim}}}} \quad (2.16)$$

where we have used the damped standard deviation

$$Q_j = \left(\sqrt{\text{Var}(\{m\}_j)}\right)^\alpha \quad (2.17)$$

instead of the spread.

2.3.2 Vegas’ Dark Side



Under construction!

A partial solution of this problem will be presented in section 2.5.

2.4 Multi Channel Sampling

Even if Vegas performs well for a large class of integrands, many important applications do not lead to a factorizable distribution. The class of integrands that can be integrated efficiently by Vegas can be enlarged substantially by using multi channel methods. The new class will include almost all integrals appearing in high energy physics simulations.



The first version of this section is now obsolete. Consult [3] instead.

2.5 *Revolving*



Under construction!

2.6 *Parallelization*

Traditionally, parallel processing has not played a large rôle in simulations for high energy physics. A natural and trivial method of utilizing many processors will run many instances of the same (serial) program with different values of the input parameters in parallel. Typical matrix elements and phase space integrals offer few opportunities for small scale parallelization.

On the other hand, parameter fitting has become possible recently for observables involving a phase space integration. In this case, fast evaluation of the integral is essential and parallel execution becomes an interesting option.

A different approach to parallelizing Vegas has been presented recently [21].

2.6.1 *Multilinear Structure of the Sampling Algorithm*

In order to discuss the problems with parallelizing adaptive integration algorithms and to present solutions, it helps to introduce some mathematical notation. A sampling S is a map from the space π of point sets and the space F of functions to the real (or complex) numbers

$$\begin{aligned} S : \pi \times F &\rightarrow \mathbf{R} \\ (p, f) &\mapsto I = S(p, f) \end{aligned}$$

For our purposes, we have to be more specific about the nature of the point set. In general, the point set will be characterized by a sequence of pseudo random numbers $\rho \in R$ and by one or more grids $G \in \Gamma$ used for importance or stratified sampling. A simple sampling

$$\begin{aligned} S_0 : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\rightarrow R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (\rho', G, a', f, \mu'_1, \mu'_2) = S_0(\rho, G, a, f, \mu_1, \mu_2) \end{aligned} \tag{2.18}$$

estimates the n -th moments $\mu'_n \in \mathbf{R}$ of the function $f \in F$. The integral and its standard deviation can be derived easily from the moments

$$I = \mu_1 \tag{2.19a}$$

$$\sigma^2 = \frac{1}{N-1} (\mu_2 - \mu_1^2) \tag{2.19b}$$

while the latter are more convenient for the following discussion. In addition, S_0 collects auxiliary information to be used in the grid refinement, denoted by $a \in A$. The unchanged arguments G and f have been added to the result of S_0 in (2.18), so that S_0 has identical domain and codomain and can therefore be iterated. Previous estimates μ_n may be used in the estimation of μ'_n , but a particular S_0 is free to ignore them as well. Using a little notational freedom, we augment \mathbf{R} and A with a special value \cdot , which will always be discarded by S_0 .

In an adaptive integration algorithm, there is also a refinement operation $r : \Gamma \times A \rightarrow \Gamma$ that can be extended naturally to the codomain of S_0

$$\begin{aligned} r : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\rightarrow R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (\rho, G', a, f, \mu_1, \mu_2) = r(\rho, G, a, f, \mu_1, \mu_2) \end{aligned} \quad (2.20)$$

so that $S = rS_0$ is well defined and we can specify n -step adaptive sampling as

$$S_n = S_0(rS_0)^n \quad (2.21)$$

Since, in a typical application, only the estimate of the integral and the standard deviation are used, a projection can be applied to the result of S_n :

$$\begin{aligned} P : R \times \Gamma \times A \times F \times \mathbf{R} \times \mathbf{R} &\rightarrow \mathbf{R} \times \mathbf{R} \\ (\rho, G, a, f, \mu_1, \mu_2) &\mapsto (I, \sigma) \end{aligned} \quad (2.22)$$

Then

$$(I, \sigma) = PS_0(rS_0)^n(\rho, G_0, \cdot, f, \cdot, \cdot) \quad (2.23)$$

and a good refinement prescription r , such as Vegas, will minimize the σ .

For parallelization, it is crucial to find a division of S_n or any part of it into *independent* pieces that can be evaluated in parallel. In order to be effective, r has to be applied to *all* of a and therefore a synchronization of G before and after r is appropriately. Furthermore, r usually uses only a tiny fraction of the CPU time and it makes little sense to invest a lot of effort into parallelizing it beyond what the Fortran compiler can infer from array notation. On the other hand, S_0 can be parallelized naturally, because all operations are linear, including the computation of a . We only have to make sure that the cost of communicating the results of S_0 and r back and forth during the computation of S_n do not offset any performance gain from parallel processing.

When we construct a decomposition of S_0 and prove that it does not change the results, i.e.

$$S_0 = \iota S_0 \phi \quad (2.24)$$

where ϕ is a forking operation and ι is a joining operation, we are faced with the technical problem of a parallel random number source ρ . As made explicit in (2.18), S_0 changes the state of the random number general ρ , demanding *identical* results therefore imposes a strict ordering on the operations and defeats parallelization. It is possible to devise implementations of S_0 and ρ that circumvent this problem by distributing subsequences of ρ in such a way among processes that results do not depend on the number of parallel processes.

However, a reordering of the random number sequence will only change the result by the statistical error, as long as the scale of the allowed reorderings is *bounded* and much smaller than the period of the random number generator¹ Below, we will therefore use the notation $x \approx y$ for “equal for an appropriate finite reordering of the ρ used in calculating x and y ”. For our purposes, the relation $x \approx y$ is strong enough and allows simple and efficient implementations.

Since S_0 is essentially a summation, it is natural to expect a linear structure

$$\bigoplus_i S_0(\rho_i, G_i, a_i, f, \mu_{1,i}, \mu_{2,i}) \approx S_0(\rho, G, a, f, \mu_1, \mu_2) \quad (2.25a)$$

where

$$\rho = \bigoplus_i \rho_i \quad (2.25b)$$

$$G = \bigoplus_i G_i \quad (2.25c)$$

$$a = \bigoplus_i a_i \quad (2.25d)$$

$$\mu_n = \bigoplus_i \mu_{n,i} \quad (2.25e)$$

for appropriate definitions of “ \oplus ”. For the moments, we have standard addition

$$\mu_{n,1} \oplus \mu_{n,2} = \mu_{n,1} + \mu_{n,2} \quad (2.26)$$

and since we only demand equality up to reordering, we only need that the ρ_i are statistically independent. This leaves us with G and a and we have to discuss importance sampling and stratified sampling separately.

¹Arbitrary reorderings on the scale of the period of the random number generators could select constant sequences and have to be forbidden.

Importance Sampling

In the case of naive Monte Carlo and importance sampling the natural decomposition of G is to take j copies of the same grid G/j which is identical to G , each with one j -th of the total sampling points. As long as the a are linear themselves, we can add them up just like the moments

$$a_1 \oplus a_2 = a_1 + a_2 \quad (2.27)$$

and we have found a decomposition (2.25). In the case of Vegas, the a_i are sums of function values at the sampling points. Thus they are obviously linear and this approach is applicable to Vegas in the importance sampling mode.

Stratified Sampling

The situation is more complicated in the case of stratified sampling. The first complication is that in pure stratified sampling there are only two sampling points per cell. Splitting the grid in two pieces as above provide only a very limited amount of parallelization. The second complication is that the a are no longer linear, since they correspond to a sampling of the variance per cell and no longer of function values themselves.

However, as long as the samplings contribute to disjoint bins only, we can still “add” the variances by combining bins. The solution is therefore to divide the grid into disjoint bins along the divisions of the stratification grid and to assign a set of bins to each processor.

Finer decompositions will incur higher communications costs and other resource utilization. An implementation based on PVM is described in [21], which minimizes the overhead by running identical copies of the grid G on each processor. Since most of the time is usually spent in function evaluations, it makes sense to run a full S_0 on each processor, skipping function evaluations everywhere but in the region assigned to the processor. This is a neat trick, which is unfortunately tied to the computational model of message passing systems such as PVM and MPI [11]. More general paradigms can not be supported since the separation of the state for the processors is not explicit (it is implicit in the separated address space of the PVM or MPI processes).

However, it is possible to implement (2.25) directly in an efficient manner. This is based on the observation that the grid G used by Vegas is factorized into divisions D^j for each dimension

$$G = \bigotimes_{j=1}^{n_{\text{dim}}} D^j \quad (2.28)$$

and decompositions of the D^j induce decompositions of G

$$\begin{aligned} G_1 \oplus G_2 &= \left(\bigotimes_{j=1}^{i-1} D^j \otimes D_1^i \otimes \bigotimes_{i=j+1}^{n_{\text{dim}}} D^j \right) \oplus \left(\bigotimes_{j=1}^{i-1} D^j \otimes D_2^i \otimes \bigotimes_{i=j+1}^{n_{\text{dim}}} D^j \right) \\ &= \bigotimes_{j=1}^{i-1} D^j \otimes (D_1^i \oplus D_2^i) \otimes \bigotimes_{j=i+1}^{n_{\text{dim}}} D^j \quad (2.29) \end{aligned}$$

We can translate (2.29) directly to code that performs the decomposition $D^i = D_1^i \oplus D_2^i$ discussed below and simply duplicates the other divisions $D^{j \neq i}$. A decomposition along multiple dimensions is implemented by a recursive application of (2.29).

In Vegas, the auxiliary information a inherits a factorization similar to the grid (2.28)

$$a = (d^1, \dots, d^{n_{\text{dim}}}) \quad (2.30)$$

but not a multilinear structure. Instead, *as long as the decomposition respects the stratification grid*, we find the in place of (2.29)

$$a_1 \oplus a_2 = (d_1^1 + d_2^1, \dots, d_1^i \oplus d_2^i, \dots, d_1^{n_{\text{dim}}} + d_2^{n_{\text{dim}}}) \quad (2.31)$$

with “+” denoting the standard addition of the bin contents and “ \oplus ” denoting the aggregation of disjoint bins. If the decomposition of the division would break up cells of the stratification grid (2.31) would be incorrect, because, as discussed above, the variance is not linear.

Now it remains to find a decomposition

$$D^i = D_1^i \oplus D_2^i \quad (2.32)$$

for both the pure stratification mode and the pseudo stratification mode of vegas (cf. figure 2.1). In the pure stratification mode, the stratification grid is strictly finer than the adaptive grid and we can decompose along either of them immediately. Technically, a decomposition along the coarser of the two is straightforward. Since the adaptive grid already has more than 25 bins, a decomposition along the stratification grid makes no practical sense and the decomposition along the adaptive grid has been implemented. The sampling algorithm S_0 can be applied *unchanged* to the individual grids resulting from the decomposition.

For pseudo stratified sampling (cf. figure 2.2), the situation is more complicated, because the adaptive and the stratification grid do not share bin boundaries. Since Vegas does *not* use the variance in this mode, it would be theoretically possible to decompose along the adaptive grid and to mimic the



Figure 2.3: Forking one dimension d of a grid into three parts $ds(1)$, $ds(2)$, and $ds(3)$. The picture illustrates the most complex case of pseudo stratified sampling (cf. fig. 2.2).

incomplete bins of the stratification grid in the sampling algorithm. However, this would be a technical complication, destroying the universality of S_0 . Therefore, the adaptive grid is subdivided in a first step in

$$\text{lcm} \left(\frac{\text{lcm}(n_f, n_g)}{n_f}, n_x \right) \quad (2.33)$$

bins,² such that the adaptive grid is strictly finer than the stratification grid. This procedure is shown in figure 2.3.

2.6.2 State and Message Passing

2.6.3 Random Numbers

In the parallel example sitting on top of MPI [11] takes advantage of the ability of Knuth's generator [15] to generate statistically independent subse-

²The coarsest grid covering the division of n_g bins into n_f forks has $n_g / \text{gcd}(n_f, n_g) = \text{lcm}(n_f, n_g) / n_f$ bins per fork.

quences. However, since the state of the random number generator is explicit in all procedure calls, other means of obtaining subsequences can be implemented in a trivial wrapper.

The results of the parallel example will depend on the number of processors, because this effects the subsequences being used. Of course, the variation will be compatible with the statistical error. It must be stressed that the results are deterministic for a given number of processors and a given set of random number generator seeds. Since parallel computing environments allow to fix the number of processors, debugging of exceptional conditions is possible.

2.6.4 Practice

In this section we show three implementations of S_n : one serial, and two parallel, based on HPF [9, 10, 14] and MPI [11], respectively. From these examples, it should be obvious how to adapt VAMP to other parallel computing paradigms.

Serial

Here is a bare bones serial version of S_n , for comparison with the parallel versions below. The real implementation of `vamp_sample_grid` in the module `vamp` includes some error handling, diagnostics and the projection P (cf. (2.22)):

```

14  $\langle$ Serial implementation of  $S_n = S_0(rS_0)^n$   $\rangle \equiv$ 
  subroutine vamp_sample_grid (rng, g, iterations, func)
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grid), intent(inout) :: g
    integer, intent(in) :: iterations
     $\langle$ Interface declaration for func 22 $\rangle$ 
    integer :: iteration
    iterate: do iteration = 1, iterations
      call vamp_sample_grid0 (rng, g, func)
      call vamp_refine_grid (g)
    end do iterate
  end subroutine vamp_sample_grid

```

HPF

The HPF version of S_n is based on decomposing the grid `g` as described in section 2.6.1 and lining up the components in an array `gs`. The elements of `gs` can then be processed in parallel. This version can be compiled with any

Fortran compiler and a more complete version of this procedure (including error handling, diagnostics and the projection P) is included with VAMP as `vamp_sample_grid_parallel` in the module `vamp`. This way, the algorithm can be tested on a serial machine, but there will obviously be no performance gain.

Instead of one random number generator state `rng`, it takes an array consisting of one state per processor. These `rng(:)` are assumed to be initialized, such that the resulting sequences are statistically independent. For this purpose, Knuth's random number generator [15] is most convenient and is included with VAMP (see the example on page 16). Before each S_0 , the procedure `vamp_distribute_work` determines a good decomposition of the grid `d` into `size(rng)` pieces. This decomposition is encoded in the array `d` where `d(1,:)` holds the dimensions along which to split the grid and `d(2,:)` holds the corresponding number of divisions. Using this information, the grid is decomposed by `vamp_fork_grid`. The HPF compiler will then distribute the `!hpf$ independent` loop among the processors. Finally, `vamp_join_grid` gathers the results.

15 \langle Parallel implementation of $S_n = S_0(rS_0)^n$ (HPF) 15 $\rangle \equiv$

```

subroutine vamp_sample_grid_hpf (rng, g, iterations, func)
  type(tao_random_state), dimension(:), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: iterations
  <Interface declaration for func 22>
  type(vamp_grid), dimension(:), allocatable :: gs, gx
  !hpf$ processors p(number_of_processors())
  !hpf$ distribute gs(cyclic(1)) onto p
  integer, dimension(:,:), pointer :: d
  integer :: iteration, num_workers
  iterate: do iteration = 1, iterations
    call vamp_distribute_work (size (rng), vamp_rigid_divisions (g), d)
    num_workers = max (1, product (d(2,:)))
    if (num_workers > 1) then
      allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
      call vamp_create_empty_grid (gs)
      call vamp_fork_grid (g, gs, gx, d)
      !hpf$ independent
      do i = 1, num_workers
        call vamp_sample_grid0 (rng(i), gs(i), func)
      end do
      call vamp_join_grid (g, gs, gx, d)
      call vamp_delete_grid (gs)
      deallocate (gs, gx)
    end if
  end do
end subroutine vamp_sample_grid_hpf
```



```

        else
            call vamp_sample_grid0 (rng(1), g, func)
        end if
        call vamp_refine_grid (g)
    end do iterate
end subroutine vamp_sample_grid_hpf

```

Since `vamp_sample_grid0` performs the bulk of the computation, an almost linear speedup with the number of processors can be achieved, if `vamp_distribute_work` finds a good decomposition of the grid. The version of `vamp_distribute_work` distributed with VAMP does a good job in most cases, but will not be able to use all processors if their number is a prime number larger than the number of divisions in the stratification grid. Therefore it can be beneficial to tune `vamp_distribute_work` to specific hardware. Furthermore, using a finer stratification grid can improve performance.

For definiteness, here is an example of how to set up the array of random number generators for HPF. Note that this simple seeding procedure only guarantees statistically independent sequences with Knuth's random number generator [15] and will fail with other approaches.

```

16 <Parallel usage of  $S_n = S_0(rS_0)^n$  (HPF) 16>≡
    type(tao_random_state), dimension(:), allocatable :: rngs
    !hpf$ processors p(number_of_processors())
    !hpf$ distribute gs(cyclic(1)) onto p
    integer :: i, seed
    ! ...
    allocate (rngs(number_of_processors()))
    seed = 42 ! can be read from a file, of course ...
    !hpf$ independent
    do i = 1, size (rngs)
        call tao_random_create (rngs(i), seed + i)
    end do
    ! ...
    call vamp_sample_grid_hpf (rngs, g, 6, func)
    ! ...

```

MPI

The MPI version is more low level, because we have to keep track of message passing ourselves. Note that we have made this synchronization points explicit with three `if ... then ... else ... end if` blocks: forking, sampling, and joining. These blocks could be merged (without any performance gain) at the expense of readability. We assume that `rng` has been initialized

in each process such that the sequences are again statistically independent.

17 *Parallel implementation of $S_n = S_0(rS_0)^n$ (MPI) 17* \equiv

```

subroutine vamp_sample_grid_mpi (rng, g, iterations, func)
  type(tao_random_state), dimension(:), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: iterations
  Interface declaration for func 22
  type(vamp_grid), dimension(:), allocatable :: gs, gx
  integer, dimension(:,:), pointer :: d
  integer :: num_proc, proc_id, iteration, num_workers
  call mpi90_size (num_proc)
  call mpi90_rank (proc_id)
  iterate: do iteration = 1, iterations
    if (proc_id == 0) then
      call vamp_distribute_work (num_proc, vamp_rigid_divisions (g), d)
      num_workers = max (1, product (d(2,:)))
    end if
    call mpi90_broadcast (num_workers, 0)
    if (proc_id == 0) then
      allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
      call vamp_create_empty_grid (gs)
      call vamp_fork_grid (g, gs, gx, d)
      do i = 2, num_workers
        call vamp_send_grid (gs(i), i-1, 0)
      end do
    else if (proc_id < num_workers) then
      call vamp_receive_grid (g, 0, 0)
    end if
    if (proc_id == 0) then
      if (num_workers > 1) then
        call vamp_sample_grid0 (rng, gs(1), func)
      else
        call vamp_sample_grid0 (rng, g, func)
      end if
    else if (proc_id < num_workers) then
      call vamp_sample_grid0 (rng, g, func)
    end if
    if (proc_id == 0) then
      do i = 2, num_workers
        call vamp_receive_grid (gs(i), i-1, 0)
      end do
      call vamp_join_grid (g, gs, gx, d)
      call vamp_delete_grid (gs)
    end if
  end do
end subroutine vamp_sample_grid_mpi

```

```

        deallocate (gs, gx)
        call vamp_refine_grid (g)
    else if (proc_id < num_workers) then
        call vamp_send_grid (g, 0, 0)
    end if
end do iterate
end subroutine vamp_sample_grid_mpi

```

A more complete version of this procedure is included with VAMP as well, this time as `vamp_sample_grid` in the MPI support module `vampi`.

—3—

DESIGN TRADE OFFS

There have been three competing design goals for vegas, that are not fully compatible and had to be reconciled with compromises:

- *Ease-Of-Use*: few procedures, few arguments.
- *Parallelizability*: statelessness
- *Performance and Flexibility*: rich interface, functionality.

In fact, parallelizability and ease-of-use are complementary. A parallelizable implementation has to expose *all* the internal state. In our case, this includes the state of the random number generator and the adaptive grid. A simple interface would hide such details from the user.

The modern language features introduced to Fortran in 1990 [7] allows to reconcile these competing goals. Two abstract data types `vamp_state` and `tao_random_state` hide the details of the implementation from the user and encapsulate the two states in just two variables.

Another problem with parallelizability arised from the lack of a general exception mechanism in Fortran. The Fortran90 standard [8] forbids *any* input/output (even to the terminal) as well as `stop` statements in parallelizable (`pure`) procedures. This precludes simple approaches to monitoring and error handling. In Vegas we use a simple hand crafted exception mechanism (see chapter B) for communicating error conditions to the out layers of the applications. Unfortunately this requires the explicit passing of state in argument lists.

An unfortunate consequence of the similar approach to monitoring is that monitoring is *not* possible during execution. Instead, intermediate results can only be examined after a parallelized section of code has completed.

3.1 *Programming Language*

We have chosen to implement VAMP in Fortran90/95, which some might consider a questionable choice today. Nevertheless, we are convinced that Fortran90/95 (with all its weaknesses) is, by a wide margin, the right tool for the job.

Let us consider the alternatives

- FORTRAN77 is still the dominant language in high energy physics and all running experiment's software environments are based on it. However, the standard [6] is obsolete now and the successors [7, 8] have added many desirable features, while retaining almost all of FORTRAN77 as a subset.
- C/C++ appears to be the most popular programming language in industry and among young high energy physicists. Large experiments have taken a bold move and are basing their software environment on C++.
- Typed higher order functional programming languages (ML, Haskell, etc.) are a very promising development. Unfortunately, there is not yet enough industry support for high performance optimizing compilers. While the performance penalty of these languages is not as high as commonly believed (research compilers, which do not perform extensive processor specific optimizations, result in code that runs by a factor of two or three slower than equivalent Fortran code), it is relevant for long running, computing intensive applications. In addition, these languages are syntactically and idiomatically very different from Fortran and C. Another implementation of VAMP in ML will be undertaken for research purposes to investigate new algorithms that can only be expressed awkwardly in Fortran, but we do not expect it to gain immediate popularity.

—4— USAGE

4.1 Basic Usage

`type(vamp_grid)`

`subroutine vamp_create_grid (g, domain [, num_calls] [, exc])`

Create a fresh grid for the integration domain

$$\mathcal{D} = [D_{1,1}, D_{2,1}] \times [D_{1,2}, D_{2,2}] \times \dots \times [D_{1,n}, D_{2,n}] \quad (4.1)$$

dropping all accumulated results. This function *must not* be called twice on the first argument, without an intervening

`vamp_delete_grid`. Iff the variable `num_calls` is given, it will be the number of sampling points per iteration for the call to `vamp_sample_grid`.

`subroutine vamp_delete_grid (g [, exc])`

`subroutine vamp_discard_integral (g [, num_calls] [, exc])`

Keep the current optimized grid, but drop the accumulated results for the integral (value and errors). Iff the variable `num_calls` is given, it will be the new number of sampling points per iteration for the calls to `vamp_sample_grid`.

`subroutine vamp_reshape_grid (g [, num_calls] [, exc])`

Keep the current optimized grid and the accumulated results for the integral (value and errors). The variable `num_calls` is the new number of sampling points per iteration for the calls to `vamp_sample_grid`.

`subroutine vamp_sample_grid (rng, g, func, iterations
[, integral] [, std_dev] [, avg_chi2] [, exc] [, history])`

Sample the function `func` using the grid `g` for `iterations` iterations and optimize the grid after each iteration. The results are returned in `integral`, `std_dev` and `avg_chi2`. The random number generator uses and updates the state stored in `rng`. The explicit random number state is inconvenient, but required for parallelizability.

```
subroutine vamp_integrate (rng, g, func, calls [, integral]
  [, std_dev] [, avg_chi2] [, exc] [, history])
```

This is a wrapper around the above routines, that is steered by a `integer`, `dimension(2,:)` array `calls`. For each `i`, there will be `calls(1,i)` iterations with `calls(2,i)` sampling points.

```
subroutine vamp_integrate (rng, domain, func, calls
  [, integral] [, std_dev] [, avg_chi2] [, exc] [, history])
```

A second specific form of `vamp_integrate`. This one keeps a private grid and provides the shortest—and most inflexible—calling sequence.

22 *⟨Interface declaration for func 22⟩*≡

```
interface
  pure function func (xi, prc_index, weights, channel, grids) result (f)
    use kinds
    use vamp_grid_type !NODEP!
    real(kind=default), dimension(:), intent(in) :: xi
    integer, intent(in) :: prc_index
    real(kind=default), dimension(:), intent(in), optional :: weights
    integer, intent(in), optional :: channel
    type(vamp_grid), dimension(:), intent(in), optional :: grids
    real(kind=default) :: f
  end function func
end interface
```

4.1.1 Basic Example

In Fortran95, the function to be sampled *must* be **pure**, i.e. have no side effects to allow parallelization. The optional arguments `weights` and `channel` *must* be declared to allow the compiler to verify the interface, but they are ignored during basic use. Their use for multi channel sampling will be explained below. Here's a Gaussian

$$f(x) = e^{-\frac{1}{2} \sum_i x_i^2} \quad (4.2)$$

```

23a <basic.f90 23a>≡
  module basic_fct
    use kinds
    implicit none
    private
    public :: fct
  contains
    function fct (x, weights, channel) result (f_x)
      real(kind=default), dimension(:), intent(in) :: x
      real(kind=default), dimension(:), intent(in), optional :: weights
      integer, intent(in), optional :: channel
      real(kind=default) :: f_x
      f_x = exp (-0.5 * sum (x*x))
    end function fct
  end module basic_fct

```

In the main program, we need to import five modules. The customary module `kinds` defines `double` as the kind for double precision floating point numbers. The model `exceptions` provides simple error handling support (parallelizable routines are not allowed to issue error messages themselves, but must pass them along). The module `tao_random_numbers` hosts the random number generator used and `vamp` is the adaptive iteration module proper. Finally, the application module `basic_fct` has to be imported as well.

```

23b <basic.f90 23a>+≡
  program basic
    use kinds
    use exceptions
    use tao_random_numbers
    use vamp
    use basic_fct
    implicit none

```

Then we define four variables for an error message, the random number generator state and the adaptive integration grid. We also declare a variable for holding the integration domain and variables for returning the result. In this case we integrate the 7-dimensional hypercube.

```

23c <basic.f90 23a>+≡
  type(exception) :: exc
  type(tao_random_state) :: rng
  type(vamp_grid) :: grid
  real(kind=default), dimension(2,7) :: domain
  real(kind=default) :: integral, error, chi2
  domain(1,:) = -1.0
  domain(2,:) = 1.0

```


Initialize and seed the random number generator. Initialize the grid for 10 000 sampling points.

```
24a <basic.f90 23a>+≡
    call tao_random_create (rng, seed=0)
    call clear_exception (exc)
    call vamp_create_grid (grid, domain, num_calls=10000, exc=exc)
    call handle_exception (exc)
```

Warm up the grid in six low statistics iterations. Clear the error status before and check it after the sampling.

```
24b <basic.f90 23a>+≡
    call clear_exception (exc)
    call vamp_sample_grid (rng, grid, fct, 6, exc=exc)
    call handle_exception (exc)
```

Throw away the intermediate results and reshape the grid for 100 000 sampling points—keeping the adapted grid—and do four iterations of a higher statistics integration

```
24c <basic.f90 23a>+≡
    call clear_exception (exc)
    call vamp_discard_integral (grid, num_calls=100000, exc=exc)
    call handle_exception (exc)
    call clear_exception (exc)
    call vamp_sample_grid (rng, grid, fct, 4, integral, error, chi2, exc=exc)
    call handle_exception (exc)
    print *, "integral = ", integral, "+/-", error, " (chi^2 = ", chi2, ")"
end program basic
```

Since this is the most common use, there is a convenience routine available and the following code snippet is equivalent:

```
24d <Alternative to basic.f90 24d>≡
    integer, dimension(2,2) :: calls
    calls(:,1) = (/ 6, 10000 /)
    calls(:,2) = (/ 4, 100000 /)
    call clear_exception (exc)
    call vamp_integrate (rng, domain, fct, calls, integral, error, chi2, exc=exc)
    call handle_exception (exc)
```

4.2 Advanced Usage



Caveat emptor: no magic of literate programming can guarantee that the following remains in sync with the implementation. This has to be maintained manually.

All `real` variables are declared as `real(kind=default)` in the source and the variable `double` is imported from the module `kinds` (see appendix A.1). The representation of real numbers can therefore be changed by changing `double` in `kinds`.

4.2.1 *Types*

```
type(vamp_grid)
type(vamp_grids)
type(vamp_history)
type(exception)
  (from module exceptions)
```

4.2.2 *Shared Arguments*

Arguments keep their name across procedures, in order to make the Fortran90 keyword interface consistent.

```
real, intent(in) :: accuracy
```

Terminate S_n after $n' < n$ iterations, if relative error is smaller than `accuracy`. Specifically, the termination condition is

$$\frac{\text{std_dev}}{\text{integral}} < \text{accuracy} \quad (4.3)$$

```
real, intent(out) :: avg_chi2
```

The average χ^2 of the iterations.

```
integer, intent(in) :: channel
```

Call `func` with this optional argument. Multi channel sampling uses this to emulate arrays of functions

```
logical, intent(in) :: covariance
```

Collect covariance data.

```
type(exception), intent(inout) :: exc
```

Exceptional conditions are reported in `exc`.

```
type(vamp_grid), intent(inout) :: g
```

Unless otherwise noted, `g` denotes the active sampling grid in the documentation below.

```
type(vamp_histories), dimension(:), intent(inout) ::  
  histories
```

Diagnostic information for multi channel sampling.

```
type(vamp_history), dimension(:), intent(inout) ::  
  history
```

Diagnostic information for single channel sampling or summary of multi channel sampling.

```
real, intent(out) :: integral
```

The current best estimate of the integral.

```
integer, intent(in) :: iterations
```

```
real, dimension(:,:), intent(in) :: map
```

```
integer, intent(in) :: num_calls
```

The number of sampling points.

```
integer, dimension(:), intent(in) :: num_div
```

Number of divisions of the adaptive grid in each dimension.

```
logical, intent(in) :: quadrupole
```

Allow “quadrupole oscillations” of the sampling grid (cf. section [2.3.1](#)).

```
type(tao_random_state), intent(inout) :: rng
```

Unless otherwise noted, `rng` denotes the source of random numbers used for sampling in the documentation below.

```
real, intent(out) :: std_dev
```

The current best estimate of the error on the integral.

```
logical, intent(in) :: stratified
```

Try to use stratified sampling.

```
real(kind=default), dimension(:), intent(in) :: weights
```

```
...
```

4.2.3 *Single Channel Procedures*

```
subroutine vamp_create_grid (g, domain, num_calls
    [, quadrupole] [, stratified] [, covariance] [, map] [, exc])

    real, dimension(:,:), intent(in) :: domain

subroutine vamp_create_empty_grid (g)

subroutine vamp_discard_integral (g [, num_calls]
    [, stratified] [, quadrupole] [, covariance] [, exc])

subroutine vamp_reshape_grid (g [, num_calls] [, num_div]
    [, stratified] [, quadrupole] [, covariance] [, exc])

subroutine vamp_sample_grid (rng, g, func, iterations
    [, integral] [, std_dev] [, avg_chi2] [, accuracy] [, channel]
    [, weights] [, exc] [, history])

    func

     $S_n$  with  $n = \text{iterations}$ 

subroutine vamp_sample_grid0 (rng, g, func, [, channel]
    [, weights] [, exc])

    func

     $S_0$ 

subroutine vamp_refine_grid (g, [, exc])

     $r$ 

subroutine vamp_average_iterations (g, iteration, integral,
    std_dev, avg_chi2)

    integer, intent(in) :: iteration
    Number of iterations so far (needed for  $\chi^2$ ).

subroutine vamp_integrate (g, func, calls [, integral]
    [, std_dev] [, avg_chi2] [, accuracy] [, covariance])

    type(vamp_grid), intent(inout) :: g
    func
```

```

integer, dimension(:,:), intent(in) :: calls

subroutine vamp_integratex (region, func, calls [, integral]
[, std_dev] [, avg_chi2] [, stratified] [, accuracy] [, pancake]
[, cigar])

real, dimension(:,:), intent(in) :: region
func
integer, dimension(:,:), intent(in) :: calls
integer, intent(in) :: pancake
integer, intent(in) :: cigar

subroutine vamp_copy_grid (lhs, rhs)

type(vamp_grid), intent(inout) :: lhs
type(vamp_grid), intent(in) :: rhs

subroutine vamp_delete_grid (g)

type(vamp_grid), intent(inout) :: g

```

4.2.4 *Inout/Output and Marshling*

```

subroutine vamp_write_grid (g, [, ...])

type(vamp_grid), intent(inout) :: g

subroutine vamp_read_grid (g, [, ...])

type(vamp_grid), intent(inout) :: g

subroutine vamp_write_grids (g, [, ...])

type(vamp_grids), intent(inout) :: g

subroutine vamp_read_grids (g, [, ...])

type(vamp_grids), intent(inout) :: g

pure subroutine vamp_marshall_grid (g, integer_buffer,
double_buffer)

```

```

type(vamp_grid), intent(in) :: g
integer, dimension(:), intent(inout) ::
    integer_buffer
real(kind=default), dimension(:), intent(inout)
    :: double_buffer

```

Marshal the grid `g` in the integer array `integer_buffer` and the real array `double_buffer`, which must have at least the sizes obtained from call `vamp_marshall_grid_size (g, integer_size, double_size)`.



Note that we can not use the `transfer` intrinsic function for marshalling types that contain pointers that substitute for allocatable array components. `transfer` would copy the pointers in this case and not where they point to!

```

pure subroutine vamp_marshall_grid_size (g, integer_size,
    double_size)

```

```

type(vamp_grid), intent(in) :: g
integer :: words

```

Compute the sizes of the arrays required for marshaling the grid `g`.

```

pure subroutine vamp_unmarshal_grid (g, integer_buffer,
    double_buffer)

```

```

type(vamp_grid), intent(inout) :: g
integer, dimension(:), intent(in) ::
    integer_buffer
real(kind=default), dimension(:), intent(in) ::
    double_buffer

```

Marshaling and unmarshaling need to use two separate buffers for integers and floating point numbers. In a homogeneous network, the intrinsic procedure `transfer` could be used to store the floating point numbers in the integer array. In a heterogeneous network this will fail. However, message passing environments provide methods for sending floating point numbers. For example, here's how to send a grid from process 0 to process 1 in MPI [11]

29 *<MPI communication example 29>*≡
 call vamp_marshall_grid_size (g, isize, dsize)

```

allocate (ibuf(isize), dbuf(dsize))
call mpi_comm_rank (MPI_COMM_WORLD, proc_id, errno)
select case (proc_id)
  case (0)
    call vamp_marshall_grid (g, ibuf, dbuf)
    call mpi_send (ibuf, size (ibuf), MPI_INTEGER, &
                  1, 1, MPI_COMM_WORLD, errno)
    call mpi_send (dbuf, size (dbuf), MPI_DOUBLE_PRECISION, &
                  1, 2, MPI_COMM_WORLD, errno)
  case (1)
    call mpi_recv (ibuf, size (ibuf), MPI_INTEGER, &
                  0, 1, MPI_COMM_WORLD, status, errno)
    call mpi_recv (dbuf, size (dbuf), MPI_DOUBLE_PRECISION, &
                  0, 2, MPI_COMM_WORLD, status, errno)
    call vamp_unmarshal_grid (g, ibuf, dbuf)
end select

```

assuming that double is such that MPI_DOUBLE_PRECISION corresponds to real(kind=default). The module vampi provides two high level functions `vamp_send_grid` and `vamp_receive_grid` that handle the low level details:

```

30 <MPI communication example' 30>≡
  call mpi_comm_rank (MPI_COMM_WORLD, proc_id, errno)
  select case (proc_id)
    case (0)
      call vamp_send_grid (g, 1, 0)
    case (1)
      call vamp_receive_grid (g, 0, 0)
  end select

  subroutine vamp_marshall_history_size (g, [, ...])

    type(vamp_grid), intent(inout) :: g

  subroutine vamp_marshall_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g

  subroutine vamp_unmarshal_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g

```

4.2.5 Multi Channel Procedures

$$g \circ \phi_i = \left| \frac{\partial \phi_i}{\partial x} \right|^{-1} \left(\alpha_i g_i + \sum_{\substack{j=1 \\ j \neq i}}^{N_c} \alpha_j (g_j \circ \pi_{ij}) \left| \frac{\partial \pi_{ij}}{\partial x} \right| \right). \quad (4.4)$$

31a *Interface declaration for phi 31a* \equiv

```
interface
  pure function phi (xi, channel) result (x)
    use kinds
    real(kind=default), dimension(:), intent(in) :: xi
    integer, intent(in) :: channel
    real(kind=default), dimension(size(xi)) :: x
  end function phi
end interface
```

31b *Interface declaration for ihp 31b* \equiv

```
interface
  pure function ihp (x, channel) result (xi)
    use kinds
    real(kind=default), dimension(:), intent(in) :: x
    integer, intent(in) :: channel
    real(kind=default), dimension(size(x)) :: xi
  end function ihp
end interface
```

31c *Interface declaration for jacobian 31c* \equiv

```
interface
  pure function jacobian (x, prc_index, channel) result (j)
    use kinds
    use vamp_grid_type !NODEP!
    real(kind=default), dimension(:), intent(in) :: x
    integer, intent(in) :: prc_index
    integer, intent(in) :: channel
    real(kind=default) :: j
  end function jacobian
end interface
```

```
function vamp_multi_channel (func, phi, ihp, jacobian, x,
  weights1, grids)
```

```
  real(kind=default), dimension(:), intent(in) :: x
  real(kind=default), dimension(:), intent(in) ::
    weights
```



```

integer, intent(in) :: channel
type(vamp_grid), dimension(:), intent(in) ::
    grids

function vamp_multi_channel0 (func, phi, jacobian, x,
    weights1)

    real(kind=default), dimension(:), intent(in) :: x
    real(kind=default), dimension(:), intent(in) ::
        weights
    integer, intent(in) :: channel

subroutine vamp_check_jacobian (rng, n, channel, region,
    delta, [, x_delta])

    type(tao_random_state), intent(inout) :: rng
    integer, intent(in) :: n
    integer, intent(in) :: channel
    real(kind=default), dimension(:,:), intent(in) ::
        region
    real(kind=default), intent(out) :: delta
    real(kind=default), dimension(:), intent(out),
        optional :: x_delta

```

Verify that

$$g(\phi(x)) = \frac{1}{\left| \frac{\partial \phi}{\partial x} \right| (x)} \quad (4.5)$$

```

subroutine vamp_copy_grids (lhs, rhs)

    type(vamp_grids), intent(inout) :: lhs
    type(vamp_grids), intent(in) :: rhs

subroutine vamp_delete_grids (g)

    type(vamp_grids), intent(inout) :: g

subroutine vamp_create_grids (g, domain, num_calls, weights
    [, maps] [, stratified])

```

```

    type(vamp_grids), intent(inout) :: g
    real, dimension(:,:), intent(in) :: domain
    integer, intent(in) :: num_calls
    real, dimension(:), intent(in) :: weights
    real, dimension(:,:,:), intent(in) :: maps

subroutine vamp_create_empty_grids (g)

    type(vamp_grids), intent(inout) :: g

subroutine vamp_discard_integrals (g [, num_calls]
    [, stratified])

    type(vamp_grids), intent(inout) :: g
    integer, intent(in) :: num_calls

subroutine vamp_refine_weights (g [, power)

    type(vamp_grids), intent(inout) :: g
    real, intent(in) :: power

subroutine vamp_update_weights (g, weights [, num_calls]
    [, stratified])

    type(vamp_grids), intent(inout) :: g
    real, dimension(:), intent(in) :: weights
    integer, intent(in) :: num_calls

subroutine vamp_reshape_grids (g, num_calls [, stratified])

    type(vamp_grids), intent(inout) :: g
    integer, intent(in) :: num_calls

subroutine vamp_reduce_channels (g, [, ...])

    type(vamp_grid), intent(inout) :: g

subroutine vamp_sample_grids (g, func, iterations [, integral]
    [, std_dev] [, accuracy] [, covariance] [, variance])

```

```

    type(vamp_grids), intent(inout) :: g
    func
    integer, intent(in) :: iterations
function vamp_sum_channels (x, weights, func)

    real, dimension(:), intent(in) :: x
    real, dimension(:), intent(in) :: weights
    func

```

4.2.6 Event Generation

```

subroutine vamp_next_event (g, [, ...])
subroutine vamp_warmup_grid (g, [, ...])

    type(vamp_grid), intent(inout) :: g
    func
    integer, intent(in) :: iterations
subroutine vamp_warmup_grids (g, [, ...])

    type(vamp_grids), intent(inout) :: g
    func
    integer, intent(in) :: iterations

```

4.2.7 Parallelization

```

subroutine vamp_fork_grid (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_join_grid (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_fork_grid_joints (g, [, ...])

```

```

    type(vamp_grid), intent(inout) :: g
subroutine vamp_sample_grid_parallel (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_distribute_work (g, [, ...])

    type(vamp_grid), intent(inout) :: g

```

4.2.8 *Diagnostics*

```

subroutine vamp_create_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_copy_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_delete_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_terminate_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_get_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_get_history_single (g, [, ...])

    type(vamp_grid), intent(inout) :: g
subroutine vamp_print_history (g, [, ...])

    type(vamp_grid), intent(inout) :: g

```

 Discuss why the value of the integral in each channel differs.

4.2.9 *Other Procedures*

```
subroutine vamp_rigid_divisions (g, [, ...])  
    type(vamp_grid), intent(inout) :: g  
function vamp_get_covariance (g, [, ...])  
    type(vamp_grid), intent(inout) :: g  
subroutine vamp_nullify_covariance (g, [, ...])  
    type(vamp_grid), intent(inout) :: g  
function vamp_get_variance (g, [, ...])  
    type(vamp_grid), intent(inout) :: g  
subroutine vamp_nullify_variance (g, [, ...])  
    type(vamp_grid), intent(inout) :: g
```

4.2.10 *(Currently) Undocumented Procedures*


```
subroutine (... , [, ...])  
function (... , [, ...])
```

—5—

IMPLEMENTATION

5.1 *The Abstract Datatype `division`*

```
37a <divisions.f90 37a>≡
! divisions.f90 --
<Copyleft notice 1>
module divisions
  use kinds
  use exceptions
  use vamp_stat
  use utils
  use iso_fortran_env
  implicit none
  private
  <Declaration of divisions procedures 38a>
  <Interfaces of divisions procedures 60e>
  <Variables in divisions 45d>
  <Declaration of divisions types 37b>
  <Constants in divisions 64a>
  character(len=*), public, parameter :: DIVISIONS_RCS_ID = &
    "$Id: divisions.nw 314 2010-04-17 20:32:33Z ohl $"
  contains
    <Implementation of divisions procedures 38b>
end module divisions
```

 vamp_apply_equivalences from vamp accesses %variance ...

```
37b <Declaration of divisions types 37b>≡
type, public :: division
  private
  real(kind=default), dimension(:), pointer :: x => null ()
```

```

real(kind=default), dimension(:), pointer :: integral => null ()
real(kind=default), dimension(:), pointer, &
                                public :: variance => null ()
! real(kind=default), dimension(:), pointer :: efficiency => null ()
real(kind=default) :: x_min, x_max
real(kind=default) :: x_min_true, x_max_true
real(kind=default) :: dx, dxg
integer :: ng = 0
logical :: stratified = .true.
end type division

```

5.1.1 Creation, Manipulation & Injection

38a *<Declaration of divisions procedures 38a>*≡

```

public :: create_division, create_empty_division
public :: copy_division, delete_division
public :: set_rigid_division, reshape_division

```

38b *<Implementation of divisions procedures 38b>*≡

```

elemental subroutine create_division &
    (d, x_min, x_max, x_min_true, x_max_true)
type(division), intent(out) :: d
real(kind=default), intent(in) :: x_min, x_max
real(kind=default), intent(in), optional :: x_min_true, x_max_true
allocate (d%x(0:1), d%integral(1), d%variance(1))
! allocate (d%efficiency(1))
d%x(0) = 0.0
d%x(1) = 1.0
d%x_min = x_min
d%x_max = x_max
d%dx = d%x_max - d%x_min
d%stratified = .false.
d%ng = 1
d%dxg = 1.0 / d%ng
if (present (x_min_true)) then
    d%x_min_true = x_min_true
else
    d%x_min_true = x_min
end if
if (present (x_max_true)) then
    d%x_max_true = x_max_true
else
    d%x_max_true = x_max

```

```

        end if
    end subroutine create_division

39a ⟨Implementation of divisions procedures 38b⟩+≡
    elemental subroutine create_empty_division (d)
        type(division), intent(out) :: d
        nullify (d%x, d%integral, d%variance)
        ! nullify (d%efficiency)
    end subroutine create_empty_division

39b ⟨Implementation of divisions procedures 38b⟩+≡
    elemental subroutine set_rigid_division (d, ng)
        type(division), intent(inout) :: d
        integer, intent(in) :: ng
        d%stratified = ng > 1
        d%ng = ng
        d%dxg = real (ubound (d%x, dim=1), kind=default) / d%ng
    end subroutine set_rigid_division


$$\text{dxg} = \frac{n_{\text{div}}}{n_g} \quad (5.1)$$


such that  $0 < \text{cell} \cdot \text{dxg} < n_{\text{div}}$ 

39c ⟨Implementation of divisions procedures 38b⟩+≡
    elemental subroutine reshape_division (d, max_num_div, ng, use_variance)
        type(division), intent(inout) :: d
        integer, intent(in) :: max_num_div
        integer, intent(in), optional :: ng
        logical, intent(in), optional :: use_variance
        real(kind=default), dimension(:), allocatable :: old_x, m
        integer :: num_div, equ_per_adap
        if (present (ng)) then
            if (max_num_div > 1) then
                d%stratified = ng > 1
            else
                d%stratified = .false.
            end if
        else
            d%stratified = .false.
        end if
        if (d%stratified) then
            d%ng = ng
            ⟨Initialize stratified sampling 42⟩
        else
            num_div = max_num_div
            d%ng = 1

```


n_{dim}	$N_{\text{calls}}^{\text{max}}(n_g = 25)$
2	$1 \cdot 10^3$
3	$3 \cdot 10^4$
4	$8 \cdot 10^5$
5	$2 \cdot 10^7$
6	$5 \cdot 10^8$

Table 5.1: To stratify or not to stratify.

```

end if
d%dxg = real (num_div, kind=default) / d%ng
allocate (old_x(0:ubound(d%x,dim=1)), m(ubound(d%x,dim=1)))
old_x = d%x
<Set m to (1,1,...) or to rebinning weights from d%variance 40a>
<Resize arrays, iff necessary 40b>
d%x = rebin (m, old_x, num_div)
deallocate (old_x, m)
end subroutine reshape_division

40a <Set m to (1,1,...) or to rebinning weights from d%variance 40a>≡
  if (present (use_variance)) then
    if (use_variance) then
      m = rebinning_weights (d%variance)
    else
      m = 1.0
    end if
  else
    m = 1.0
  end if

40b <Resize arrays, iff necessary 40b>≡
  if (ubound (d%x, dim=1) /= num_div) then
    deallocate (d%x, d%integral, d%variance)
    ! deallocate (d%efficiency)
    allocate (d%x(0:num_div), d%integral(num_div), d%variance(num_div))
    ! allocate (d%efficiency(num_div))
  end if

```

Genuinely stratified sampling will superimpose an equidistant grid on the adaptive grid, as shown in figure 5.2. Obviously, this is only possible when the number of cells of the stratification grid is large enough, specifically when $n_g \geq n_{\text{div}}^{\text{min}} = n_{\text{div}}^{\text{max}}/2 = 25$. This condition can be met by a high number of sampling points or by a low dimensionality of the integration region (cf. table 5.1).



Figure 5.1: **vegas** grid structure for non-stratified sampling. N.B.: the grid and the weight functions $p_{1,2}$ are only in qualitative agreement.



Figure 5.2: **vegas** grid structure for genuinely stratified sampling, which is used in low dimensions. N.B.: the grid and the weight functions $p_{1,2}$ are only in qualitative agreement.

For a low number of sampling points and high dimensions, genuinely stratified sampling is impossible, because we would have to reduce the number n_{div} of adaptive divisions too far. Instead, we keep **stratified** false which will tell the integration routine not to concentrate the grid in the regions where the contribution to the error is largest, but to use importance sampling, i. e. concentrating the grid in the regions where the contribution to the value is largest.

In this case, the rigid grid is much coarser than the adaptive grid and furthermore, the boundaries of the cells overlap in general. The interplay of the two grids during the sampling process is shown in figure 5.3. First we determine the (integer) number k of equidistant divisions of an adaptive cell for at most $n_{\text{div}}^{\text{max}}$ divisions of the adaptive grid

$$k = \left\lfloor \frac{n_g}{n_{\text{div}}^{\text{max}}} \right\rfloor + 1 \quad (5.2a)$$

and the corresponding number n_{div} of adaptive divisions

$$n_{\text{div}} = \left\lfloor \frac{n_g}{k} \right\rfloor \quad (5.2b)$$

Finally, adjust n_g to an exact multiple of n_{div}

$$n_g = k \cdot n_{\text{div}} \quad (5.2c)$$

```

42  Initialize stratified sampling 42)≡
    if (d%ng >= max_num_div / 2) then
        d%stratified = .true.
        equ_per_adap = d%ng / max_num_div + 1
        num_div = d%ng / equ_per_adap
        if (num_div < 2) then
            d%stratified = .false.
            num_div = 2
            d%ng = 1
        else if (mod (num_div,2) == 1) then
            num_div = num_div - 1
            d%ng = equ_per_adap * num_div
        else
            d%ng = equ_per_adap * num_div
        end if
    else
        d%stratified = .false.
        num_div = max_num_div
        d%ng = 1
    end if

```



Figure 5.3: One-dimensional illustration of the **vegas** grid structure for pseudo stratified sampling, which is used in high dimensions.

Figure 5.3 on page 43 is a one-dimensional illustration of the sampling algorithm. In each cell of the rigid equidistant grid, two random points are selected (or N_{calls} in the not stratified case). For each point, the corresponding cell and relative coordinate in the adaptive grid is found, *as if the adaptive grid was equidistant* (upper arrow). Then this point is mapped according to the adapted grid (lower arrow) and the proper Jacobians are applied to the weight.

$$\prod_{j=1}^n (x_i^j - x_{i-1}^j) \cdot N^n = \text{Vol}(\text{cell}') \cdot \frac{1}{\text{Vol}(\text{cell})} = \frac{1}{p(x_i^j)} \quad (5.3)$$

```

43a  <Declaration of divisions procedures 38a>+≡
      public :: inject_division, inject_division_short

43b  <Implementation of divisions procedures 38b>+≡
      elemental subroutine inject_division (d, r, cell, x, x_mid, idx, wgt)
        type(division), intent(in) :: d
        real(kind=default), intent(in) :: r
        integer, intent(in) :: cell
        real(kind=default), intent(out) :: x, x_mid
        integer, intent(out) :: idx
        real(kind=default), intent(out) :: wgt
        real(kind=default) :: delta_x, xi
        integer :: i
        xi = (cell - r) * d%dxg + 1.0
        <Set i, delta_x, x, and wgt from xi 43c>
        idx = i
        x_mid = d%x_min + 0.5 * (d%x(i-1) + d%x(i)) * d%dx
      end subroutine inject_division

43c  <Set i, delta_x, x, and wgt from xi 43c>≡
      i = max (min (int (xi), ubound (d%x, dim=1)), 1)
      delta_x = d%x(i) - d%x(i-1)

```

```

x = d%x_min + (d%x(i-1) + (xi - i) * delta_x) * d%dx
wgt = delta_x * ubound (d%x, dim=1)

```

44a *<Implementation of divisions procedures 38b>+≡*

```

elemental subroutine inject_division_short (d, r, x, idx, wgt)
  type(division), intent(in) :: d
  real(kind=default), intent(in) :: r
  integer, intent(out) :: idx
  real(kind=default), intent(out) :: x, wgt
  real(kind=default) :: delta_x, xi
  integer :: i
  xi = r * ubound (d%x, dim=1) + 1.0
  <Set i, delta_x, x, and wgt from xi 43c>
  idx = i
end subroutine inject_division_short

```

5.1.2 Grid Refinement

44b *<Declaration of divisions procedures 38a>+≡*

```

public :: record_integral, record_variance, clear_integral_and_variance
! public :: record_efficiency

```

44c *<Implementation of divisions procedures 38b>+≡*

```

elemental subroutine record_integral (d, i, f)
  type(division), intent(inout) :: d
  integer, intent(in) :: i
  real(kind=default), intent(in) :: f
  d%integral(i) = d%integral(i) + f
  if (.not. d%stratified) then
    d%variance(i) = d%variance(i) + f*f
  end if
end subroutine record_integral

```

44d *<Implementation of divisions procedures 38b>+≡*

```

elemental subroutine record_variance (d, i, var_f)
  type(division), intent(inout) :: d
  integer, intent(in) :: i
  real(kind=default), intent(in) :: var_f
  if (d%stratified) then
    d%variance(i) = d%variance(i) + var_f
  end if
end subroutine record_variance

```

44e *<Implementation of divisions procedures (removed from WHIZARD) 44e>≡*

```

elemental subroutine record_efficiency (d, i, eff)

```

```

    type(division), intent(inout) :: d
    integer, intent(in) :: i
    real(kind=default), intent(in) :: eff
    ! d%efficiency(i) = d%efficiency(i) + eff
end subroutine record_efficiency

```

45a *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental subroutine clear_integral_and_variance (d)
 type(division), intent(inout) :: d
 d%integral = 0.0
 d%variance = 0.0
 ! d%efficiency = 0.0
 end subroutine clear_integral_and_variance

45b *⟨Declaration of divisions procedures 38a⟩*+≡
 public :: refine_division

45c *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental subroutine refine_division (d)
 type(division), intent(inout) :: d
 character(len=*), parameter :: FN = "refine_division"
 d%x = rebin (rebinning_weights (d%variance), d%x, size (d%variance))
 end subroutine refine_division

Smooth the $d_i = \bar{f}_i \Delta x_i$

$$\begin{aligned}
 d_1 &\rightarrow \frac{1}{2}(d_1 + d_2) \\
 d_2 &\rightarrow \frac{1}{3}(d_1 + d_2 + d_3) \\
 &\dots \\
 d_{n-1} &\rightarrow \frac{1}{3}(d_{n-2} + d_{n-1} + d_n) \\
 d_n &\rightarrow \frac{1}{2}(d_{n-1} + d_n)
 \end{aligned} \tag{5.4}$$

As long as the initial `num_div` ≥ 6 , we know that `num_div` ≥ 3 .

45d *⟨Variables in divisions 45d⟩*≡
 integer, private, parameter :: MIN_NUM_DIV = 3

Here the Fortran90 array notation really shines, but we have to handle the cases `nd` ≤ 2 specially, because the `quadrupole` option can lead to small `nds`. The equivalent Fortran77 code [2] is orders of magnitude less obvious ¹ Also

¹Some old timers call this a feature, however.

protect against vanishing d_i that will blow up the logarithm.

$$m_i = \left(\frac{\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j} - 1}{\ln \left(\frac{\bar{f}_i \Delta x_i}{\sum_j \bar{f}_j \Delta x_j} \right)} \right)^\alpha \quad (5.5)$$

46a *⟨Implementation of divisions procedures 38b⟩*+≡

```
pure function rebinning_weights (d) result (m)
  real(kind=default), dimension(:), intent(in) :: d
  real(kind=default), dimension(size(d)) :: m
  real(kind=default), dimension(size(d)) :: smooth_d
  real(kind=default), parameter :: ALPHA = 1.5
  integer :: nd
  ⟨Bail out if any (d == NaN) 46c⟩
  nd = size (d)
  if (nd > 2) then
    smooth_d(1) = (d(1) + d(2)) / 2.0
    smooth_d(2:nd-1) = (d(1:nd-2) + d(2:nd-1) + d(3:nd)) / 3.0
    smooth_d(nd) = (d(nd-1) + d(nd)) / 2.0
  else
    smooth_d = d
  end if
  if (all (smooth_d < tiny (1.0_default))) then
    m = 1.0_default
  else
    smooth_d = smooth_d / sum (smooth_d)
    where (smooth_d < tiny (1.0_default))
      smooth_d = tiny (1.0_default)
    end where
    where (smooth_d /= 1._default)
      m = ((smooth_d - 1.0) / (log (smooth_d)))**ALPHA
    elsewhere
      m = 1.0_default
    endwhere
  end if
end function rebinning_weights
```

46b *⟨Declaration of divisions procedures 38a⟩*+≡

```
private :: rebinning_weights
```



The NaN test is probably not portable:

46c *⟨Bail out if any (d == NaN) 46c⟩*≡

```
if (any (d /= d)) then
  m = 1.0
```

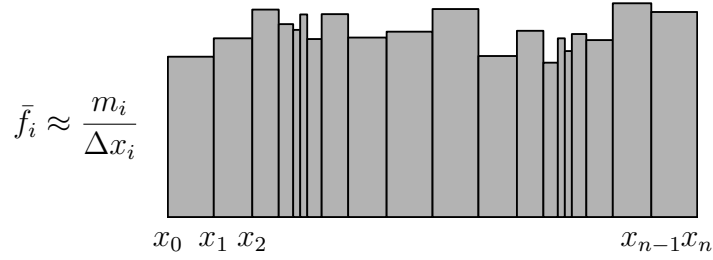


Figure 5.4: Typical weights used in the rebinning algorithm.

```

return
end if

```

Take a binning x and return a new binning with `num_div` bins with the m homogeneously distributed:

```

47a  <Implementation of divisions procedures 38b>+≡
      pure function rebin (m, x, num_div) result (x_new)
        real(kind=default), dimension(:), intent(in) :: m
        real(kind=default), dimension(0:), intent(in) :: x
        integer, intent(in) :: num_div
        real(kind=default), dimension(0:num_div) :: x_new
        integer :: i, k
        real(kind=default) :: step, delta
        step = sum (m) / num_div
        k = 0
        delta = 0.0
        x_new(0) = x(0)
        do i = 1, num_div - 1
          <Increment k until  $\sum m_k \geq \Delta$  and keep the surplus in  $\delta$  47c>
          <Interpolate the new  $x_i$  from  $x_k$  and  $\delta$  48a>
        end do
        x_new(num_div) = 1.0
      end function rebin

```

```

47b  <Declaration of divisions procedures 38a>+≡
      private :: rebin

```

We increment k until another Δ (a.k.a. `step`) of the integral has been accumulated (cf. figure 5.4). The mismatch will be corrected below.

```

47c  <Increment k until  $\sum m_k \geq \Delta$  and keep the surplus in  $\delta$  47c>≡
      do
        if (step <= delta) then
          exit

```



```

        end if
        k = k + 1
        delta = delta + m(k)
    end do
    delta = delta - step
48a ⟨Interpolate the new  $x_i$  from  $x_k$  and  $\delta$  48a⟩≡
    x_new(i) = x(k) - (x(k) - x(k-1)) * delta / m(k)

```

5.1.3 Probability Density

48b *⟨Declaration of divisions procedures 38a⟩*+≡
 public :: probability

$$\xi = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \in [0, 1] \quad (5.6)$$

and

$$\int_{x_{\min}}^{x_{\max}} dx p(x) = 1 \quad (5.7)$$

48c *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental function probability (d, x) result (p)
 type(division), intent(in) :: d
 real(kind=default), intent(in) :: x
 real(kind=default) :: p
 real(kind=default) :: xi
 integer :: hi, mid, lo
 xi = (x - d%x_min) / d%dx
 if ((xi >= 0) .and. (xi <= 1)) then
 lo = lbound (d%x, dim=1)
 hi = ubound (d%x, dim=1)
 bracket: do
 if (lo >= hi - 1) then
 p = 1.0 / (ubound (d%x, dim=1) * d%dx * (d%x(hi) - d%x(hi-1)))
 return
 end if
 mid = (hi + lo) / 2
 if (xi > d%x(mid)) then
 lo = mid
 else
 hi = mid
 end if
 end do bracket
 else
 p = 0
 end if

```

    end if
end function probability

```

5.1.4 *Quadrupole*

```

49a <Declaration of divisions procedures 38a>+≡
    public :: quadrupole_division

49b <Implementation of divisions procedures 38b>+≡
    elemental function quadrupole_division (d) result (q)
        type(division), intent(in) :: d
        real(kind=default) :: q
        !!!    q = value_spread_percent (rebinning_weights (d%variance))
        q = standard_deviation_percent (rebinning_weights (d%variance))
    end function quadrupole_division

```

5.1.5 *Forking and Joining*

The goal is to split a division in such a way, that we can later sample the pieces separately and combine the results.

```

49c <Declaration of divisions procedures 38a>+≡
    public :: fork_division, join_division, sum_division

```



Caveat emptor: splitting divisions can lead to `num_div < 3` and the application *must not* try to refine such grids before merging them again!

```

49d <Implementation of divisions procedures 38b>+≡
    pure subroutine fork_division (d, ds, sum_calls, num_calls, exc)
        type(division), intent(in) :: d
        type(division), dimension(:), intent(inout) :: ds
        integer, intent(in) :: sum_calls
        integer, dimension(:), intent(inout) :: num_calls
        type(exception), intent(inout), optional :: exc
        character(len=*), parameter :: FN = "fork_division"
        integer, dimension(size(ds)) :: n0, n1
        integer, dimension(0:size(ds)) :: n, ds_ng
        integer :: i, j, num_div, num_forks, nx
        real(kind=default), dimension(:), allocatable :: d_x, d_integral, d_variance
        ! real(kind=default), dimension(:), allocatable :: d_efficiency
        num_div = ubound (d%x, dim=1)
        num_forks = size (ds)
        if (d%ng == 1) then
            <Fork an importance sampling division 51a>

```

```

else if (num_div >= num_forks) then
  if (modulo (d%ng, num_div) == 0) then
    ⟨Fork a pure stratified sampling division 51d⟩
  else
    ⟨Fork a pseudo stratified sampling division 52c⟩
  end if
else
  if (present (exc)) then
    call raise_exception (exc, EXC_FATAL, FN, "internal error")
  end if
  num_calls = 0
end if
end subroutine fork_division

```

50 *⟨Implementation of divisions procedures 38b⟩*+≡

```

pure subroutine join_division (d, ds, exc)
  type(division), intent(inout) :: d
  type(division), dimension(:), intent(in) :: ds
  type(exception), intent(inout), optional :: exc
  character(len=*), parameter :: FN = "join_division"
  integer, dimension(size(ds)) :: n0, n1
  integer, dimension(0:size(ds)) :: n, ds_ng
  integer :: i, j, num_div, num_forks, nx
  real(kind=default), dimension(:), allocatable :: d_x, d_integral, d_variance
! real(kind=default), dimension(:), allocatable :: d_efficiency
  num_div = ubound (d%x, dim=1)
  num_forks = size (ds)
  if (d%ng == 1) then
    ⟨Join importance sampling divisions 51b⟩
  else if (num_div >= num_forks) then
    if (modulo (d%ng, num_div) == 0) then
      ⟨Join pure stratified sampling divisions 52a⟩
    else
      ⟨Join pseudo stratified sampling divisions 54a⟩
    end if
  else
    if (present (exc)) then
      call raise_exception (exc, EXC_FATAL, FN, "internal error")
    end if
  end if
end subroutine join_division

```

Importance Sampling

Importance sampling ($d\%ng == 1$) is trivial, since we can just sample `size(ds)` copies of the same grid with (almost) the same number of points

```
51a <Fork an importance sampling division 51a>≡
    if (d%stratified) then
        call raise_exception (exc, EXC_FATAL, FN, &
                               "ng == 1 incompatible w/ stratification")
    else
        call copy_division (ds, d)
        num_calls(2:) = ceiling (real (sum_calls) / num_forks)
        num_calls(1) = sum_calls - sum (num_calls(2:))
    end if
```

and sum up the results in the end:

```
51b <Join importance sampling divisions 51b>≡
    call sum_division (d, ds)
```

Note, however, that this is only legitimate as long as $d\%ng == 1$ implies $d\%stratified == .false.$, because otherwise the sampling code would be incorrect (cf. `var_f` on page 88).

Stratified Sampling

For stratified sampling, we have to work a little harder, because there are just two points per cell and we have to slice along the lines of the stratification grid. Actually, we are slicing along the adaptive grid, since it has a reasonable size. Slicing along the stratification grid could be done using the method below. However, in this case *very* large adaptive grids would be shipped from one process to the other and the communication costs will outweigh the gains from parallel processing.

```
51c <Setup to fork a pure stratified sampling division 51c>≡
    n = (num_div * (/ (j, j=0,num_forks) /)) / num_forks
    n0(1:num_forks) = n(0:num_forks-1)
    n1(1:num_forks) = n(1:num_forks)

51d <Fork a pure stratified sampling division 51d>≡
    <Setup to fork a pure stratified sampling division 51c>
    do i = 1, num_forks
        call copy_array_pointer (ds(i)%x, d%x(n0(i):n1(i)), lb = 0)
        call copy_array_pointer (ds(i)%integral, d%integral(n0(i)+1:n1(i)))
        call copy_array_pointer (ds(i)%variance, d%variance(n0(i)+1:n1(i)))
        ! call copy_array_pointer (ds(i)%efficiency, d%efficiency(n0(i)+1:n1(i)))
        ds(i)%x = (ds(i)%x - ds(i)%x(0)) / (d%x(n1(i)) - d%x(n0(i)))
    end do
```

```

ds%x_min = d%x_min + d%dx * d%x(n0)
ds%x_max = d%x_min + d%dx * d%x(n1)
ds%dx = ds%x_max - ds%x_min
ds%x_min_true = d%x_min_true
ds%x_max_true = d%x_max_true
ds%stratified = d%stratified
ds%ng = (d%ng * (n1 - n0)) / num_div
num_calls = sum_calls ! this is a misnomer, it remains "calls per cell" here
ds%dxg = real (n1 - n0, kind=default) / ds%ng

```

Joining is the exact inverse, but we're only interested in `d%integral` and `d%variance` for the grid refinement:

```

52a <Join pure stratified sampling divisions 52a>≡
    <Setup to fork a pure stratified sampling division 51c>
    do i = 1, num_forks
        d%integral(n0(i)+1:n1(i)) = ds(i)%integral
        d%variance(n0(i)+1:n1(i)) = ds(i)%variance
        ! d%efficiency(n0(i)+1:n1(i)) = ds(i)%efficiency
    end do

```

Pseudo Stratified Sampling

The coarsest grid covering the division of n_g bins into n_f forks has $n_g / \gcd(n_f, n_g) = \text{lcm}(n_f, n_g) / n_f$ bins per fork. Therefore, we need

$$\text{lcm}\left(\frac{\text{lcm}(n_f, n_g)}{n_f}, n_x\right) \quad (5.8)$$

divisions of the adaptive grid (if n_x is the number of bins in the original adaptive grid).

Life would be much easier, if we knew that n_f divides n_g . However, this is hard to maintain in real life applications. We can try to achieve this if possible, but the algorithms must be prepared to handle the general case.

```

52b <Setup to fork a pseudo stratified sampling division 52b>≡
    nx = lcm (d%ng / gcd (num_forks, d%ng), num_div)
    ds_ng = (d%ng * (/ (j, j=0,num_forks) /)) / num_forks
    n = (nx * ds_ng) / d%ng
    n0(1:num_forks) = n(0:num_forks-1)
    n1(1:num_forks) = n(1:num_forks)

52c <Fork a pseudo stratified sampling division 52c>≡
    <Setup to fork a pseudo stratified sampling division 52b>
    allocate (d_x(0:nx), d_integral(nx), d_variance(nx))
    ! allocate (d_efficiency(nx))

```

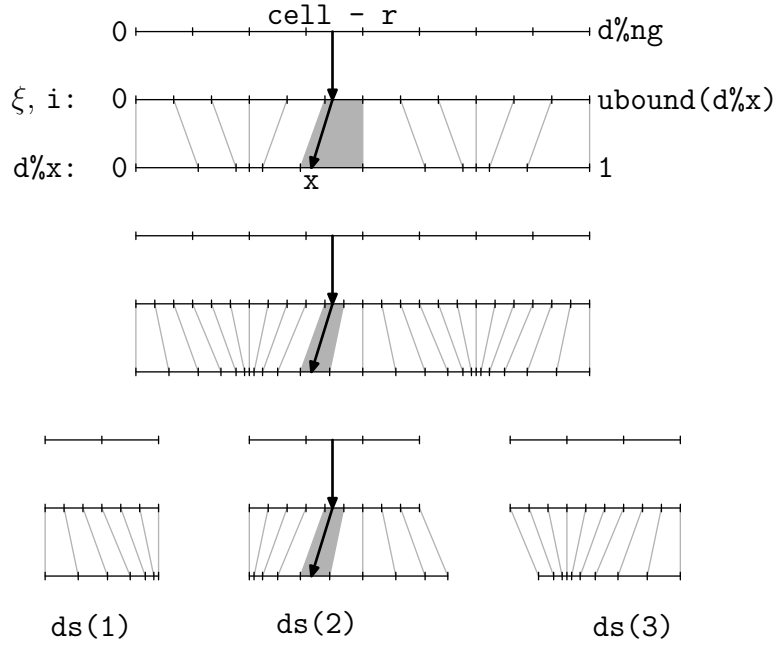


Figure 5.5: Forking one dimension d of a grid into three parts $ds(1)$, $ds(2)$, and $ds(3)$. The picture illustrates the most complex case of pseudo stratified sampling (cf. fig. 5.3).

```

call subdivide (d_x, d%x)
call distribute (d_integral, d%integral)
call distribute (d_variance, d%variance)
! call distribute (d_efficiency, d%efficiency)
do i = 1, num_forks
  call copy_array_pointer (ds(i)%x, d_x(n0(i):n1(i)), lb = 0)
  call copy_array_pointer (ds(i)%integral, d_integral(n0(i)+1:n1(i)))
  call copy_array_pointer (ds(i)%variance, d_variance(n0(i)+1:n1(i)))
!  call copy_array_pointer (ds(i)%efficiency, d_efficiency(n0(i)+1:n1(i)))
  ds(i)%x = (ds(i)%x - ds(i)%x(0)) / (d_x(n1(i)) - d_x(n0(i)))
end do
ds%x_min = d%x_min + d%dx * d_x(n0)
ds%x_max = d%x_min + d%dx * d_x(n1)
ds%dx = ds%x_max - ds%x_min
ds%x_min_true = d%x_min_true
ds%x_max_true = d%x_max_true
ds%stratified = d%stratified
ds%ng = ds_ng(1:num_forks) - ds_ng(0:num_forks-1)
num_calls = sum_calls ! this is a misnomer, it remains "calls per cell" here

```

```

ds%dxg = real (n1 - n0, kind=default) / ds%ng
deallocate (d_x, d_integral, d_variance)
! deallocate (d_efficiency)

54a <Join pseudo stratified sampling divisions 54a>≡
<Setup to fork a pseudo stratified sampling division 52b>
allocate (d_x(0:nx), d_integral(nx), d_variance(nx))
! allocate (d_efficiency(nx))
do i = 1, num_forks
    d_integral(n0(i)+1:n1(i)) = ds(i)%integral
    d_variance(n0(i)+1:n1(i)) = ds(i)%variance
    ! d_efficiency(n0(i)+1:n1(i)) = ds(i)%efficiency
end do
call collect (d%integral, d_integral)
call collect (d%variance, d_variance)
! call collect (d%efficiency, d_efficiency)
deallocate (d_x, d_integral, d_variance)
! deallocate (d_efficiency)

54b <Declaration of divisions procedures 38a>+≡
private :: subdivide
private :: distribute
private :: collect

54c <Implementation of divisions procedures 38b>+≡
pure subroutine subdivide (x, x0)
    real(kind=default), dimension(0:), intent(inout) :: x
    real(kind=default), dimension(0:), intent(in) :: x0
    integer :: i, n, n0
    n0 = ubound (x0, dim=1)
    n = ubound (x, dim=1) / n0
    x(0) = x0(0)
    do i = 1, n
        x(i:n) = x0(0:n0-1) * real (n - i) / n + x0(1:n0) * real (i) / n
    end do
end subroutine subdivide

54d <Implementation of divisions procedures 38b>+≡
pure subroutine distribute (x, x0)
    real(kind=default), dimension(:), intent(inout) :: x
    real(kind=default), dimension(:), intent(in) :: x0
    integer :: i, n
    n = ubound (x, dim=1) / ubound (x0, dim=1)
    do i = 1, n
        x(i:n) = x0 / n
    end do

```

```
end subroutine distribute
```

55a *⟨Implementation of divisions procedures 38b⟩+≡*

```
pure subroutine collect (x0, x)
  real(kind=default), dimension(:), intent(inout) :: x0
  real(kind=default), dimension(:), intent(in) :: x
  integer :: i, n, n0
  n0 = ubound (x0, dim=1)
  n = ubound (x, dim=1) / n0
  do i = 1, n0
    x0(i) = sum (x((i-1)*n+1:i*n))
  end do
end subroutine collect
```

Trivia

55b *⟨Implementation of divisions procedures 38b⟩+≡*

```
pure subroutine sum_division (d, ds)
  type(division), intent(inout) :: d
  type(division), dimension(:), intent(in) :: ds
  integer :: i
  d%integral = 0.0
  d%variance = 0.0
  ! d%efficiency = 0.0
  do i = 1, size (ds)
    d%integral = d%integral + ds(i)%integral
    d%variance = d%variance + ds(i)%variance
    ! d%efficiency = d%efficiency + ds(i)%efficiency
  end do
end subroutine sum_division
```

55c *⟨Declaration of divisions procedures 38a⟩+≡*

```
public :: debug_division
public :: dump_division
```

55d *⟨Implementation of divisions procedures 38b⟩+≡*

```
subroutine debug_division (d, prefix)
  type(division), intent(in) :: d
  character(len=*), intent(in) :: prefix
  print "(1x,a,2(a,1x,i3,1x,f10.7))", prefix, ": d%x: ", &
    lbound(d%x,dim=1), d%x(lbound(d%x,dim=1)), &
    " ... ", &
    ubound(d%x,dim=1), d%x(ubound(d%x,dim=1))
  print "(1x,a,2(a,1x,i3,1x,f10.7))", prefix, ": d%i: ", &
    lbound(d%integral,dim=1), d%integral(lbound(d%integral,dim=1)), &
```



```

" ... ", &
ubound(d%integral,dim=1), d%integral(ubound(d%integral,dim=1))
print "(1x,a,2(a,1x,i3,1x,f10.7))", prefix, ": d%v: ", &
lbound(d%variance,dim=1), d%variance(lbound(d%variance,dim=1)), &
" ... ", &
ubound(d%variance,dim=1), d%variance(ubound(d%variance,dim=1))
! print "(1x,a,2(a,1x,i3,1x,f10.7))", prefix, ": d%e: ", &
! lbound(d%efficiency,dim=1), d%efficiency(lbound(d%efficiency,dim=1)), &
! " ... ", &
! ubound(d%efficiency,dim=1), d%efficiency(ubound(d%efficiency,dim=1))
end subroutine debug_division

```

56a *<Implementation of divisions procedures 38b>+≡*

```

subroutine dump_division (d, prefix)
  type(division), intent(in) :: d
  character(len=*), intent(in) :: prefix
! print "(2(1x,a),100(1x,f10.7))", prefix, ":x: ", d%x
  print "(2(1x,a),100(1x,f10.7))", prefix, ":x: ", d%x(1:)
  print "(2(1x,a),100(1x,e10.3))", prefix, ":i: ", d%integral
  print "(2(1x,a),100(1x,e10.3))", prefix, ":v: ", d%variance
! print "(2(1x,a),100(1x,e10.3))", prefix, ":e: ", d%efficiency
end subroutine dump_division

```

5.1.6 Inquiry

Trivial, but necessary for making divisions an abstract data type:

56b *<Declaration of divisions procedures 38a>+≡*

```

public :: inside_division, stratified_division
public :: volume_division, rigid_division, adaptive_division

```

56c *<Implementation of divisions procedures 38b>+≡*

```

elemental function inside_division (d, x) result (theta)
  type(division), intent(in) :: d
  real(kind=default), intent(in) :: x
  logical :: theta
  theta = (x >= d%x_min_true) .and. (x <= d%x_max_true)
end function inside_division

```

56d *<Implementation of divisions procedures 38b>+≡*

```

elemental function stratified_division (d) result (yorn)
  type(division), intent(in) :: d
  logical :: yorn
  yorn = d%stratified
end function stratified_division

```

57a *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental function volume_division (d) result (vol)
 type(division), intent(in) :: d
 real(kind=default) :: vol
 vol = d%dx
end function volume_division

57b *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental function rigid_division (d) result (n)
 type(division), intent(in) :: d
 integer :: n
 n = d%ng
end function rigid_division

57c *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental function adaptive_division (d) result (n)
 type(division), intent(in) :: d
 integer :: n
 n = ubound (d%x, dim=1)
end function adaptive_division

5.1.7 Diagnostics

57d *⟨Declaration of divisions types 37b⟩*+≡
 type, public :: div_history
 private
 logical :: stratified
 integer :: ng, num_div
 real(kind=default) :: x_min, x_max, x_min_true, x_max_true
 real(kind=default) :: &
 spread_f_p, stddev_f_p, spread_p, stddev_p, spread_m, stddev_m
end type div_history

57e *⟨Declaration of divisions procedures 38a⟩*+≡
 public :: copy_history, summarize_division

57f *⟨Implementation of divisions procedures 38b⟩*+≡
 elemental function summarize_division (d) result (s)
 type(division), intent(in) :: d
 type(div_history) :: s
 real(kind=default), dimension(:), allocatable :: p, m
 allocate (p(ubound(d%x,dim=1)), m(ubound(d%x,dim=1)))
 p = probabilities (d%x)
 m = rebinning_weights (d%variance)
 s%ng = d%ng

```

s%num_div = ubound (d%x, dim=1)
s%stratified = d%stratified
s%x_min = d%x_min
s%x_max = d%x_max
s%x_min_true = d%x_min_true
s%x_max_true = d%x_max_true
s%spread_f_p = value_spread_percent (d%integral)
s%stddev_f_p = standard_deviation_percent (d%integral)
s%spread_p = value_spread_percent (p)
s%stddev_p = standard_deviation_percent (p)
s%spread_m = value_spread_percent (m)
s%stddev_m = standard_deviation_percent (m)
deallocate (p, m)
end function summarize_division

58a  <Declaration of divisions procedures 38a>+≡
      private :: probabilities

58b  <Implementation of divisions procedures 38b>+≡
      pure function probabilities (x) result (p)
        real(kind=default), dimension(0:), intent(in) :: x
        real(kind=default), dimension(ubound(x,dim=1)) :: p
        integer :: num_div
        num_div = ubound (x, dim=1)
        p = 1.0 / (x(1:num_div) - x(0:num_div-1))
        p = p / sum(p)
      end function probabilities

58c  <Implementation of divisions procedures 38b>+≡
      subroutine print_history (h, tag)
        type(div_history), dimension(:), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        call write_history (output_unit, h, tag)
        flush (output_unit)
      end subroutine print_history

58d  <Implementation of divisions procedures 38b>+≡
      subroutine write_history (u, h, tag)
        integer, intent(in) :: u
        type(div_history), dimension(:), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        character(len=BUFFER_SIZE) :: pfx
        character(len=1) :: s
        integer :: i
        if (present (tag)) then
          pfx = tag

```

```

else
  pfx = "[vamp]"
end if
if ((minval (h%x_min) == maxval (h%x_min)) &
    .and. (minval (h%x_max) == maxval (h%x_max))) then
  write (u, "(1X,A11,1X,2X,1X,2(E10.3,A4,E10.3,A7))") pfx, &
    h(1)%x_min, " <= ", h(1)%x_min_true, &
    " < x < ", h(1)%x_max_true, " <= ", h(1)%x_max
else
  do i = 1, size (h)
    write (u, "(1X,A11,1X,I2,1X,2(E10.3,A4,E10.3,A7))") pfx, &
      i, h(i)%x_min, " <= ", h(i)%x_min_true, &
      " < x < ", h(i)%x_max_true, " <= ", h(i)%x_max
  end do
end if
write (u, "(1X,A11,1X,A2,2(1X,A3),A1,6(1X,A8))") pfx, &
  "it", "nd", "ng", "", &
  "spr(f/p)", "dev(f/p)", "spr(m)", "dev(m)", "spr(p)", "dev(p)"
iterations: do i = 1, size (h)
  if (h(i)%stratified) then
    s = "*"
  else
    s = ""
  end if
  write (u, "(1X,A11,1X,I2,2(1X,I3),A1,6(1X,F7.2,A1))") pfx, &
    i, h(i)%num_div, h(i)%ng, s, &
    h(i)%spread_f_p, "%", h(i)%stddev_f_p, "%", &
    h(i)%spread_m, "%", h(i)%stddev_m, "%", &
    h(i)%spread_p, "%", h(i)%stddev_p, "%"
end do iterations
flush (u)
end subroutine write_history

```

59a *<Variables in divisions 45d>+≡*
integer, private, parameter :: BUFFER_SIZE = 50

59b *<Declaration of divisions procedures 38a>+≡*
public :: print_history, write_history

59c *<Declaration of divisions procedures (removed from WHIZARD) 59c>≡*
public :: division_x, division_integral
public :: division_variance, division_efficiency

59d *<Implementation of divisions procedures (removed from WHIZARD) 44e>+≡*
pure subroutine division_x (x, d)
real(kind=default), dimension(:), pointer :: x

```

    type(division), intent(in) :: d
    call copy_array_pointer (x, d%x, 0)
end subroutine division_x

```

60a *⟨Implementation of divisions procedures (removed from WHIZARD) 44e⟩*+≡

```

pure subroutine division_integral (integral, d)
  real(kind=default), dimension(:), pointer :: integral
  type(division), intent(in) :: d
  call copy_array_pointer (integral, d%integral)
end subroutine division_integral

```

60b *⟨Implementation of divisions procedures (removed from WHIZARD) 44e⟩*+≡

```

pure subroutine division_variance (variance, d)
  real(kind=default), dimension(:), pointer :: variance
  type(division), intent(in) :: d
  call copy_array_pointer (variance, d%variance, 0)
end subroutine division_variance

```

60c *⟨Implementation of divisions procedures (removed from WHIZARD) 44e⟩*+≡

```

pure subroutine division_efficiency (eff, d)
  real(kind=default), dimension(:), pointer :: eff
  type(division), intent(in) :: d
  call copy_array_pointer (eff, d%efficiency, 0)
end subroutine division_efficiency

```

5.1.8 I/O

60d *⟨Declaration of divisions procedures 38a⟩*+≡

```

public :: write_division
private :: write_division_unit, write_division_name
public :: read_division
private :: read_division_unit, read_division_name
public :: write_division_raw
private :: write_division_raw_unit, write_division_raw_name
public :: read_division_raw
private :: read_division_raw_unit, read_division_raw_name

```

60e *⟨Interfaces of divisions procedures 60e⟩*≡

```

interface write_division
  module procedure write_division_unit, write_division_name
end interface
interface read_division
  module procedure read_division_unit, read_division_name
end interface
interface write_division_raw

```

```

        module procedure write_division_raw_unit, write_division_raw_name
    end interface
    interface read_division_raw
        module procedure read_division_raw_unit, read_division_raw_name
    end interface

```

It makes no sense to read or write `d%integral`, `d%variance`, and `d%efficiency`, because they are only used during sampling.

61a *⟨Implementation of divisions procedures 38b⟩*+≡

```

subroutine write_division_unit (d, unit, write_integrals)
    type(division), intent(in) :: d
    integer, intent(in) :: unit
    logical, intent(in), optional :: write_integrals
    logical :: write_integrals0
    integer :: i
    write_integrals0 = .false.
    if (present(write_integrals)) write_integrals0 = write_integrals
    write (unit = unit, fmt = descr_fmt) "begin type(division) :: d"
    write (unit = unit, fmt = integer_fmt) "ubound(d%x,1) = ", ubound (d%x, dim=1)
    write (unit = unit, fmt = integer_fmt) "d%ng = ", d%ng
    write (unit = unit, fmt = logical_fmt) "d%stratified = ", d%stratified
    write (unit = unit, fmt = double_fmt) "d%dx = ", d%dx
    write (unit = unit, fmt = double_fmt) "d%dxg = ", d%dxg
    write (unit = unit, fmt = double_fmt) "d%x_min = ", d%x_min
    write (unit = unit, fmt = double_fmt) "d%x_max = ", d%x_max
    write (unit = unit, fmt = double_fmt) "d%x_min_true = ", d%x_min_true
    write (unit = unit, fmt = double_fmt) "d%x_max_true = ", d%x_max_true
    write (unit = unit, fmt = descr_fmt) "begin d%x"
    do i = 0, ubound (d%x, dim=1)
        if (write_integrals0 .and. i/=0) then
            write (unit = unit, fmt = double_array_fmt) &
                i, d%x(i), d%integral(i), d%variance(i)
        else
            write (unit = unit, fmt = double_array_fmt) i, d%x(i)
        end if
    end do
    write (unit = unit, fmt = descr_fmt) "end d%x"
    write (unit = unit, fmt = descr_fmt) "end type(division)"
end subroutine write_division_unit

```

61b *⟨Variables in divisions 45d⟩*+≡

```

character(len=*), parameter, private :: &
    descr_fmt =          "(1x,a)", &
    integer_fmt =        "(1x,a15,1x,i15)", &

```

```

logical_fmt =      "(1x,a15,1x,l1)", &
double_fmt =      "(1x,a15,1x,e30.22)", &
double_array_fmt = "(1x,i15,1x,3(e30.22))"

```

62 *<Implementation of divisions procedures 38b>+≡*

```

subroutine read_division_unit (d, unit, read_integrals)
  type(division), intent(inout) :: d
  integer, intent(in) :: unit
  logical, intent(in), optional :: read_integrals
  logical :: read_integrals0
  integer :: i, idum, num_div
  character(len=80) :: chdum
  read_integrals0 = .false.
  if (present(read_integrals)) read_integrals0 = read_integrals
  read (unit = unit, fmt = descr_fmt) chdum
  read (unit = unit, fmt = integer_fmt) chdum, num_div
  <Insure that ubound (d%x, dim=1) == num_div 63a>
  read (unit = unit, fmt = integer_fmt) chdum, d%ng
  read (unit = unit, fmt = logical_fmt) chdum, d%stratified
  read (unit = unit, fmt = double_fmt) chdum, d%dx
  read (unit = unit, fmt = double_fmt) chdum, d%dxg
  read (unit = unit, fmt = double_fmt) chdum, d%x_min
  read (unit = unit, fmt = double_fmt) chdum, d%x_max
  read (unit = unit, fmt = double_fmt) chdum, d%x_min_true
  read (unit = unit, fmt = double_fmt) chdum, d%x_max_true
  read (unit = unit, fmt = descr_fmt) chdum
  do i = 0, ubound (d%x, dim=1)
    if (read_integrals0 .and. i/=0) then
      read (unit = unit, fmt = double_array_fmt) &
        & idum, d%x(i), d%integral(i), d%variance(i)
    else
      read (unit = unit, fmt = double_array_fmt) idum, d%x(i)
    end if
  end do
  read (unit = unit, fmt = descr_fmt) chdum
  read (unit = unit, fmt = descr_fmt) chdum
  if (.not.read_integrals0) then
    d%integral = 0.0
    d%variance = 0.0
  !   d%efficiency = 0.0
  end if
end subroutine read_division_unit

```



What happened to d%efficiency?

```

63a  <Insure that ubound (d%x, dim=1) == num_div 63a>≡
      if (associated (d%x)) then
        if (ubound (d%x, dim=1) /= num_div) then
          deallocate (d%x, d%integral, d%variance)
!       deallocate (d%efficiency)
          allocate (d%x(0:num_div), d%integral(num_div), d%variance(num_div))
!       allocate (d%efficiency(num_div))
        end if
      else
        allocate (d%x(0:num_div), d%integral(num_div), d%variance(num_div))
!       allocate (d%efficiency(num_div))
      end if

63b  <Implementation of divisions procedures 38b>+≡
      subroutine write_division_name (d, name, write_integrals)
        type(division), intent(in) :: d
        character(len=*), intent(in) :: name
        logical, intent(in), optional :: write_integrals
        integer :: unit
        call find_free_unit (unit)
        open (unit = unit, action = "write", status = "replace", file = name)
        call write_division_unit (d, unit, write_integrals)
        close (unit = unit)
      end subroutine write_division_name

63c  <Implementation of divisions procedures 38b>+≡
      subroutine read_division_name (d, name, read_integrals)
        type(division), intent(inout) :: d
        character(len=*), intent(in) :: name
        logical, intent(in), optional :: read_integrals
        integer :: unit
        call find_free_unit (unit)
        open (unit = unit, action = "read", status = "old", file = name)
        call read_division_unit (d, unit, read_integrals)
        close (unit = unit)
      end subroutine read_division_name

63d  <Implementation of divisions procedures 38b>+≡
      subroutine write_division_raw_unit (d, unit, write_integrals)
        type(division), intent(in) :: d
        integer, intent(in) :: unit
        logical, intent(in), optional :: write_integrals
        logical :: write_integrals0
        integer :: i
        write_integrals0 = .false.

```



```

if (present(write_integrals)) write_integrals0 = write_integrals
write (unit = unit) MAGIC_DIVISION_BEGIN
write (unit = unit) ubound (d%x, dim=1)
write (unit = unit) d%ng
write (unit = unit) d%stratified
write (unit = unit) d%dx
write (unit = unit) d%dxg
write (unit = unit) d%x_min
write (unit = unit) d%x_max
write (unit = unit) d%x_min_true
write (unit = unit) d%x_max_true
do i = 0, ubound (d%x, dim=1)
  if (write_integrals0 .and. i/=0) then
    write (unit = unit) d%x(i), d%integral(i), d%variance(i)
  else
    write (unit = unit) d%x(i)
  end if
end do
write (unit = unit) MAGIC_DIVISION_END
end subroutine write_division_raw_unit

```

64a *<Constants in divisions 64a>*≡

```

integer, parameter, private :: MAGIC_DIVISION = 11111111
integer, parameter, private :: MAGIC_DIVISION_BEGIN = MAGIC_DIVISION + 1
integer, parameter, private :: MAGIC_DIVISION_END = MAGIC_DIVISION + 2

```

64b *<Implementation of divisions procedures 38b>*+≡

```

subroutine read_division_raw_unit (d, unit, read_integrals)
  type(division), intent(inout) :: d
  integer, intent(in) :: unit
  logical, intent(in), optional :: read_integrals
  logical :: read_integrals0
  integer :: i, num_div, magic
  character(len=*), parameter :: FN = "read_division_raw_unit"
  read_integrals0 = .false.
  if (present(read_integrals)) read_integrals0 = read_integrals
  read (unit = unit) magic
  if (magic /= MAGIC_DIVISION_BEGIN) then
    print *, FN, " fatal: expecting magic ", MAGIC_DIVISION_BEGIN, &
      ", found ", magic
    stop
  end if
  read (unit = unit) num_div
<Insure that ubound (d%x, dim=1) == num_div 63a>
  read (unit = unit) d%ng

```

```

read (unit = unit) d%stratified
read (unit = unit) d%dx
read (unit = unit) d%dxg
read (unit = unit) d%x_min
read (unit = unit) d%x_max
read (unit = unit) d%x_min_true
read (unit = unit) d%x_max_true
do i = 0, ubound (d%x, dim=1)
  if (read_integrals0 .and. i/=0) then
    read (unit = unit) d%x(i), d%integral(i), d%variance(i)
  else
    read (unit = unit) d%x(i)
  end if
end do
if (.not.read_integrals0) then
  d%integral = 0.0
  d%variance = 0.0
!   d%efficiency = 0.0
end if
read (unit = unit) magic
if (magic /= MAGIC_DIVISION_END) then
  print *, FN, " fatal: expecting magic ", MAGIC_DIVISION_END, &
    ", found ", magic
  stop
end if
end subroutine read_division_raw_unit

```

65a *⟨Implementation of divisions procedures 38b⟩*+≡

```

subroutine write_division_raw_name (d, name, write_integrals)
  type(division), intent(in) :: d
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: write_integrals
  integer :: unit
  call find_free_unit (unit)
  open (unit = unit, action = "write", status = "replace", &
    form = "unformatted", file = name)
  call write_division_unit (d, unit, write_integrals)
  close (unit = unit)
end subroutine write_division_raw_name

```

65b *⟨Implementation of divisions procedures 38b⟩*+≡

```

subroutine read_division_raw_name (d, name, read_integrals)
  type(division), intent(inout) :: d
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: read_integrals

```

```

integer :: unit
call find_free_unit (unit)
open (unit = unit, action = "read", status = "old", &
      form = "unformatted", file = name)
call read_division_unit (d, unit, read_integrals)
close (unit = unit)
end subroutine read_division_raw_name

```

5.1.9 Marshaling

Note that we can not use the `transfer` intrinsic function for marshalling types that contain pointers that substitute for allocatable array components. `transfer` will copy the pointers in this case and not where they point to!

66a *<Declaration of divisions procedures 38a>+≡*

```

public :: marshal_division_size, marshal_division, unmarshal_division

```

66b *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine marshal_division (d, ibuf, dbuf)
  type(division), intent(in) :: d
  integer, dimension(:), intent(inout) :: ibuf
  real(kind=default), dimension(:), intent(inout) :: dbuf
  integer :: num_div
  num_div = ubound (d%x, dim=1)
  ibuf(1) = d%ng
  ibuf(2) = num_div
  if (d%stratified) then
    ibuf(3) = 1
  else
    ibuf(3) = 0
  end if
  dbuf(1) = d%x_min
  dbuf(2) = d%x_max
  dbuf(3) = d%x_min_true
  dbuf(4) = d%x_max_true
  dbuf(5) = d%dx
  dbuf(6) = d%dxg
  dbuf(7:7+num_div) = d%x
  dbuf(8+ num_div:7+2*num_div) = d%integral
  dbuf(8+2*num_div:7+3*num_div) = d%variance
  ! dbuf(8+3*num_div:7+4*num_div) = d%efficiency
end subroutine marshal_division

```

66c *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine marshal_division_size (d, iwords, dwords)

```

```

type(division), intent(in) :: d
integer, intent(out) :: iwords, dwords
iwords = 3
dwords = 7 + 3 * ubound (d%x, dim=1)
! dwords = 7 + 4 * ubound (d%x, dim=1)
end subroutine marshal_division_size

```

67a *⟨Implementation of divisions procedures 38b⟩+≡*

```

pure subroutine unmarshal_division (d, ibuf, dbuf)
  type(division), intent(inout) :: d
  integer, dimension(:), intent(in) :: ibuf
  real(kind=default), dimension(:), intent(in) :: dbuf
  integer :: num_div
  d%ng = ibuf(1)
  num_div = ibuf(2)
  d%stratified = ibuf(3) /= 0
  d%x_min = dbuf(1)
  d%x_max = dbuf(2)
  d%x_min_true = dbuf(3)
  d%x_max_true = dbuf(4)
  d%dx = dbuf(5)
  d%dxg = dbuf(6)
  ⟨Insure that ubound (d%x, dim=1) == num_div 63a⟩
  d%x = dbuf(7:7+num_div)
  d%integral = dbuf(8+ num_div:7+2*num_div)
  d%variance = dbuf(8+2*num_div:7+3*num_div)
  ! d%efficiency = dbuf(8+3*num_div:7+4*num_div)
end subroutine unmarshal_division

```

67b *⟨Declaration of divisions procedures 38a⟩+≡*

```

public :: marshal_div_history_size, marshal_div_history, unmarshal_div_history

```

67c *⟨Implementation of divisions procedures 38b⟩+≡*

```

pure subroutine marshal_div_history (h, ibuf, dbuf)
  type(div_history), intent(in) :: h
  integer, dimension(:), intent(inout) :: ibuf
  real(kind=default), dimension(:), intent(inout) :: dbuf
  ibuf(1) = h%ng
  ibuf(2) = h%num_div
  if (h%stratified) then
    ibuf(3) = 1
  else
    ibuf(3) = 0
  end if
  dbuf(1) = h%x_min

```

```

    dbuf(2) = h%x_max
    dbuf(3) = h%x_min_true
    dbuf(4) = h%x_max_true
    dbuf(5) = h%spread_f_p
    dbuf(6) = h%stddev_f_p
    dbuf(7) = h%spread_p
    dbuf(8) = h%stddev_p
    dbuf(9) = h%spread_m
    dbuf(10) = h%stddev_m
end subroutine marshal_div_history
68a  <Implementation of divisions procedures 38b>+≡
    pure subroutine marshal_div_history_size (h, iwords, dwords)
        type(div_history), intent(in) :: h
        integer, intent(out) :: iwords, dwords
        iwords = 3
        dwords = 10
    end subroutine marshal_div_history_size
68b  <Implementation of divisions procedures 38b>+≡
    pure subroutine unmarshal_div_history (h, ibuf, dbuf)
        type(div_history), intent(inout) :: h
        integer, dimension(:), intent(in) :: ibuf
        real(kind=default), dimension(:), intent(in) :: dbuf
        h%ng = ibuf(1)
        h%num_div = ibuf(2)
        h%stratified = ibuf(3) /= 0
        h%x_min = dbuf(1)
        h%x_max = dbuf(2)
        h%x_min_true = dbuf(3)
        h%x_max_true = dbuf(4)
        h%spread_f_p = dbuf(5)
        h%stddev_f_p = dbuf(6)
        h%spread_p = dbuf(7)
        h%stddev_p = dbuf(8)
        h%spread_m = dbuf(9)
        h%stddev_m = dbuf(10)
    end subroutine unmarshal_div_history

```

5.1.10 Boring Copying and Deleting of Objects

```

68c  <Implementation of divisions procedures 38b>+≡
    elemental subroutine copy_division (lhs, rhs)
        type(division), intent(inout) :: lhs

```

```

type(division), intent(in) :: rhs
if (associated (rhs%x)) then
    call copy_array_pointer (lhs%x, rhs%x, lb = 0)
else if (associated (lhs%x)) then
    deallocate (lhs%x)
end if
if (associated (rhs%integral)) then
    call copy_array_pointer (lhs%integral, rhs%integral)
else if (associated (lhs%integral)) then
    deallocate (lhs%integral)
end if
if (associated (rhs%variance)) then
    call copy_array_pointer (lhs%variance, rhs%variance)
else if (associated (lhs%variance)) then
    deallocate (lhs%variance)
end if
! if (associated (rhs%efficiency)) then
!     call copy_array_pointer (lhs%efficiency, rhs%efficiency)
! else if (associated (lhs%efficiency)) then
!     deallocate (lhs%efficiency)
! end if
lhs%dx = rhs%dx
lhs%dxg = rhs%dxg
lhs%x_min = rhs%x_min
lhs%x_max = rhs%x_max
lhs%x_min_true = rhs%x_min_true
lhs%x_max_true = rhs%x_max_true
lhs%ng = rhs%ng
lhs%stratified = rhs%stratified
end subroutine copy_division

```

69a *<Implementation of divisions procedures 38b>+≡*
 elemental subroutine delete_division (d)
 type(division), intent(inout) :: d
 if (associated (d%x)) then
 deallocate (d%x, d%integral, d%variance)
 ! deallocate (d%efficiency)
 end if
end subroutine delete_division

69b *<Implementation of divisions procedures 38b>+≡*
 elemental subroutine copy_history (lhs, rhs)
 type(div_history), intent(out) :: lhs
 type(div_history), intent(in) :: rhs


```

lhs%stratified = rhs%stratified
lhs%ng = rhs%ng
lhs%num_div = rhs%num_div
lhs%x_min = rhs%x_min
lhs%x_max = rhs%x_max
lhs%x_min_true = rhs%x_min_true
lhs%x_max_true = rhs%x_max_true
lhs%spread_f_p = rhs%spread_f_p
lhs%stddev_f_p = rhs%stddev_f_p
lhs%spread_p = rhs%spread_p
lhs%stddev_p = rhs%stddev_p
lhs%spread_m = rhs%spread_m
lhs%stddev_m = rhs%stddev_m
end subroutine copy_history

```


5.2 The Abstract Datatype *vamp_grid*

70a $\langle \text{vamp.f90 70a} \rangle \equiv$
`! vamp.f90 --`
 $\langle \text{Copyleft notice 1} \rangle$

 NAG f95 requires this split. Check with the Fortran community, if it is really necessary, or a bug! The problem is that this split forces us to expose the components of `vamp_grid`.

NB: with the introduction of `vamp_equivalences`, this question has (probably) become academic.

70b $\langle \text{vamp.f90 70a} \rangle + \equiv$
`module vamp_grid_type`
`use kinds`
`use divisions`
`private`
 $\langle \text{Declaration of vamp_grid_type types 76a} \rangle$
`end module vamp_grid_type`

 By WK for WHIZARD.

70c $\langle \text{vamp.f90 70a} \rangle + \equiv$
`module vamp_equivalences`
`use kinds`
`use divisions`

```

    use vamp_grid_type !NODEP!
    implicit none
    private
    <Declaration of vamp_equivalences procedures 71e>
    <Constants in vamp_equivalences 71c>
    <Declaration of vamp_equivalences types 71a>
    character(len=*), public, parameter :: VAMP_EQUIVALENCES_RCS_ID = &
        "$Id: vamp.nw 317 2010-04-18 00:31:03Z ohl $"
    contains
    <Implementation of vamp_equivalences procedures 71d>
    end module vamp_equivalences

71a <Declaration of vamp_equivalences types 71a>≡
    type, public :: vamp_equivalence_t
        integer :: left, right
        integer, dimension(:), allocatable :: permutation
        integer, dimension(:), allocatable :: mode
    end type vamp_equivalence_t

71b <Declaration of vamp_equivalences types 71a>+≡
    type, public :: vamp_equivalences_t
        type(vamp_equivalence_t), dimension(:), allocatable :: eq
        integer :: n_eq, n_ch
        integer, dimension(:), allocatable :: pointer
        logical, dimension(:), allocatable :: independent
        integer, dimension(:), allocatable :: equivalent_to_ch
        integer, dimension(:), allocatable :: multiplicity
        integer, dimension(:), allocatable :: symmetry
        logical, dimension(:, :), allocatable :: div_is_invariant
    end type vamp_equivalences_t

71c <Constants in vamp_equivalences 71c>≡
    integer, parameter, public :: &
        VEQ_IDENTITY = 0, VEQ_INVERT = 1, VEQ_SYMMETRIC = 2, VEQ_INVARIANT = 3

71d <Implementation of vamp_equivalences procedures 71d>≡
    subroutine vamp_equivalence_init (eq, n_dim)
        type(vamp_equivalence_t), intent(inout) :: eq
        integer, intent(in) :: n_dim
        allocate (eq%permutation(n_dim), eq%mode(n_dim))
    end subroutine vamp_equivalence_init

71e <Declaration of vamp_equivalences procedures 71e>≡
    public :: vamp_equivalences_init

```


72a *<Implementation of vamp_equivalences procedures 71d>+≡*

```
subroutine vamp_equivalences_init (eq, n_eq, n_ch, n_dim)
  type(vamp_equivalences_t), intent(inout) :: eq
  integer, intent(in) :: n_eq, n_ch, n_dim
  integer :: i
  eq%n_eq = n_eq
  eq%n_ch = n_ch
  allocate (eq%eq(n_eq))
  allocate (eq%pointer(n_ch+1))
  do i=1, n_eq
    call vamp_equivalence_init (eq%eq(i), n_dim)
  end do
  allocate (eq%independent(n_ch), eq%equivalent_to_ch(n_ch))
  allocate (eq%multiplicity(n_ch), eq%symmetry(n_ch))
  allocate (eq%div_is_invariant(n_ch, n_dim))
  eq%independent = .true.
  eq%equivalent_to_ch = 0
  eq%multiplicity = 0
  eq%symmetry = 0
  eq%div_is_invariant = .false.
end subroutine vamp_equivalences_init
```

72b *<Implementation of vamp_equivalences procedures 71d>+≡*

```
subroutine vamp_equivalence_final (eq)
  type(vamp_equivalence_t), intent(inout) :: eq
  deallocate (eq%permutation, eq%mode)
end subroutine vamp_equivalence_final
```

72c *<Declaration of vamp_equivalences procedures 71e>+≡*

```
public :: vamp_equivalences_final
```

72d *<Implementation of vamp_equivalences procedures 71d>+≡*

```
subroutine vamp_equivalences_final (eq)
  type(vamp_equivalences_t), intent(inout) :: eq
  ! integer :: i
  ! do i=1, eq%n_eq
  !   call vamp_equivalence_final (eq%eq(i))
  ! end do
  if (allocated (eq%eq)) deallocate (eq%eq)
  if (allocated (eq%pointer)) deallocate (eq%pointer)
  if (allocated (eq%multiplicity)) deallocate (eq%multiplicity)
  if (allocated (eq%symmetry)) deallocate (eq%symmetry)
  if (allocated (eq%independent)) deallocate (eq%independent)
  if (allocated (eq%equivalent_to_ch)) deallocate (eq%equivalent_to_ch)
  if (allocated (eq%div_is_invariant)) deallocate (eq%div_is_invariant)
```

```

        eq%n_eq = 0
        eq%n_ch = 0
    end subroutine vamp_equivalences_final

73a  <Implementation of vamp_equivalences procedures 71d>+≡
    subroutine vamp_equivalence_write (eq, unit)
        integer, intent(in), optional :: unit
        integer :: u
        type(vamp_equivalence_t), intent(in) :: eq
        u = 6; if (present (unit)) u = unit
        write (u, "(1x,A,2(1x,I4))") "Equivalent channels:", eq%left, eq%right
        write (u, "(1x,A,25(1x,I2))") " Permutation:", eq%permutation
        write (u, "(1x,A,25(1x,I2))") " Mode:          ", eq%mode
    end subroutine vamp_equivalence_write

73b  <Declaration of vamp_equivalences procedures 71e>+≡
    public :: vamp_equivalences_write

73c  <Implementation of vamp_equivalences procedures 71d>+≡
    subroutine vamp_equivalences_write (eq, unit)
        type(vamp_equivalences_t), intent(in) :: eq
        integer, intent(in), optional :: unit
        integer :: u
        integer :: ch, i
        u = 6; if (present (unit)) u = unit
        write (u, *) "Inequivalent channels:"
        do ch=1, eq%n_ch
            if (eq%independent(ch)) then
                write (u, *) " Channel", ch, ":", &
                    & " Mult. =", eq%multiplicity(ch), &
                    & " Symm. =", eq%symmetry(ch), &
                    & " Invar.:", eq%div_is_invariant(ch,:)
            end if
        end do
        write (u, *) "Equivalence list:"
        if (allocated (eq%eq)) then
            do i=1, size (eq%eq)
                call vamp_equivalence_write (eq%eq(i), u)
            end do
        else
            write (u, *) "[not allocated]"
        end if
    end subroutine vamp_equivalences_write

73d  <Declaration of vamp_equivalences procedures 71e>+≡
    public :: vamp_equivalence_set

```

```

74a  <Implementation of vamp_equivalences procedures 71d>+≡
      subroutine vamp_equivalence_set (eq, i, left, right, perm, mode)
        type(vamp_equivalences_t), intent(inout) :: eq
        integer, intent(in) :: i
        integer, intent(in) :: left, right
        integer, dimension(:), intent(in) :: perm, mode
        eq%eq(i)%left = left
        eq%eq(i)%right = right
        eq%eq(i)%permutation = perm
        eq%eq(i)%mode = mode
      end subroutine vamp_equivalence_set

74b  <Declaration of vamp_equivalences procedures 71e>+≡
      public :: vamp_equivalences_complete

74c  <Implementation of vamp_equivalences procedures 71d>+≡
      subroutine vamp_equivalences_complete (eq)
        type(vamp_equivalences_t), intent(inout) :: eq
        integer :: i, ch
        ch = 0
        do i=1, eq%n_eq
          if (ch /= eq%eq(i)%left) then
            ch = eq%eq(i)%left
            eq%pointer(ch) = i
          end if
        end do
        eq%pointer(ch+1) = eq%n_eq + 1
        do ch=1, eq%n_ch
          call set_multiplicities (eq%eq(eq%pointer(ch):eq%pointer(ch+1)-1))
        end do
        ! call write (6, eq)
      contains
        subroutine set_multiplicities (eq_ch)
          type(vamp_equivalence_t), dimension(:), intent(in) :: eq_ch
          integer :: i
          if (.not. all(eq_ch%left == ch) .or. eq_ch(1)%right > ch) then
            do i = 1, size (eq_ch)
              call vamp_equivalence_write (eq_ch(i))
            end do
            stop "VAMP: Equivalences: Something's wrong with equivalence ordering"
          end if
          eq%symmetry(ch) = count (eq_ch%right == ch)
          if (mod (size(eq_ch), eq%symmetry(ch)) /= 0) then
            do i = 1, size (eq_ch)
              call vamp_equivalence_write (eq_ch(i))
            end do
          end if
        end subroutine set_multiplicities
      end subroutine vamp_equivalences_complete

```

```

        end do
        stop "VAMP: Equivalences: Something's wrong with permutation count"
    end if
    eq%multiplicity(ch) = size (eq_ch) / eq%symmetry(ch)
    eq%independent(ch) = all (eq_ch%right >= ch)
    eq%equivalent_to_ch(ch) = eq_ch(1)%right
    eq%div_is_invariant(ch,:) = eq_ch(1)%mode == VEQ_INVARIANT
end subroutine set_multiplicities
end subroutine vamp_equivalences_complete

75a <vamp.f90 70a>+≡
module vamp_rest
    use kinds
    use utils
    use exceptions
    use divisions
    use tao_random_numbers
    use vamp_stat
    use linalg
    use iso_fortran_env
    use vamp_grid_type !NODEP!
    use vamp_equivalences !NODEP!
    implicit none
    private
    <Declaration of vamp procedures 76b>
    <Interfaces of vamp procedures 93b>
    <Constants in vamp 149a>
    <Declaration of vamp types 103c>
    <Variables in vamp 78a>
    character(len=*), public, parameter :: VAMP_RCS_ID = &
        "$Id: vamp.nw 317 2010-04-18 00:31:03Z ohl $"
contains
    <Implementation of vamp procedures 76d>
end module vamp_rest

75b <vamp.f90 70a>+≡
module vamp
    use vamp_grid_type    !NODEP!
    use vamp_rest         !NODEP!
    use vamp_equivalences !NODEP!
    public
end module vamp

```

N.B.: In Fortran95 we will be able to give default initializations to components of the type. In particular, we can use the `null ()` intrinsic to initialize

the pointers to a disassociated state. Until then, the user *must* call the initializer `vamp_create_grid` himself of herself, because we can't check for the allocation status of the pointers in Fortran90 or F.

76a \langle Declaration of `vamp_grid_type` types **76a** $\rangle \equiv$

```

type, public :: vamp_grid
  ! private ! forced by use association in interface
  type(dimension), dimension(:), pointer :: div => null ()
  real(kind=default), dimension(:,:), pointer :: map => null ()
  real(kind=default), dimension(:), pointer :: mu_x => null ()
  real(kind=default), dimension(:), pointer :: sum_mu_x => null ()
  real(kind=default), dimension(:,:), pointer :: mu_xx => null ()
  real(kind=default), dimension(:,:), pointer :: sum_mu_xx => null ()
  real(kind=default), dimension(2) :: mu
  real(kind=default) :: sum_integral, sum_weights, sum_chi2
  real(kind=default) :: calls, dv2g, jacobi
  real(kind=default) :: f_min, f_max
  real(kind=default) :: mu_gi, sum_mu_gi
  integer, dimension(:), pointer :: num_div => null ()
  integer :: num_calls, calls_per_cell
  logical :: stratified = .true.
  logical :: all_stratified = .true.
  logical :: quadrupole = .false.
  logical :: independent
  integer :: equivalent_to_ch, multiplicity
end type vamp_grid

```

76b \langle Declaration of `vamp` procedures **76b** $\rangle \equiv$

```

public :: vamp_copy_grid, vamp_delete_grid

```

5.2.1 Initialization

76c \langle Declaration of `vamp` procedures **76b** $\rangle + \equiv$

```

public :: vamp_create_grid, vamp_create_empty_grid

```

Create a fresh grid for the integration domain

$$\mathcal{D} = [D_{1,1}, D_{2,1}] \times [D_{1,2}, D_{2,2}] \times \dots \times [D_{1,n}, D_{2,n}] \quad (5.9)$$

dropping all accumulated results. This function *must not* be called twice on the first argument, without an intervening `vamp_delete_grid`. Iff the second variable is given, it will be the number of sampling points for the call to `vamp_sample_grid`.

76d \langle Implementation of `vamp` procedures **76d** $\rangle \equiv$

```

pure subroutine vamp_create_grid &

```

```

    (g, domain, num_calls, num_div, &
      stratified, quadrupole, covariance, map, exc)
type(vamp_grid), intent(inout) :: g
real(kind=default), dimension(:,:), intent(in) :: domain
integer, intent(in) :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole, covariance
real(kind=default), dimension(:,:), intent(in), optional :: map
type(exception), intent(inout), optional :: exc
character(len=*), parameter :: FN = "vamp_create_grid"
real(kind=default), dimension(size(domain,dim=2)) :: &
    x_min, x_max, x_min_true, x_max_true
integer :: ndim
ndim = size (domain, dim=2)
allocate (g%div(ndim), g%num_div(ndim))
x_min = domain(1,:)
x_max = domain(2,:)
if (present (map)) then
    allocate (g%map(ndim,ndim))
    g%map = map
    x_min_true = x_min
    x_max_true = x_max
    call map_domain (g%map, x_min_true, x_max_true, x_min, x_max)
    call create_division (g%div, x_min, x_max, x_min_true, x_max_true)
else
    nullify (g%map)
    call create_division (g%div, x_min, x_max)
end if
g%num_calls = num_calls
if (present (num_div)) then
    g%num_div = num_div
else
    g%num_div = NUM_DIV_DEFAULT
end if
g%stratified = .true.
g%quadrupole = .false.
g%independent = .true.
g%equivalent_to_ch = 0
g%multiplicity = 1
nullify (g%mu_x, g%mu_xx, g%sum_mu_x, g%sum_mu_xx)
call vamp_discard_integral &
    (g, num_calls, num_div, stratified, quadrupole, covariance, exc)
end subroutine vamp_create_grid

```

Below, we assume that $\text{NUM_DIV_DEFAULT} \geq 6$, but we will never go that low anyway.

78a $\langle \text{Variables in vamp 78a} \rangle \equiv$
`integer, private, parameter :: NUM_DIV_DEFAULT = 20`
 Given a linear map M , find a domain \mathcal{D}_0 such that

$$\mathcal{D} \subset M\mathcal{D}_0 \quad (5.10)$$

78b $\langle \text{Declaration of vamp procedures 76b} \rangle + \equiv$
`private :: map_domain`

If we can assume that M is orthogonal $M^{-1} = M^T$, then we just have to rotate \mathcal{D} and determine the maximal and minimal extension of the corners:

$$\mathcal{D}_0^T = \overline{\mathcal{D}^T M} \quad (5.11)$$

The corners are just the powerset of the maximal and minimal extension in each coordinate. It is determined most easily with binary counting:

78c $\langle \text{Implementation of vamp procedures 76d} \rangle + \equiv$

```

pure subroutine map_domain (map, true_xmin, true_xmax, xmin, xmax)
  real(kind=default), dimension(:, :), intent(in) :: map
  real(kind=default), dimension(:), intent(in) :: true_xmin, true_xmax
  real(kind=default), dimension(:), intent(out) :: xmin, xmax
  real(kind=default), dimension(2**size(xmin), size(xmin)) :: corners
  integer, dimension(size(xmin)) :: zero_to_n
  integer :: j, ndim, perm
  ndim = size (xmin)
  zero_to_n = (/ (j, j=0, ndim-1) /)
  do perm = 1, 2**ndim
    corners (perm, :) = &
      merge (true_xmin, true_xmax, btest (perm-1, zero_to_n))
  end do
  corners = matmul (corners, map)
  xmin = minval (corners, dim=1)
  xmax = maxval (corners, dim=1)
end subroutine map_domain

```

78d $\langle \text{Implementation of vamp procedures 76d} \rangle + \equiv$

```

elemental subroutine vamp_create_empty_grid (g)
  type(vamp_grid), intent(inout) :: g
  nullify (g%div, g%num_div, g%map, g%mu_x, g%mu_xx, g%sum_mu_x, g%sum_mu_xx)
end subroutine vamp_create_empty_grid

```

78e $\langle \text{Declaration of vamp procedures 76b} \rangle + \equiv$
`public :: vamp_discard_integral`

Keep the current optimized grid, but drop the accumulated results for the integral (value and errors). If the second variable is given, it will be the new number of sampling points for the next call to `vamp_sample_grid`.

79a *⟨Implementation of vamp procedures 76d⟩*+≡

```
pure subroutine vamp_discard_integral &
  (g, num_calls, num_div, stratified, quadrupole, covariance, exc, &
   & independent, equivalent_to_ch, multiplicity)
type(vamp_grid), intent(inout) :: g
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole, covariance
type(exception), intent(inout), optional :: exc
logical, intent(in), optional :: independent
integer, intent(in), optional :: equivalent_to_ch, multiplicity
character(len=*), parameter :: FN = "vamp_discard_integral"
g%mu = 0.0
g%mu_gi = 0.0
g%sum_integral = 0.0
g%sum_weights = 0.0
g%sum_chi2 = 0.0
g%sum_mu_gi = 0.0
if (associated (g%sum_mu_x)) then
  g%sum_mu_x = 0.0
  g%sum_mu_xx = 0.0
end if
call set_grid_options (g, num_calls, num_div, stratified, quadrupole, &
  independent, equivalent_to_ch, multiplicity)
if ((present (num_calls)) &
  .or. (present (num_div)) &
  .or. (present (stratified)) &
  .or. (present (quadrupole)) &
  .or. (present (covariance))) then
  call vamp_reshape_grid &
    (g, g%num_calls, g%num_div, &
     g%stratified, g%quadrupole, covariance, exc)
end if
end subroutine vamp_discard_integral
```

79b *⟨Declaration of vamp procedures 76b⟩*+≡

```
private :: set_grid_options
```

79c *⟨Implementation of vamp procedures 76d⟩*+≡

```
pure subroutine set_grid_options &
  (g, num_calls, num_div, stratified, quadrupole, &
```



```

        independent, equivalent_to_ch, multiplicity)
type(vamp_grid), intent(inout) :: g
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
logical, intent(in), optional :: independent
integer, intent(in), optional :: equivalent_to_ch, multiplicity
if (present (num_calls)) then
    g%num_calls = num_calls
end if
if (present (num_div)) then
    g%num_div = num_div
end if
if (present (stratified)) then
    g%stratified = stratified
end if
if (present (quadrupole)) then
    g%quadrupole = quadrupole
end if
if (present (independent)) then
    g%independent = independent
end if
if (present (equivalent_to_ch)) then
    g%equivalent_to_ch = equivalent_to_ch
end if
if (present (multiplicity)) then
    g%multiplicity = multiplicity
end if
end subroutine set_grid_options

```

Setting Up the Initial Grid

Keep the current optimized grid and the accumulated results for the integral (value and errors). The second variable will be the new number of sampling points for the next call to `vamp_sample_grid`.

80 *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_reshape_grid_internal &
    (g, num_calls, num_div, &
     stratified, quadrupole, covariance, exc, use_variance, &
     independent, equivalent_to_ch, multiplicity)
type(vamp_grid), intent(inout) :: g
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div

```

```

logical, intent(in), optional :: stratified, quadrupole, covariance
type(exception), intent(inout), optional :: exc
logical, intent(in), optional :: use_variance
logical, intent(in), optional :: independent
integer, intent(in), optional :: equivalent_to_ch, multiplicity
integer :: ndim, num_cells
integer, dimension(size(g%div)) :: ng
character(len=*), parameter :: FN = "vamp_reshape_grid_internal"
ndim = size (g%div)
call set_grid_options &
    (g, num_calls, num_div, stratified, quadrupole, &
    & independent, equivalent_to_ch, multiplicity)
<Adjust grid and other state for new num_calls 82>
g%all_stratified = all (stratified_division (g%div))
if (present (covariance)) then
    ndim = size (g%div)
    if (covariance .and. (.not. associated (g%mu_x))) then
        allocate (g%mu_x(ndim), g%mu_xx(ndim,ndim))
        allocate (g%sum_mu_x(ndim), g%sum_mu_xx(ndim,ndim))
        g%sum_mu_x = 0.0
        g%sum_mu_xx = 0.0
    else if ((.not. covariance) .and. (associated (g%mu_x))) then
        deallocate (g%mu_x, g%mu_xx, g%sum_mu_x, g%sum_mu_xx)
    end if
end if
end subroutine vamp_reshape_grid_internal

```

The `use_variance` argument is too dangerous for careless users, because the `variance` in the divisions will contain garbage before sampling and after reshaping. Build a fence with another routine.

81a *<Declaration of vamp procedures 76b>+≡*

```

private :: vamp_reshape_grid_internal
public :: vamp_reshape_grid

```

81b *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_reshape_grid &
    (g, num_calls, num_div, stratified, quadrupole, covariance, exc, &
    independent, equivalent_to_ch, multiplicity)
type(vamp_grid), intent(inout) :: g
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole, covariance
type(exception), intent(inout), optional :: exc
logical, intent(in), optional :: independent

```

```

integer, intent(in), optional :: equivalent_to_ch, multiplicity
call vamp_reshape_grid_internal &
    (g, num_calls, num_div, stratified, quadrupole, covariance, &
    exc, use_variance = .false., &
    independent=independent, equivalent_to_ch=equivalent_to_ch, &
    multiplicity=multiplicity)
end subroutine vamp_reshape_grid

```

vegas operates in three different modes, which are chosen according to explicit user requests and to the relation of the requested number of sampling points to the dimensionality of the integration domain.

The simplest case is when the user has overwritten the default of stratified sampling with the optional argument **stratified** in the call to **vamp_create_grid**. Then sample points will be chosen randomly with equal probability in each cell of the adaptive grid, as displayed in figure 5.1.

The implementation is actually shared with the stratified case described below, by pretending that there is just a single stratification cell. The number of divisions for the adaptive grid is set to a compile time maximum value.

If the user has agreed on stratified sampling then there are two cases, depending on the dimensionality of the integration region and the number of sample points. First we determine the number of divisions n_g (i. e. **ng**) of the rigid grid such that there will be two sampling points per cell.

$$N_{\text{calls}} = 2 \cdot (n_g)^{n_{\text{dim}}} \quad (5.12)$$

The additional optional argument \hat{n}_g specifies an anisotropy in the shape

$$n_{g,j} = \frac{\hat{n}_{g,j}}{\left(\prod_j \hat{n}_{g,j}\right)^{1/n_{\text{dim}}}} \left(\frac{N}{2}\right)^{1/n_{\text{dim}}} \quad (5.13)$$

NB:

$$\prod_j n_{g,j} = \frac{N}{2} \quad (5.14)$$

```

82  <Adjust grid and other state for new num_calls 82>≡
    if (g%stratified) then
        ng = (g%num_calls / 2.0 + 0.25)**(1.0/ndim)
        ! ng = ng * real (g%num_div, kind=default) &
        !       / (product (real (g%num_div, kind=default)))** (1.0/ndim)
    else
        ng = 1
    end if
    call reshape_division (g%div, g%num_div, ng, use_variance)

```

$$\begin{aligned}
& \text{call clear_integral_and_variance (g\%div)} \\
& \text{num_cells} = \text{product (rigid_division (g\%div))} \\
& \text{g\%calls_per_cell} = \text{max (g\%num_calls / num_cells, 2)} \\
& \text{g\%calls} = \text{real (g\%calls_per_cell) * real (num_cells)} \\
& \text{jacobi} = J = \frac{\text{Volume}}{N_{\text{calls}}} \tag{5.15}
\end{aligned}$$

and

$$\text{dv2g} = \frac{N_{\text{calls}}^2 ((\Delta x)^{n_{\text{dim}}})^2}{N_{\text{calls/cell}}^2 (N_{\text{calls/cell}} - 1)} = \frac{\left(\frac{N_{\text{calls}}}{N_{\text{cells}}}\right)^2}{N_{\text{calls/cell}}^2 (N_{\text{calls/cell}} - 1)} \tag{5.16}$$

- 83a *<Adjust grid and other state for new num_calls 82>+≡*
`g%jacobi = product (volume_division (g%div)) / g%calls`
`g%dv2g = (g%calls / num_cells)**2 &`
`/ g%calls_per_cell / g%calls_per_cell / (g%calls_per_cell - 1.0)`
- 83b *<Adjust grid and other state for new num_calls 82>+≡*
`g%f_min = 1.0`
`g%f_max = 0.0`
- 83c *<Declaration of vamp procedures 76b>+≡*
`public :: vamp_rigid_divisions`
`public :: vamp_get_covariance, vamp_nullify_covariance`
`public :: vamp_get_variance, vamp_nullify_variance`
- 83d *<Implementation of vamp procedures 76d>+≡*
`pure function vamp_rigid_divisions (g) result (ng)`
`type(vamp_grid), intent(in) :: g`
`integer, dimension(size(g%div)) :: ng`
`ng = rigid_division (g%div)`
`end function vamp_rigid_divisions`
- 83e *<Implementation of vamp procedures 76d>+≡*
`pure function vamp_get_covariance (g) result (cov)`
`type(vamp_grid), intent(in) :: g`
`real(kind=default), dimension(size(g%div),size(g%div)) :: cov`
`if (associated (g%mu_x)) then`
`if (abs (g%sum_weights) <= tiny (cov(1,1))) then`
`where (g%sum_mu_xx == 0.0_default)`
`cov = 0.0`
`elsewhere`
`cov = huge (cov(1,1))`
`endwhere`
`else`
`cov = g%sum_mu_xx / g%sum_weights &`

```

- outer_product (g%sum_mu_x, g%sum_mu_x) / g%sum_weights**2
    end if
else
    cov = 0.0
end if
end function vamp_get_covariance
84a ⟨Implementation of vamp procedures 76d⟩+≡
    elemental subroutine vamp_nullify_covariance (g)
        type(vamp_grid), intent(inout) :: g
        if (associated (g%mu_x)) then
            g%sum_mu_x = 0
            g%sum_mu_xx = 0
        end if
    end subroutine vamp_nullify_covariance
84b ⟨Implementation of vamp procedures 76d⟩+≡
    elemental function vamp_get_variance (g) result (v)
        type(vamp_grid), intent(in) :: g
        real(kind=default) :: v
        if (abs (g%sum_weights) <= tiny (v)) then
            if (g%sum_mu_gi == 0.0_default) then
                v = 0.0
            else
                v = huge (v)
            end if
        else
            v = g%sum_mu_gi / g%sum_weights
        end if
    end function vamp_get_variance
84c ⟨Implementation of vamp procedures 76d⟩+≡
    elemental subroutine vamp_nullify_variance (g)
        type(vamp_grid), intent(inout) :: g
        g%sum_mu_gi = 0
    end subroutine vamp_nullify_variance

```

5.2.2 Sampling

```

84d ⟨Declaration of vamp procedures 76b⟩+≡
    public :: vamp_sample_grid
    public :: vamp_sample_grid0
    public :: vamp_refine_grid
    public :: vamp_refine_grids

```

Simple Non-Adaptive Sampling: S_0

85a *⟨Implementation of vamp procedures 76d⟩* \equiv

```

pure subroutine vamp_sample_grid0 &
    (rng, g, func, prc_index, channel, weights, grids, exc)
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grid), intent(inout) :: g
    integer, intent(in) :: prc_index
    integer, intent(in), optional :: channel
    real(kind=default), dimension(:), intent(in), optional :: weights
    type(vamp_grid), dimension(:), intent(in), optional :: grids
    type(exception), intent(inout), optional :: exc
    ⟨Interface declaration for func 22⟩
    character(len=*), parameter :: FN = "vamp_sample_grid0"
    ⟨Local variables in vamp_sample_grid0 86a⟩
    integer :: ndim
    ndim = size (g%div)
    ⟨Check optional arguments in vamp_sample_grid0 89a⟩
    ⟨Reset counters in vamp_sample_grid0 85c⟩
    loop_over_cells: do
        ⟨Sample calls_per_cell points in the current cell 86c⟩
        ⟨Collect integration and grid optimization data for current cell 88a⟩
        ⟨Count up cell, exit if done 85b⟩
    end do loop_over_cells
    ⟨Collect results of vamp_sample_grid0 88b⟩
end subroutine vamp_sample_grid0

```

Count cells like a n_g -ary number—i.e. $(1, \dots, 1, 1), (1, \dots, 1, 2), \dots, (1, \dots, 1, n_g), (1, \dots, 2, 1), \dots, (n_g, \dots, n_g, n_g - 1), (n_g, \dots, n_g, n_g)$ —and terminate when `all (cell == 1)` again.

85b *⟨Count up cell, exit if done 85b⟩* \equiv

```

do j = ndim, 1, -1
    cell(j) = modulo (cell(j), rigid_division (g%div(j))) + 1
    if (cell(j) /= 1) then
        cycle loop_over_cells
    end if
end do
exit loop_over_cells

```

85c *⟨Reset counters in vamp_sample_grid0 85c⟩* \equiv

```

g%mu = 0.0
cell = 1
call clear_integral_and_variance (g%div)
if (associated (g%mu_x)) then
    g%mu_x = 0.0

```

```

        g%mu_xx = 0.0
    end if
    if (present (channel)) then
        g%mu_gi = 0.0
    end if
86a  <Local variables in vamp_sample_grid0 86a>≡
      real(kind=default), parameter :: &
        eps = tiny (1._default) / epsilon (1._default)
      character(len=6) :: buffer
86b  <Local variables in vamp_sample_grid0 86a>+≡
      integer :: j, k
      integer, dimension(size(g%div)) :: cell
86c  <Sample calls_per_cell points in the current cell 86c>≡
      sum_f = 0.0
      sum_f2 = 0.0
      do k = 1, g%calls_per_cell
        <Get x in the current cell 86d>
        <f = wgt * func (x, weights, channel), iff x inside true_domain 87a>
        <Collect integration and grid optimization data for x from f 87b>
      end do

```

We are using the generic procedure `tao_random_number` from the `tao_random_numbers` module for generating an array of uniform deviates. A better alternative would be to pass the random number generator as an argument to `vamp_sample_grid`. Unfortunately, it is not possible to pass *generic* procedures in Fortran90, Fortran95, or F. While we could export a specific procedure from `tao_random_numbers`, a more serious problem is that we have to pass the state `rng` of the random number generator as a `tao_random_state` anyway and we have to hardcode the random number generator anyway.

```

86d  <Get x in the current cell 86d>≡
      call tao_random_number (rng, r)
      call inject_division (g%div, real (r, kind=default), &
        cell, x, x_mid, ia, wgts)
      wgt = g%jacobi * product (wgts)
      if (associated (g%map)) then
        x = matmul (g%map, x)
      end if

```

This somewhat contorted nested if constructs allow to minimize the number of calls to `func`. This is useful, since `func` is the most expensive part of real world applications. Also `func` might be singular outside of `true_domain`.

The original `vegas` used to call `f = wgt * func (x, wgt)` below to allow `func` to use `wgt` (i.e. $1/p(x)$) for integrating another function at the same

time. This form of “parallelism” relies on side effects and is therefore impossible with pure functions. Consequently, it is not supported in the current implementation.

```

87a  <f = wgt * func (x, weights, channel), iff x inside true_domain 87a>≡
      if (associated (g%map)) then
        if (all (inside_division (g%div, x))) then
          f = wgt * func (x, prc_index, weights, channel, grids)
        else
          f = 0.0
        end if
      else
        f = wgt * func (x, prc_index, weights, channel, grids)
      end if

```

```

87b  <Collect integration and grid optimization data for x from f 87b>≡
      if (g%f_min > g%f_max) then
        g%f_min = f * g%calls
        g%f_max = f * g%calls
      else if (f * g%calls < g%f_min) then
        g%f_min = f * g%calls
      else if (f * g%calls > g%f_max) then
        g%f_max = f * g%calls
      end if

```

```

87c  <Collect integration and grid optimization data for x from f 87b>+≡
      f2 = f * f
      sum_f = sum_f + f
      sum_f2 = sum_f2 + f2
      call record_integral (g%div, ia, f)
      ! call record_efficiency (g%div, ia, f/g%f_max)
      if ((associated (g%mu_x)) .and. (.not. g%all_stratified)) then
        g%mu_x = g%mu_x + x * f
        g%mu_xx = g%mu_xx + outer_product (x, x) * f
      end if
      if (present (channel)) then
        g%mu_gi = g%mu_gi + f2
      end if

```

```

87d  <Local variables in vamp_sample_grid0 86a>+≡
      real(kind=default) :: wgt, f, f2, sum_f, sum_f2, var_f
      real(kind=default), dimension(size(g%div)):: x, x_mid, wgt_s
      real(kind=default), dimension(size(g%div)):: r
      integer, dimension(size(g%div)) :: ia

```

$$\sigma^2 \cdot N_{\text{calls/cell}}^2 (N_{\text{calls/cell}} - 1) = \text{var}(f) = N^2 \sigma^2 \left(\left\langle \frac{f^2}{p} \right\rangle - \langle f \rangle^2 \right) \quad (5.17)$$

88a \langle Collect integration and grid optimization data for current cell 88a $\rangle \equiv$

```

var_f = sum_f2 * g%calls_per_cell - sum_f**2
if (var_f <= 0.0) then
  var_f = tiny (1.0_default)
end if
g%mu = g%mu + (/ sum_f, var_f /)
call record_variance (g%div, ia, var_f)
if ((associated (g%mu_x)) .and. g%all_stratified) then
  if (associated (g%map)) then
    x_mid = matmul (g%map, x_mid)
  end if
  g%mu_x = g%mu_x + x_mid * var_f
  g%mu_xx = g%mu_xx + outer_product (x_mid, x_mid) * var_f
end if

```

$$\sigma^2 = \frac{\left(\frac{N_{\text{calls}}}{N_{\text{cells}}}\right)^2}{N_{\text{calls/cell}}^2 (N_{\text{calls/cell}} - 1)} \sum_{\text{cells}} \sigma_{\text{cell}}^2 \cdot N_{\text{calls/cell}}^2 (N_{\text{calls/cell}} - 1) \quad (5.18)$$

where the N_{calls}^2 cancels the corresponding factor in the Jacobian and the N_{cells}^{-2} is the result of stratification.

88b \langle Collect results of vamp_sample_grid0 88b $\rangle \equiv$

```

g%mu(2) = g%mu(2) * g%dv2g
if (g%mu(2) < eps * max (g%mu(1)**2, 1._default)) then
  g%mu(2) = eps * max (g%mu(1)**2, 1._default)
end if

```

88c \langle Collect results of vamp_sample_grid0 88b $\rangle + \equiv$

```

if (g%mu(1)>0) then
  g%sum_integral = g%sum_integral + g%mu(1) / g%mu(2)
  g%sum_weights = g%sum_weights + 1.0 / g%mu(2)
  g%sum_chi2 = g%sum_chi2 + g%mu(1)**2 / g%mu(2)
  if (associated (g%mu_x)) then
    if (g%all_stratified) then
      g%mu_x = g%mu_x / g%mu(2)
      g%mu_xx = g%mu_xx / g%mu(2)
    else
      g%mu_x = g%mu_x / g%mu(1)
      g%mu_xx = g%mu_xx / g%mu(1)
    end if
  end if
  g%sum_mu_x = g%sum_mu_x + g%mu_x / g%mu(2)
  g%sum_mu_xx = g%sum_mu_xx + g%mu_xx / g%mu(2)
end if
if (present (channel)) then

```


```

        g%sum_mu_gi = g%sum_mu_gi + g%mu_gi / g%mu(2)
    end if
else
    if (present(channel) .and. g%mu(1)==0) then
        write (buffer, "(I6)") channel
        call raise_exception (exc, EXC_WARN, "! vamp", &
            "Function identically zero in channel " // buffer)
    else if (present(channel) .and. g%mu(1)<0) then
        write (buffer, "(I6)") channel
        call raise_exception (exc, EXC_ERROR, "! vamp", &
            "Negative integral in channel " // buffer)
    end if
    g%sum_integral = 0
    g%sum_chi2 = 0
    g%sum_weights = 0
end if

89a  <Check optional arguments in vamp_sample_grid0 89a>≡
    if (present (channel) .neqv. present (weights)) then
        call raise_exception (exc, EXC_FATAL, FN, &
            "channel and weights required together")
        return
    end if

89b  <Declaration of vamp procedures 76b>+≡
    public :: vamp_probability

89c  <Implementation of vamp procedures 76d>+≡
    pure function vamp_probability (g, x) result (p)
        type(vamp_grid), intent(in) :: g
        real(kind=default), dimension(:), intent(in) :: x
        real(kind=default) :: p
        p = product (probability (g%div, x))
    end function vamp_probability

 %variance should be private to division

89d  <Implementation of vamp procedures 76d>+≡
    subroutine vamp_apply_equivalences (g, eq)
        type(vamp_grids), intent(inout) :: g
        type(vamp_equivalences_t), intent(in) :: eq
        integer :: n_ch, n_dim, nb, i, ch, ch_src, dim, dim_src
        integer, dimension(:,:), allocatable :: n_bin
        real(kind=default), dimension(:,:,:), allocatable :: var_tmp
        n_ch = size (g%grids)
        if (n_ch == 0) return

```

```

n_dim = size (g%grids(1)%div)
allocate (n_bin(n_ch, n_dim))
do ch = 1, n_ch
  do dim = 1, n_dim
    n_bin(ch, dim) = size (g%grids(ch)%div(dim)%variance)
  end do
end do
allocate (var_tmp (maxval(n_bin), n_dim, n_ch))
var_tmp = 0
do i=1, eq%n_eq
  ch = eq%eq(i)%left
  ch_src = eq%eq(i)%right
  do dim=1, n_dim
    nb = n_bin(ch_src, dim)
    dim_src = eq%eq(i)%permutation(dim)
    select case (eq%eq(i)%mode(dim))
    case (VEQ_IDENTITY)
      var_tmp(:nb,dim,ch) = var_tmp(:nb,dim,ch) &
        & + g%grids(ch_src)%div(dim_src)%variance
    case (VEQ_INVERT)
      var_tmp(:nb,dim,ch) = var_tmp(:nb,dim,ch) &
        & + g%grids(ch_src)%div(dim_src)%variance(nb:1:-1)
    case (VEQ_SYMMETRIC)
      var_tmp(:nb,dim,ch) = var_tmp(:nb,dim,ch) &
        & + g%grids(ch_src)%div(dim_src)%variance / 2 &
        & + g%grids(ch_src)%div(dim_src)%variance(nb:1:-1)/2
    case (VEQ_INVARIANT)
      var_tmp(:nb,dim,ch) = 1
    end select
  end do
end do
do ch=1, n_ch
  do dim=1, n_dim
    g%grids(ch)%div(dim)%variance = var_tmp(:n_bin(ch, dim),dim,ch)
  end do
end do
deallocate (var_tmp)
deallocate (n_bin)
end subroutine vamp_apply_equivalences

```

Grid Refinement: r

$$n_{\text{div},j} \rightarrow \frac{Q_j n_{\text{div},j}}{\left(\prod_j Q_j\right)^{1/n_{\text{dim}}}} \quad (5.19)$$

where

$$Q_j = \left(\sqrt{\text{Var}(\{m\}_j)} \right)^\alpha \quad (5.20)$$

- 91a *⟨Implementation of vamp procedures 76d⟩*+≡

```

pure subroutine vamp_refine_grid (g, exc)
  type(vamp_grid), intent(inout) :: g
  type(exception), intent(inout), optional :: exc
  real(kind=default), dimension(size(g%div)) :: quad
  integer :: ndim
  if (g%quadrupole) then
    ndim = size (g%div)
    quad = (quadrupole_division (g%div))**QUAD_POWER
    call vamp_reshape_grid_internal &
      (g, use_variance = .true., exc = exc, &
        num_div = int (quad / product (quad)**(1.0/ndim) * g%num_div))
  else
    call refine_division (g%div)
  end if
end subroutine vamp_refine_grid

```
- 91b *⟨Implementation of vamp procedures 76d⟩*+≡

```

subroutine vamp_refine_grids (g)
  type(vamp_grids), intent(inout) :: g
  integer :: ch
  do ch=1, size(g%grids)
    call refine_division (g%grids(ch)%div)
  end do
end subroutine vamp_refine_grids

```
- 91c *⟨Variables in vamp 78a⟩*+≡

```

real(kind=default), private, parameter :: QUAD_POWER = 0.5_default

```

Adaptive Sampling: $S_n = S_0(rS_0)^n$

- 91d *⟨Implementation of vamp procedures 76d⟩*+≡

```

pure subroutine vamp_sample_grid &
  (rng, g, func, prc_index, iterations, &
    integral, std_dev, avg_chi2, accuracy, &
    channel, weights, grids, exc, history)

```

```

type(tao_random_state), intent(inout) :: rng
type(vamp_grid), intent(inout) :: g
integer, intent(in) :: prc_index
integer, intent(in) :: iterations
real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
real(kind=default), intent(in), optional :: accuracy
integer, intent(in), optional :: channel
real(kind=default), dimension(:), intent(in), optional :: weights
type(vamp_grid), dimension(:), intent(in), optional :: grids
type(exception), intent(inout), optional :: exc
type(vamp_history), dimension(:), intent(inout), optional :: history
<Interface declaration for func 22>
character(len=*), parameter :: FN = "vamp_sample_grid"
real(kind=default) :: local_integral, local_std_dev, local_avg_chi2
integer :: iteration, ndim
ndim = size(g%div)
iterate: do iteration = 1, iterations
    call vamp_sample_grid0 &
        (rng, g, func, prc_index, channel, weights, grids, exc)
    call vamp_average_iterations &
        (g, iteration, local_integral, local_std_dev, local_avg_chi2)
    <Trace results of vamp_sample_grid 103d>
    <Exit iterate if accuracy has been reached 94a>
    if (iteration < iterations) call vamp_refine_grid (g)
end do iterate
<Copy results of vamp_sample_grid to dummy variables 93c>
end subroutine vamp_sample_grid

```

Assuming that the iterations have been statistically independent, we can combine them with the usual formulae.

$$\bar{I} = \sigma_I^2 \sum_i \frac{I_i}{\sigma_i^2} \quad (5.21a)$$

$$\frac{1}{\sigma_I^2} = \sum_i \frac{1}{\sigma_i^2} \quad (5.21b)$$

$$\chi^2 = \sum_i \frac{(I_i - \bar{I})^2}{\sigma_i^2} = \sum_i \frac{I_i^2}{\sigma_i^2} - \bar{I} \sum_i \frac{I_i}{\sigma_i^2} \quad (5.21c)$$

92 *<Implementation of vamp procedures 76d>+≡*

```

elemental subroutine vamp_average_iterations_grid &
    (g, iteration, integral, std_dev, avg_chi2)
type(vamp_grid), intent(in) :: g
integer, intent(in) :: iteration

```

```

real(kind=default), intent(out) :: integral, std_dev, avg_chi2
real(kind=default), parameter :: eps = 1000 * epsilon (1._default)
if (g%sum_weights>0) then
  integral = g%sum_integral / g%sum_weights
  std_dev = sqrt (1.0 / g%sum_weights)
  avg_chi2 = &
    max ((g%sum_chi2 - g%sum_integral * integral) / (iteration-0.99), &
      0.0_default)
  if (avg_chi2 < eps * g%sum_chi2) avg_chi2 = 0
else
  integral = 0
  std_dev = 0
  avg_chi2 = 0
end if
end subroutine vamp_average_iterations_grid

```

93a *⟨Declaration of vamp procedures 76b⟩*+≡
 public :: vamp_average_iterations
 private :: vamp_average_iterations_grid

93b *⟨Interfaces of vamp procedures 93b⟩*≡
 interface vamp_average_iterations
 module procedure vamp_average_iterations_grid
 end interface

Lepage suggests [1] to reweight the contributions as in the following improved formulae, which we might implement as an option later.

$$\bar{I} = \frac{1}{\left(\sum_i \frac{I_i^2}{\sigma_i^2}\right)^2} \sum_i I_i \frac{I_i^2}{\sigma_i^2} \quad (5.22a)$$

$$\frac{1}{\sigma_I^2} = \frac{1}{(\bar{I})^2} \sum_i \frac{I_i^2}{\sigma_i^2} \quad (5.22b)$$

$$\chi^2 = \sum_i \frac{(I_i - \bar{I})^2}{(\bar{I})^2} \frac{I_i^2}{\sigma_i^2} \quad (5.22c)$$

Iff possible, copy the result to the caller's variables:

93c *⟨Copy results of vamp_sample_grid to dummy variables 93c⟩*≡
 if (present (integral)) then
 integral = local_integral
 end if
 if (present (std_dev)) then
 std_dev = local_std_dev
 end if

```

if (present (avg_chi2)) then
  avg_chi2 = local_avg_chi2
end if

```

94a *<Exit iterate if accuracy has been reached 94a>≡*

```

if (present (accuracy)) then
  if (local_std_dev <= accuracy * local_integral) then
    call raise_exception (exc, EXC_INFO, FN, &
      "requested accuracy reached")
    exit iterate
  end if
end if
end if

```

5.2.3 Forking and Joining

94b *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_fork_grid
private :: vamp_fork_grid_single, vamp_fork_grid_multi
public :: vamp_join_grid
private :: vamp_join_grid_single, vamp_join_grid_multi

```

94c *<Interfaces of vamp procedures 93b>+≡*

```

interface vamp_fork_grid
  module procedure vamp_fork_grid_single, vamp_fork_grid_multi
end interface
interface vamp_join_grid
  module procedure vamp_join_grid_single, vamp_join_grid_multi
end interface

```

Caveat emptor: splitting divisions can lead to `num_div < 3` and the application must not try to refine such grids before merging them again! `d == 0` is special.

94d *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_fork_grid_single (g, gs, d, exc)
  type(vamp_grid), intent(in) :: g
  type(vamp_grid), dimension(:), intent(inout) :: gs
  integer, intent(in) :: d
  type(exception), intent(inout), optional :: exc
  character(len=*), parameter :: FN = "vamp_fork_grid_single"
  type(division), dimension(:), allocatable :: d_tmp
  integer :: i, j, num_grids, num_div, ndim, num_cells
  num_grids = size (gs)
  ndim = size (g%div)
  <Allocate or resize the divisions 97a>

```

```

do j = 1, ndim
  if (j == d) then
    <call fork_division (g%div(j), gs%div(j), g%calls_per_cell, ...) 96c>
  else
    <call copy_division (gs%div(j), g%div(j)) 96e>
  end if
end do
if (d == 0) then
  <Handle g%calls_per_cell for d == 0 95a>
end if
<Copy the rest of g to the gs 95b>
end subroutine vamp_fork_grid_single
Divide the sampling points among identical grids
95a <Handle g%calls_per_cell for d == 0 95a>≡
  if (any (stratified_division (g%div))) then
    call raise_exception (exc, EXC_FATAL, FN, &
      "d == 0 incompatible w/ stratification")
  else
    gs(2:)%calls_per_cell = ceiling (real (g%calls_per_cell) / num_grids)
    gs(1)%calls_per_cell = g%calls_per_cell - sum (gs(2:)%calls_per_cell)
  end if
95b <Copy the rest of g to the gs 95b>≡
  do i = 1, num_grids
    call copy_array_pointer (gs(i)%num_div, g%num_div)
    if (associated (g%map)) then
      call copy_array_pointer (gs(i)%map, g%map)
    end if
    if (associated (g%mu_x)) then
      call create_array_pointer (gs(i)%mu_x, ndim)
      call create_array_pointer (gs(i)%sum_mu_x, ndim)
      call create_array_pointer (gs(i)%mu_xx, (/ ndim, ndim /))
      call create_array_pointer (gs(i)%sum_mu_xx, (/ ndim, ndim /))
    end if
  end do
Reset results
95c <Copy the rest of g to the gs 95b>+≡
  gs%mu(1) = 0.0
  gs%mu(2) = 0.0
  gs%sum_integral = 0.0
  gs%sum_weights = 0.0
  gs%sum_chi2 = 0.0
  gs%mu_gi = 0.0

```



```
gs%sum_mu_gi = 0.0
```

96a *<Copy the rest of g to the gs 95b>+≡*

```
gs%stratified = g%stratified
gs%all_stratified = g%all_stratified
gs%quadrupole = g%quadrupole
```

96b *<Copy the rest of g to the gs 95b>+≡*

```
do i = 1, num_grids
  num_cells = product (rigid_division (gs(i)%div))
  gs(i)%calls = gs(i)%calls_per_cell * num_cells
  gs(i)%num_calls = gs(i)%calls
  gs(i)%jacobi = product (volume_division (gs(i)%div)) / gs(i)%calls
  gs(i)%dv2g = (gs(i)%calls / num_cells)**2 &
    / gs(i)%calls_per_cell / gs(i)%calls_per_cell / (gs(i)%calls_per_cell - 1.0)
end do
gs%f_min = g%f_min * (gs%jacobi * gs%calls) / (g%jacobi * g%calls)
gs%f_max = g%f_max * (gs%jacobi * gs%calls) / (g%jacobi * g%calls)
```

This could be self-explaining, if the standard would allow Note that we can get away with copying just the pointers, because `fork_division` does the dirty work for the memory management.

96c *<call fork_division (g%div(j), gs%div(j), g%calls_per_cell, ...) 96c>≡*

```
allocate (d_tmp(num_grids))
do i = 1, num_grids
  d_tmp(i) = gs(i)%div(j)
end do
call fork_division (g%div(j), d_tmp, g%calls_per_cell, gs%calls_per_cell, exc)
do i = 1, num_grids
  gs(i)%div(j) = d_tmp(i)
end do
deallocate (d_tmp)
<Bail out if exception exc raised 96d>
```

96d *<Bail out if exception exc raised 96d>≡*

```
if (present (exc)) then
  if (exc%level > EXC_WARN) then
    return
  end if
end if
```

We have to do a deep copy (`gs(i)%div(j) = g%div(j)` does not suffice), because `copy_division` handles the memory management.

96e *<call copy_division (gs%div(j), g%div(j)) 96e>≡*

```
do i = 1, num_grids
  call copy_division (gs(i)%div(j), g%div(j))
```

```

end do

97a <Allocate or resize the divisions 97a>≡
  num_div = size (g%div)
  do i = 1, size (gs)
    if (associated (gs(i)%div)) then
      if (size (gs(i)%div) /= num_div) then
        allocate (gs(i)%div(num_div))
        call create_empty_division (gs(i)%div)
      end if
    else
      allocate (gs(i)%div(num_div))
      call create_empty_division (gs(i)%div)
    end if
  end do

97b <Implementation of vamp procedures 76d>+≡
  pure subroutine vamp_join_grid_single (g, gs, d, exc)
    type(vamp_grid), intent(inout) :: g
    type(vamp_grid), dimension(:), intent(inout) :: gs
    integer, intent(in) :: d
    type(exception), intent(inout), optional :: exc
    type(division), dimension(:), allocatable :: d_tmp
    integer :: i, j, num_grids
    num_grids = size (gs)
    do j = 1, size (g%div)
      if (j == d) then
        <call join_division (g%div(j), gs%div(j)) 97c>
      else
        <call sum_division (g%div(j), gs%div(j)) 97d>
      end if
    end do
    <Combine the rest of gs onto g 98a>
  end subroutine vamp_join_grid_single

97c <call join_division (g%div(j), gs%div(j)) 97c>≡
  allocate (d_tmp(num_grids))
  do i = 1, num_grids
    d_tmp(i) = gs(i)%div(j)
  end do
  call join_division (g%div(j), d_tmp, exc)
  deallocate (d_tmp)
  <Bail out if exception exc raised 96d>

97d <call sum_division (g%div(j), gs%div(j)) 97d>≡
  allocate (d_tmp(num_grids))

```

```

do i = 1, num_grids
  d_tmp(i) = gs(i)%div(j)
end do
call sum_division (g%div(j), d_tmp)
deallocate (d_tmp)

```

98a *⟨Combine the rest of gs onto g 98a⟩*≡

```

g%f_min = minval (gs%f_min * (g%jacobi * g%calls) / (gs%jacobi * gs%calls))
g%f_max = maxval (gs%f_max * (g%jacobi * g%calls) / (gs%jacobi * gs%calls))
g%mu(1) = sum (gs%mu(1))
g%mu(2) = sum (gs%mu(2))
g%mu_gi = sum (gs%mu_gi)
g%sum_mu_gi = g%sum_mu_gi + g%mu_gi / g%mu(2)
g%sum_integral = g%sum_integral + g%mu(1) / g%mu(2)
g%sum_chi2 = g%sum_chi2 + g%mu(1)**2 / g%mu(2)
g%sum_weights = g%sum_weights + 1.0 / g%mu(2)
if (associated (g%mu_x)) then
  do i = 1, num_grids
    g%mu_x = g%mu_x + gs(i)%mu_x
    g%mu_xx = g%mu_xx + gs(i)%mu_xx
  end do
  g%sum_mu_x = g%sum_mu_x + g%mu_x / g%mu(2)
  g%sum_mu_xx = g%sum_mu_xx + g%mu_xx / g%mu(2)
end if

```

The following is made a little bit hairy by the fact that `vamp_fork_grid` can't join grids onto a non-existing grid² therefore we have to keep a tree of joints. Maybe it would be the right thing to handle this tree of joints as a tree with pointers, but since we need the leaves flattened anyway (as food for multiple `vamp_sample_grid`) we use a similar storage layout for the joints.

98b *⟨Idioms 98b⟩*≡

```

type(vamp_grid), dimension(:), allocatable :: gx
integer, dimension(:,:), allocatable :: dim
...
allocate (gx(vamp_fork_grid_joints (dim)))
call vamp_fork_grid (g, gs, gx, dim, exc)
...
call vamp_join_grid (g, gs, gx, dim, exc)

```

98c *⟨Implementation of vamp procedures 76d⟩*+≡

```

pure recursive subroutine vamp_fork_grid_multi (g, gs, gx, d, exc)
  type(vamp_grid), intent(in) :: g

```

²It would be possible to make it possible by changing many things under the hood, but it doesn't really make sense, anyway.

```

type(vamp_grid), dimension(:), intent(inout) :: gs, gx
integer, dimension(:,:), intent(in) :: d
type(exception), intent(inout), optional :: exc
character(len=*), parameter :: FN = "vamp_fork_grid_multi"
integer :: i, offset, stride, joints_offset, joints_stride
select case (size (d, dim=2))
  case (0)
    return
  case (1)
    call vamp_fork_grid_single (g, gs, d(1,1), exc)
  case default
    offset = 1
    stride = product (d(2,2:))
    joints_offset = 1 + d(2,1)
    joints_stride = vamp_fork_grid_joints (d(:,2:))
    call vamp_create_empty_grid (gx(1:d(2,1)))
    call vamp_fork_grid_single (g, gx(1:d(2,1)), d(1,1), exc)
    do i = 1, d(2,1)
      call vamp_fork_grid_multi &
        (gx(i), gs(offset:offset+stride-1), &
         gx(joints_offset:joints_offset+joints_stride-1), &
         d(:,2:), exc)
      offset = offset + stride
      joints_offset = joints_offset + joints_stride
    end do
  end select
end subroutine vamp_fork_grid_multi

```

99a \langle Declaration of vamp procedures 76b $\rangle + \equiv$

```
public :: vamp_fork_grid_joints
```

$$\sum_{n=1}^{N-1} \prod_{i_n=1}^n d_{i_n} = d_1(1 + d_2(1 + d_3(1 + \dots(1 + d_{N-1}) \dots))) \quad (5.23)$$

99b \langle Implementation of vamp procedures 76d $\rangle + \equiv$

```

pure function vamp_fork_grid_joints (d) result (s)
  integer, dimension(:,:), intent(in) :: d
  integer :: s
  integer :: i
  s = 0
  do i = size (d, dim=2) - 1, 1, -1
    s = (s + 1) * d(2,i)
  end do
end function vamp_fork_grid_joints

```

100a *⟨Implementation of vamp procedures 76d⟩+≡*

```

pure recursive subroutine vamp_join_grid_multi (g, gs, gx, d, exc)
  type(vamp_grid), intent(inout) :: g
  type(vamp_grid), dimension(:), intent(inout) :: gs, gx
  integer, dimension(:,:), intent(in) :: d
  type(exception), intent(inout), optional :: exc
  character(len=*), parameter :: FN = "vamp_join_grid_multi"
  integer :: i, offset, stride, joints_offset, joints_stride
  select case (size (d, dim=2))
    case (0)
      return
    case (1)
      call vamp_join_grid_single (g, gs, d(1,1), exc)
    case default
      offset = 1
      stride = product (d(2,2:))
      joints_offset = 1 + d(2,1)
      joints_stride = vamp_fork_grid_joints (d(:,2:))
      do i = 1, d(2,1)
        call vamp_join_grid_multi &
          (gx(i), gs(offset:offset+stride-1), &
           gx(joints_offset:joints_offset+joints_stride-1), &
           d(:,2:), exc)
        offset = offset + stride
        joints_offset = joints_offset + joints_stride
      end do
      call vamp_join_grid_single (g, gx(1:d(2,1)), d(1,1), exc)
      call vamp_delete_grid (gx(1:d(2,1)))
  end select
end subroutine vamp_join_grid_multi

```

5.2.4 *Parallel Execution*

100b *⟨Declaration of vamp procedures 76b⟩+≡*

```

public :: vamp_sample_grid_parallel
public :: vamp_distribute_work

```

HPF [9, 10, 14]:

100c *⟨Implementation of vamp procedures 76d⟩+≡*

```

subroutine vamp_sample_grid_parallel &
  (rng, g, func, prc_index, iterations, &
   integral, std_dev, avg_chi2, accuracy, &
   channel, weights, grids, exc, history)

```

```

type(tao_random_state), dimension(:), intent(inout) :: rng
type(vamp_grid), intent(inout) :: g
integer, intent(in) :: prc_index
integer, intent(in) :: iterations
real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
real(kind=default), intent(in), optional :: accuracy
integer, intent(in), optional :: channel
real(kind=default), dimension(:), intent(in), optional :: weights
type(vamp_grid), dimension(:), intent(in), optional :: grids
type(exception), intent(inout), optional :: exc
type(vamp_history), dimension(:), intent(inout), optional :: history
<Interface declaration for func 22>
character(len=*), parameter :: FN = "vamp_sample_grid_parallel"
real(kind=default) :: local_integral, local_std_dev, local_avg_chi2
type(exception), dimension(size(rng)) :: excs
type(vamp_grid), dimension(:), allocatable :: gs, gx
!hpf$ processors p(number_of_processors())
!hpf$ distribute gs(cyclic(1)) onto p
integer, dimension(:,:), pointer :: d
integer :: iteration, i
integer :: num_workers
nullify (d)
call clear_exception (excs)
iterate: do iteration = 1, iterations
    call vamp_distribute_work (size (rng), vamp_rigid_divisions (g), d)
    num_workers = max (1, product (d(2,:)))
    if (num_workers > 1) then
        allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
        call vamp_create_empty_grid (gs)
        ! vamp_fork_grid is certainly not local. Speed freaks might
        ! want to tune it to the processor topology, but the gain will be small.
        call vamp_fork_grid (g, gs, gx, d, exc)
        !hpf$ independent
        do i = 1, num_workers
            call vamp_sample_grid0 &
                (rng(i), gs(i), func, prc_index, &
                 channel, weights, grids, exc)
        end do
        <Gather exceptions in vamp_sample_grid_parallel 102a>
        call vamp_join_grid (g, gs, gx, d, exc)
        call vamp_delete_grid (gs)
        deallocate (gs, gx)
    else

```

```

        call vamp_sample_grid0 &
            (rng(1), g, func, prc_index, channel, weights, grids, exc)
    end if
    <Bail out if exception exc raised 96d>
    call vamp_average_iterations &
        (g, iteration, local_integral, local_std_dev, local_avg_chi2)
    <Trace results of vamp_sample_grid 103d>
    <Exit iterate if accuracy has been reached 94a>
    if (iteration < iterations) call vamp_refine_grid (g)
end do iterate
deallocate (d)
<Copy results of vamp_sample_grid to dummy variables 93c>
end subroutine vamp_sample_grid_parallel

```

102a <Gather exceptions in vamp_sample_grid_parallel 102a>≡

```

    if ((present (exc)) .and. (any (excs(1:num_workers)%level > 0))) then
        call gather_exceptions (exc, excs(1:num_workers))
    end if

```

We could sort d such that (5.23) is minimized

$$d_1 \leq d_2 \leq \dots \leq d_N \quad (5.24)$$

but the gain will be negligible.

102b <Implementation of vamp procedures 76d>+≡

```

pure subroutine vamp_distribute_work (num_workers, ng, d)
    integer, intent(in) :: num_workers
    integer, dimension(:), intent(in) :: ng
    integer, dimension(:, :), pointer :: d
    integer, dimension(32) :: factors
    integer :: n, num_factors, i, j
    integer, dimension(size(ng)) :: num_forks
    integer :: nfork
    try: do n = num_workers, 1, -1
        call factorize (n, factors, num_factors)
        num_forks = 1
        do i = num_factors, 1, -1
            j = sum (maxloc (ng / num_forks))
            nfork = num_forks(j) * factors(i)
            if (nfork <= ng(j)) then
                num_forks(j) = nfork
            else
                cycle try
            end if
        end do
    end do

```

```

        <Accept distribution among n workers 103a>
    end do try
end subroutine vamp_distribute_work
103a <Accept distribution among n workers 103a>≡
    j = count (num_forks > 1)
    if (associated (d)) then
        if (size (d, dim = 2) /= j) then
            deallocate (d)
            allocate (d(2,j))
        end if
    else
        allocate (d(2,j))
    end if
103b <Accept distribution among n workers 103a>+≡
    j = 1
    do i = 1, size (ng)
        if (num_forks(i) > 1) then
            d(:,j) = (/ i, num_forks(i) /)
            j = j + 1
        end if
    end do
    return

```

5.2.5 Diagnostics

```

103c <Declaration of vamp types 103c>≡
    type, public :: vamp_history
    private
    real(kind=default) :: &
        integral, std_dev, avg_integral, avg_std_dev, avg_chi2, f_min, f_max
    integer :: calls
    logical :: stratified
    logical :: verbose
    type(div_history), dimension(:), pointer :: div => null ()
end type vamp_history
103d <Trace results of vamp_sample_grid 103d>≡
    if (present (history)) then
        if (iteration <= size (history)) then
            call vamp_get_history &
                (history(iteration), g, local_integral, local_std_dev, &
                    local_avg_chi2)
        else

```



```

        call raise_exception (exc, EXC_WARN, FN, "history too short")
    end if
    call vamp_terminate_history (history(iteration+1:))
end if

104a <Declaration of vamp procedures 76b>+≡
    public :: vamp_create_history, vamp_copy_history, vamp_delete_history
    public :: vamp_terminate_history
    public :: vamp_get_history, vamp_get_history_single

104b <Interfaces of vamp procedures 93b>+≡
    interface vamp_get_history
        module procedure vamp_get_history_single
    end interface

104c <Implementation of vamp procedures 76d>+≡
    elemental subroutine vamp_create_history (h, ndim, verbose)
        type(vamp_history), intent(out) :: h
        integer, intent(in), optional :: ndim
        logical, intent(in), optional :: verbose
        if (present (verbose)) then
            h%verbose = verbose
        else
            h%verbose = .false.
        end if
        h%calls = 0.0
        if (h%verbose .and. (present (ndim))) then
            if (associated (h%div)) then
                deallocate (h%div)
            end if
            allocate (h%div(ndim))
        end if
    end subroutine vamp_create_history

104d <Implementation of vamp procedures 76d>+≡
    elemental subroutine vamp_terminate_history (h)
        type(vamp_history), intent(inout) :: h
        h%calls = 0.0
    end subroutine vamp_terminate_history

104e <Implementation of vamp procedures 76d>+≡
    pure subroutine vamp_get_history_single (h, g, integral, std_dev, avg_chi2)
        type(vamp_history), intent(inout) :: h
        type(vamp_grid), intent(in) :: g
        real(kind=default), intent(in) :: integral, std_dev, avg_chi2
        h%calls = g%calls

```

```

    h%stratified = g%all_stratified
    h%integral = g%mu(1)
    h%std_dev = sqrt (g%mu(2))
    h%avg_integral = integral
    h%avg_std_dev = std_dev
    h%avg_chi2 = avg_chi2
    h%f_min = g%f_min
    h%f_max = g%f_max
    if (h%verbose) then
        ⟨Adjust h%div iff necessary 105a⟩
        call copy_history (h%div, summarize_division (g%div))
    end if
end subroutine vamp_get_history_single

105a ⟨Adjust h%div iff necessary 105a⟩≡
    if (associated (h%div)) then
        if (size (h%div) /= size (g%div)) then
            deallocate (h%div)
            allocate (h%div(size(g%div)))
        end if
    else
        allocate (h%div(size(g%div)))
    end if

105b ⟨Declaration of vamp procedures 76b⟩+≡
    public :: vamp_print_history, vamp_write_history
    private :: vamp_print_one_history, vamp_print_histories
    ! private :: vamp_write_one_history, vamp_write_histories

105c ⟨Interfaces of vamp procedures 93b⟩+≡
    interface vamp_print_history
        module procedure vamp_print_one_history, vamp_print_histories
    end interface
    interface vamp_write_history
        module procedure vamp_write_one_history_unit, vamp_write_histories_unit
    end interface

105d ⟨Implementation of vamp procedures 76d⟩+≡
    subroutine vamp_print_one_history (h, tag)
        type(vamp_history), dimension(:), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        type(div_history), dimension(:), allocatable :: h_tmp
        character(len=BUFFER_SIZE) :: pfx
        character(len=1) :: s
        integer :: i, imax, j
        if (present (tag)) then

```

```

    pfx = tag
else
    pfx = "[vamp]"
end if
print "(1X,A78)", repeat("-", 78)
print "(1X,A8,1X,A2,A9,A1,1X,A11,1X,8X,1X," &
    // "1X,A13,1X,8X,1X,A5,1X,A5)", &
    pfx, "it", "#calls", "", "integral", "average", "chi2", "eff."
imax = size(h)
iterations: do i = 1, imax
    if (h(i)%calls <= 0) then
        imax = i - 1
        exit iterations
    end if
    if (h(i)%stratified) then
        s = "*"
    else
        s = ""
    end if
    print "(1X,A8,1X,I2,I9,A1,1X,E11.4,A1,E8.2,A1," &
        // "1X,E13.6,A1,E8.2,A1,F5.1,1X,F5.3)", pfx, &
        i, h(i)%calls, s, h(i)%integral, "(", h(i)%std_dev, ")", &
        h(i)%avg_integral, "(", h(i)%avg_std_dev, ")", h(i)%avg_chi2, &
        h(i)%integral / h(i)%f_max
end do iterations
print "(1X,A78)", repeat("-", 78)
if (all(h%verbose) .and. (imax >= 1)) then
    if (associated(h(1)%div)) then
        allocate(h_tmp(imax))
        dimensions: do j = 1, size(h(1)%div)
            do i = 1, imax
                call copy_history(h_tmp(i), h(i)%div(j))
            end do
            if (present(tag)) then
                write(unit = pfx, fmt = "(A,A1,I2.2)") &
                    trim(tag(1:min(len_trim(tag),8))), "#", j
            else
                write(unit = pfx, fmt = "(A,A1,I2.2)") "[vamp]", "#", j
            end if
            call print_history(h_tmp, tag = pfx)
        end do dimensions
        print "(1X,A78)", repeat("-", 78)
    end do dimensions
    deallocate(h_tmp)
end if

```

```

        end if
    end if
    flush (output_unit)
end subroutine vamp_print_one_history

107a <Variables in vamp 78a>+≡
    integer, private, parameter :: BUFFER_SIZE = 50

107b <Implementation of vamp procedures 76d>+≡
    subroutine vamp_print_histories (h, tag)
        type(vamp_history), dimension(:, :), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        character(len=BUFFER_SIZE) :: pfx
        integer :: i
        print "(1X,A78)", repeat ("=", 78)
        channels: do i = 1, size (h, dim=2)
            if (present (tag)) then
                write (unit = pfx, fmt = "(A4,A1,I3.3)") tag, "#", i
            else
                write (unit = pfx, fmt = "(A4,A1,I3.3)") "chan", "#", i
            end if
            call vamp_print_one_history (h(:,i), pfx)
        end do channels
        print "(1X,A78)", repeat ("=", 78)
        flush (output_unit)
    end subroutine vamp_print_histories

```



```

107c <Implementation of vamp procedures 76d>+≡
    subroutine vamp_write_one_history_unit (u, h, tag)
        integer, intent(in) :: u
        type(vamp_history), dimension(:), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        type(div_history), dimension(:), allocatable :: h_tmp
        character(len=BUFFER_SIZE) :: pfx
        character(len=1) :: s
        integer :: i, imax, j
        if (present (tag)) then
            pfx = tag
        else
            pfx = "[vamp]"
        end if
        write (u, "(1X,A78)") repeat ("-", 78)
        write (u, "(1X,A8,1X,A2,A9,A1,1X,A11,1X,8X,1X," &

```

```

        // "1X,A13,1X,8X,1X,A5,1X,A5)") &
        pfx, "it", "#calls", "", "integral", "average", "chi2", "eff."
imax = size (h)
iterations: do i = 1, imax
    if (h(i)%calls <= 0) then
        imax = i - 1
        exit iterations
    end if
    ! *WK: Skip zero channel
    if (h(i)%f_max==0) cycle
    if (h(i)%stratified) then
        s = "*"
    else
        s = ""
    end if
    write (u, "(1X,A8,1X,I2,I9,A1,1X,E11.4,A1,E8.2,A1," &
        // "1X,E13.6,A1,E8.2,A1,F5.1,1X,F5.3)") pfx, &
        i, h(i)%calls, s, h(i)%integral, "(", h(i)%std_dev, ")", &
        h(i)%avg_integral, "(", h(i)%avg_std_dev, ")", h(i)%avg_chi2, &
        h(i)%integral / h(i)%f_max
end do iterations
write (u, "(1X,A78)") repeat ("-", 78)
if (all (h%verbose) .and. (imax >= 1)) then
    if (associated (h(1)%div)) then
        allocate (h_tmp(imax))
        dimensions: do j = 1, size (h(1)%div)
            do i = 1, imax
                call copy_history (h_tmp(i), h(i)%div(j))
            end do
            if (present (tag)) then
                write (unit = pfx, fmt = "(A,A1,I2.2)") &
                    trim (tag(1:min(len_trim(tag),8))), "#", j
            else
                write (unit = pfx, fmt = "(A,A1,I2.2)") "[vamp]", "#", j
            end if
            call write_history (u, h_tmp, tag = pfx)
            print "(1X,A78)", repeat ("-", 78)
        end do dimensions
        deallocate (h_tmp)
    end if
end if
flush (u)
end subroutine vamp_write_one_history_unit

```

```

subroutine vamp_write_histories_unit (u, h, tag)
  integer, intent(in) :: u
  type(vamp_history), dimension(:, :), intent(in) :: h
  character(len=*), intent(in), optional :: tag
  character(len=BUFFER_SIZE) :: pfx
  integer :: i
  write (u, "(1X,A78)") repeat ("=", 78)
  channels: do i = 1, size (h, dim=2)
    if (present (tag)) then
      write (unit = pfx, fmt = "(A4,A1,I3.3)") tag, "#", i
    else
      write (unit = pfx, fmt = "(A4,A1,I3.3)") "chan", "#", i
    end if
    call vamp_write_one_history_unit (u, h(:, i), pfx)
  end do channels
  write (u, "(1X,A78)") repeat ("=", 78)
  flush (u)
end subroutine vamp_write_histories_unit

```

5.2.6 Multi Channel

[22]

$$g(x) = \sum_i \alpha_i g_i(x) \quad (5.25a)$$

$$w(x) = \frac{f(x)}{g(x)} \quad (5.25b)$$

We want to minimize the variance $W(\alpha)$ with the subsidiary condition $\sum_i \alpha_i = 1$. We introduce a Lagrange multiplier λ :

$$\tilde{W}(\alpha) = W(\alpha) + \lambda \left(\sum_i \alpha_i - 1 \right) \quad (5.26)$$

Therefore...

$$W_i(\alpha) = -\frac{\partial}{\partial \alpha_i} W(\alpha) = \int dx g_i(x) (w(x))^2 \approx \left\langle \frac{g_i(x)}{g(x)} (w(x))^2 \right\rangle \quad (5.27)$$



Here it *really* hurts that **Fortran** has no *first-class* functions. The following can be expressed much more elegantly in a functional programming language with *first-class* functions, currying and closures. **Fortran** makes it extra painful since not even procedure pointers are supported. This puts extra burden on the users of this library.

Note that the components of `vamp_grids` are not protected. However, this is not a license for application programs to access it. Only Other libraries (e.g. for parallel processing, like `vampi`) should do so.

110a \langle Declaration of `vamp` types 103c $\rangle + \equiv$

```
type, public :: vamp_grids
  !!! private ! used by vampi
  real(kind=default), dimension(:), pointer :: weights => null ()
  type(vamp_grid), dimension(:), pointer :: grids => null ()
  integer, dimension(:), pointer :: num_calls => null ()
  real(kind=default) :: sum_chi2, sum_integral, sum_weights
end type vamp_grids
```

$$g \circ \phi_i = \left| \frac{\partial \phi_i}{\partial x} \right|^{-1} \left(\alpha_i g_i + \sum_{\substack{j=1 \\ j \neq i}}^{N_c} \alpha_j (g_j \circ \pi_{ij}) \left| \frac{\partial \pi_{ij}}{\partial x} \right| \right). \quad (5.28)$$

110b \langle Declaration of `vamp` procedures 76b $\rangle + \equiv$

```
public :: vamp_multi_channel, vamp_multi_channel0
```

110c \langle Implementation of `vamp` procedures 76d $\rangle + \equiv$

```
pure function vamp_multi_channel &
  (func, prc_index, phi, ihp, jacobian, x, weights, channel, grids) result (w_x)
  integer, intent(in) :: prc_index
  real(kind=default), dimension(:), intent(in) :: x
  real(kind=default), dimension(:), intent(in) :: weights
  integer, intent(in) :: channel
  type(vamp_grid), dimension(:), intent(in) :: grids
   $\langle$ Interface declaration for func 22 $\rangle$ 
   $\langle$ Interface declaration for phi 31a $\rangle$ 
   $\langle$ Interface declaration for ihp 31b $\rangle$ 
   $\langle$ Interface declaration for jacobian 31c $\rangle$ 
  real(kind=default) :: w_x
  integer :: i
  real(kind=default), dimension(size(x)) :: phi_x
  real(kind=default), dimension(size(weights)) :: g_phi_x, g_pi_x
  phi_x = phi (x, channel)
  do i = 1, size (weights)
    if (i == channel) then
      g_pi_x(i) = vamp_probability (grids(i), x)
    else
      g_pi_x(i) = vamp_probability (grids(i), ihp (phi_x, i))
    end if
  end do
  do i = 1, size (weights)
```

```

        g_phi_x(i) = g_pi_x(i) / g_pi_x(channel) * jacobian (phi_x, prc_index, i)
    end do
    w_x = func (phi_x, prc_index, weights, channel, grids) &
        / dot_product (weights, g_phi_x)
end function vamp_multi_channel

```

111a *<Implementation of vamp procedures 76d>+≡*

```

pure function vamp_multi_channel0 &
    (func, prc_index, phi, jacobian, x, weights, channel) result (w_x)
    integer, intent(in) :: prc_index
    real(kind=default), dimension(:), intent(in) :: x
    real(kind=default), dimension(:), intent(in) :: weights
    integer, intent(in) :: channel
    <Interface declaration for func 22>
    <Interface declaration for phi 31a>
    <Interface declaration for jacobian 31c>
    real(kind=default) :: w_x
    real(kind=default), dimension(size(x)) :: x_prime
    real(kind=default), dimension(size(weights)) :: g_phi_x
    integer :: i
    x_prime = phi (x, channel)
    do i = 1, size (weights)
        g_phi_x(i) = jacobian (x_prime, prc_index, i)
    end do
    w_x = func (x_prime, prc_index) / dot_product (weights, g_phi_x)
end function vamp_multi_channel0

```



111b *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_jacobian, vamp_check_jacobian

```

111c *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_jacobian (phi, channel, x, region, jacobian, delta_x)
    integer, intent(in) :: channel
    real(kind=default), dimension(:), intent(in) :: x
    real(kind=default), dimension(:, :), intent(in) :: region
    real(kind=default), intent(out) :: jacobian
    real(kind=default), intent(in), optional :: delta_x
    interface
        function phi (xi, channel) result (x)
            use kinds
            real(kind=default), dimension(:), intent(in) :: xi
            integer, intent(in) :: channel
            real(kind=default), dimension(size(xi)) :: x

```



```

        end function phi
    end interface
    real(kind=default), dimension(size(x)) :: x_min, x_max
    real(kind=default), dimension(size(x)) :: x_plus, x_minus
    real(kind=default), dimension(size(x),size(x)) :: d_phi
    real(kind=default), parameter :: &
        dx_default = 10.0_default**(-precision(jacobian)/3)
    real(kind=default) :: dx
    integer :: j
    if (present (delta_x)) then
        dx = delta_x
    else
        dx = dx_default
    end if
    x_min = region(1,:)
    x_max = region(2,:)
    x_minus = max (x_min, x)
    x_plus = min (x_max, x)
    do j = 1, size (x)
        x_minus(j) = max (x_min(j), x(j) - dx)
        x_plus(j) = min (x_max(j), x(j) + dx)
        d_phi(:,j) = (phi (x_plus, channel) - phi (x_minus, channel)) &
            / (x_plus(j) - x_minus(j))
        x_minus(j) = max (x_min(j), x(j))
        x_plus(j) = min (x_max(j), x(j))
    end do
    call determinant (d_phi, jacobian)
    jacobian = abs (jacobian)
end subroutine vamp_jacobian

```

$$g(\phi(x)) = \frac{1}{\left| \frac{\partial \phi}{\partial x} \right| (x)} \quad (5.29)$$

112 *⟨Implementation of vamp procedures 76d⟩*+≡

```

pure subroutine vamp_check_jacobian &
    (rng, n, func, prc_index, phi, channel, region, delta, x_delta)
type(tao_random_state), intent(inout) :: rng
integer, intent(in) :: n
integer, intent(in) :: prc_index
integer, intent(in) :: channel
real(kind=default), dimension(:,:), intent(in) :: region
real(kind=default), intent(out) :: delta
real(kind=default), dimension(:), intent(out), optional :: x_delta
⟨Interface declaration for func 22⟩
⟨Interface declaration for phi 31a⟩

```

```

real(kind=default), dimension(size(region,dim=2)) :: x, r
real(kind=default) :: jac, d
real(kind=default), dimension(0) :: wgts
integer :: i
delta = 0.0
do i = 1, max (1, n)
  call tao_random_number (rng, r)
  x = region(1,:) + (region(2,:) - region(1,:)) * r
  call vamp_jacobian (phi, channel, x, region, jac)
  d = func (phi (x, channel), prc_index, wgts, channel) * jac &
    - 1.0_default
  if (abs (d) >= abs (delta)) then
    delta = d
    if (present (x_delta)) then
      x_delta = x
    end if
  end if
end do
end subroutine vamp_check_jacobian

```

This is a subroutine to comply with F's rules, otherwise, we would code it as a function.

113a *<Declaration of vamp procedures (removed from WHIZARD) 113a>≡*
 private :: numeric_jacobian

113b *<Implementation of vamp procedures (removed from WHIZARD) 113b>≡*
 pure subroutine numeric_jacobian (phi, channel, x, region, jacobian, delta_x)
 integer, intent(in) :: channel
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:,,:), intent(in) :: region
 real(kind=default), intent(out) :: jacobian
 real(kind=default), intent(in), optional :: delta_x
<Interface declaration for phi 31a>
 real(kind=default), dimension(size(x)) :: x_min, x_max
 real(kind=default), dimension(size(x)) :: x_plus, x_minus
 real(kind=default), dimension(size(x),size(x)) :: d_phi
 real(kind=default), parameter :: &
 dx_default = 10.0_default**(-precision(jacobian)/3)
 real(kind=default) :: dx
 integer :: j
 if (present (delta_x)) then
 dx = delta_x
 else
 dx = dx_default
 end if
end subroutine numeric_jacobian

```

end if
x_min = region(1,:)
x_max = region(2,:)
x_minus = max (x_min, x)
x_plus = min (x_max, x)
do j = 1, size (x)
  x_minus(j) = max (x_min(j), x(j) - dx)
  x_plus(j) = min (x_max(j), x(j) + dx)
  d_phi(:,j) = (phi (x_plus, channel) - phi (x_minus, channel)) &
               / (x_plus(j) - x_minus(j))
  x_minus(j) = max (x_min(j), x(j))
  x_plus(j) = min (x_max(j), x(j))
end do
call determinant (d_phi, jacobian)
jacobian = abs (jacobian)
end subroutine numeric_jacobian

```

114a *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_create_grids, vamp_create_empty_grids
public :: vamp_copy_grids, vamp_delete_grids

```

The rules for optional arguments forces us to handle special cases, because we can't just pass a array section of an optional array as an actual argument (cf. 12.4.1.5(4) in [8]) even if the dummy argument is optional itself.

114b *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_create_grids &
  (g, domain, num_calls, weights, maps, num_div, &
   stratified, quadrupole, exc)
type(vamp_grids), intent(inout) :: g
real(kind=default), dimension(:,,:), intent(in) :: domain
integer, intent(in) :: num_calls
real(kind=default), dimension(:), intent(in) :: weights
real(kind=default), dimension(:,:,:), intent(in), optional :: maps
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
type(exception), intent(inout), optional :: exc
character(len=*), parameter :: FN = "vamp_create_grids"
integer :: ch, nch
nch = size (weights)
allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
g%weights = weights / sum (weights)
g%num_calls = g%weights * num_calls
do ch = 1, size (g%grids)
  if (present (maps)) then

```

```

        call vamp_create_grid &
            (g%grids(ch), domain, g%num_calls(ch), num_div, &
             stratified, quadrupole, map = maps(:, :, ch), exc = exc)
    else
        call vamp_create_grid &
            (g%grids(ch), domain, g%num_calls(ch), num_div, &
             stratified, quadrupole, exc = exc)
    end if
end do
g%sum_integral = 0.0
g%sum_chi2 = 0.0
g%sum_weights = 0.0
end subroutine vamp_create_grids

115a <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_create_empty_grids (g)
    type(vamp_grids), intent(inout) :: g
    nullify (g%grids, g%weights, g%num_calls)
end subroutine vamp_create_empty_grids

115b <Declaration of vamp procedures 76b>+≡
public :: vamp_discard_integrals

115c <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_discard_integrals &
    (g, num_calls, num_div, stratified, quadrupole, exc, eq)
    type(vamp_grids), intent(inout) :: g
    integer, intent(in), optional :: num_calls
    integer, dimension(:), intent(in), optional :: num_div
    logical, intent(in), optional :: stratified, quadrupole
    type(exception), intent(inout), optional :: exc
    type(vamp_equivalences_t), intent(in), optional :: eq
    integer :: ch
    character(len=*), parameter :: FN = "vamp_discard_integrals"
    g%sum_integral = 0.0
    g%sum_weights = 0.0
    g%sum_chi2 = 0.0
    do ch = 1, size (g%grids)
        call vamp_discard_integral (g%grids(ch))
    end do
    if (present (num_calls)) then
        call vamp_reshape_grids &
            (g, num_calls, num_div, stratified, quadrupole, exc, eq)
    end if
end subroutine vamp_discard_integrals

```

116a \langle Declaration of vamp procedures 76b $\rangle + \equiv$

```
public :: vamp_update_weights
```

We must discard the accumulated integrals, because the weight function $w = f / \sum_i \alpha_i g_i$ changes:

116b \langle Implementation of vamp procedures 76d $\rangle + \equiv$

```
pure subroutine vamp_update_weights &
  (g, weights, num_calls, num_div, stratified, quadrupole, exc)
  type(vamp_grids), intent(inout) :: g
  real(kind=default), dimension(:), intent(in) :: weights
  integer, intent(in), optional :: num_calls
  integer, dimension(:), intent(in), optional :: num_div
  logical, intent(in), optional :: stratified, quadrupole
  type(exception), intent(inout), optional :: exc
  character(len=*), parameter :: FN = "vamp_update_weights"
  if (sum (weights) > 0) then
    g%weights = weights / sum (weights)
  else
    g%weights = 1._default / size(g%weights)
  end if
  if (present (num_calls)) then
    call vamp_discard_integrals (g, num_calls, num_div, &
                               stratified, quadrupole, exc)
  else
    call vamp_discard_integrals (g, sum (g%num_calls), num_div, &
                               stratified, quadrupole, exc)
  end if
end subroutine vamp_update_weights
```

116c \langle Declaration of vamp procedures 76b $\rangle + \equiv$

```
public :: vamp_reshape_grids
```

116d \langle Implementation of vamp procedures 76d $\rangle + \equiv$

```
pure subroutine vamp_reshape_grids &
  (g, num_calls, num_div, stratified, quadrupole, exc, eq)
  type(vamp_grids), intent(inout) :: g
  integer, intent(in) :: num_calls
  integer, dimension(:), intent(in), optional :: num_div
  logical, intent(in), optional :: stratified, quadrupole
  type(exception), intent(inout), optional :: exc
  type(vamp_equivalences_t), intent(in), optional :: eq
  integer, dimension(size(g%grids(1)%num_div)) :: num_div_new
  integer :: ch
  character(len=*), parameter :: FN = "vamp_reshape_grids"
  g%num_calls = g%weights * num_calls
```

```

do ch = 1, size (g%grids)
  if (g%num_calls(ch) >= 2) then
    if (present (eq)) then
      if (present (num_div)) then
        num_div_new = num_div
      else
        num_div_new = g%grids(ch)%num_div
      end if
      where (eq%div_is_invariant(ch,:))
        num_div_new = 1
      end where
      call vamp_reshape_grid (g%grids(ch), g%num_calls(ch), &
        num_div_new, stratified, quadrupole, exc = exc, &
        independent = eq%independent(ch), &
        equivalent_to_ch = eq%equivalent_to_ch(ch), &
        multiplicity = eq%multiplicity(ch))
    else
      call vamp_reshape_grid (g%grids(ch), g%num_calls(ch), &
        num_div, stratified, quadrupole, exc = exc)
    end if
  else
    g%num_calls(ch) = 0
  end if
end do
end subroutine vamp_reshape_grids

```

117a *⟨Declaration of vamp procedures 76b⟩*+≡
 public :: vamp_sample_grids

Even if `g%num_calls` is derived from `g%weights`, we must *not* use the latter, allow for integer arithmetic in `g%num_calls`.

117b *⟨Implementation of vamp procedures 76d⟩*+≡
 pure subroutine vamp_sample_grids &
 (rng, g, func, prc_index, iterations, integral, std_dev, avg_chi2, &
 accuracy, history, histories, exc, eq, warn_error)
 type(tao_random_state), intent(inout) :: rng
 type(vamp_grids), intent(inout) :: g
 integer, intent(in) :: prc_index
 integer, intent(in) :: iterations
 real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
 real(kind=default), intent(in), optional :: accuracy
 type(vamp_history), dimension(:), intent(inout), optional :: history
 type(vamp_history), dimension(:, :), intent(inout), optional :: histories
 type(exception), intent(inout), optional :: exc

```

type(vamp_equivalences_t), intent(in), optional :: eq
logical, intent(in), optional :: warn_error
<Interface declaration for func 22>
integer :: ch, iteration
type(exception), dimension(size(g%grids)) :: excs
logical, dimension(size(g%grids)) :: active
real(kind=default), dimension(size(g%grids)) :: weights, integrals, std_devs
real(kind=default) :: local_integral, local_std_dev, local_avg_chi2
character(len=*), parameter :: FN = "vamp_sample_grids"
integrals = 0
std_devs = 0
active = (g%num_calls >= 2)
where (active)
    weights = g%num_calls
elsewhere
    weights = 0.0
endwhere
if (sum (weights) /= 0) weights = weights / sum (weights)
call clear_exception (excs)
iterate: do iteration = 1, iterations
    do ch = 1, size (g%grids)
        if (active(ch)) then
            call vamp_discard_integral (g%grids(ch))
            <Sample the grid g%grids(ch) 119a>
        else
            call vamp_nullify_variance (g%grids(ch))
            call vamp_nullify_covariance (g%grids(ch))
        end if
    end do
    if (present(eq)) call vamp_apply_equivalences (g, eq)
    if (iteration < iterations) then
        do ch = 1, size (g%grids)
            active(ch) = (integrals(ch) /= 0)
            if (active(ch)) then
                call vamp_refine_grid (g%grids(ch))
            end if
        end do
    end if
    if (present (exc) .and. (any (excs%level > 0))) then
        call gather_exceptions (exc, excs)
!       return
    end if
    call vamp_reduce_channels (g, integrals, std_devs, active)

```

```

    call vamp_average_iterations &
      (g, iteration, local_integral, local_std_dev, local_avg_chi2)
    <Trace results of vamp_sample_grids 121d>
    <Exit iterate if accuracy has been reached 94a>
  end do iterate
  <Copy results of vamp_sample_grid to dummy variables 93c>
end subroutine vamp_sample_grids

```

We must refine the grids after *all* grids have been sampled, therefore we use `vamp_sample_grid0` instead of `vamp_sample_grid`:

```

119a <Sample the grid g%grids(ch) 119a>≡
  call vamp_sample_grid0 &
    (rng, g%grids(ch), func, prc_index, &
     ch, weights, g%grids, excs(ch))
  if (present (exc) .and. present (warn_error)) then
    if (warn_error) call handle_exception (excs(ch))
  end if
  call vamp_average_iterations &
    (g%grids(ch), iteration, integrals(ch), std_devs(ch), local_avg_chi2)
  if (present (histories)) then
    if (iteration <= ubound (histories, dim=1)) then
      call vamp_get_history &
        (histories(iteration,ch), g%grids(ch), &
         integrals(ch), std_devs(ch), local_avg_chi2)
    else
      call raise_exception (exc, EXC_WARN, FN, "history too short")
    end if
    call vamp_terminate_history (histories(iteration+1:,ch))
  end if
119b <Declaration of vamp procedures 76b>+≡
  public :: vamp_reduce_channels

```

$$I = \frac{1}{N} \sum_c N_c I_c \quad (5.30a)$$

$$\sigma^2 = \frac{1}{N^2} \sum_c N_c^2 \sigma_c^2 \quad (5.30b)$$

$$N = \sum_c N_c \quad (5.30c)$$

where (5.30b) is actually

$$\sigma^2 = \frac{1}{N} (\mu_2 - \mu_1^2) = \frac{1}{N} \left(\frac{1}{N} \sum_c N_c \mu_{2,c} - I^2 \right) = \frac{1}{N} \left(\frac{1}{N} \sum_c (N_c^2 \sigma_c^2 + N_c I_c^2) - I^2 \right)$$

but the latter form suffers from numerical instability and (5.30b) is thus preferred.

120a *⟨Implementation of vamp procedures 76d⟩*+≡

```
pure subroutine vamp_reduce_channels (g, integrals, std_devs, active)
  type(vamp_grids), intent(inout) :: g
  real(kind=default), dimension(:), intent(in) :: integrals, std_devs
  logical, dimension(:), intent(in) :: active
  real(kind=default) :: this_integral, this_weight, total_calls
  real(kind=default) :: total_variance
  if (.not.any(active)) return
  total_calls = sum (g%num_calls, mask=active)
  if (total_calls > 0) then
    this_integral = sum (g%num_calls * integrals, mask=active) / total_calls
  else
    this_integral = 0
  end if
  total_variance = sum ((g%num_calls*std_devs)**2, mask=active)
  if (total_variance > 0) then
    this_weight = total_calls**2 / total_variance
  else
    this_weight = 0
  end if
  g%sum_weights = g%sum_weights + this_weight
  g%sum_integral = g%sum_integral + this_weight * this_integral
  g%sum_chi2 = g%sum_chi2 + this_weight * this_integral**2
end subroutine vamp_reduce_channels
```

120b *⟨Declaration of vamp procedures 76b⟩*+≡

```
public :: vamp_refine_weights
```

120c *⟨Implementation of vamp procedures 76d⟩*+≡

```
elemental subroutine vamp_average_iterations_grids &
  (g, iteration, integral, std_dev, avg_chi2)
  type(vamp_grids), intent(in) :: g
  integer, intent(in) :: iteration
  real(kind=default), intent(out) :: integral, std_dev, avg_chi2
  real(kind=default), parameter :: eps = 1000 * epsilon (1._default)
  if (g%sum_weights>0) then
    integral = g%sum_integral / g%sum_weights
    std_dev = sqrt (1.0 / g%sum_weights)
    avg_chi2 = &
      max ((g%sum_chi2 - g%sum_integral * integral) / (iteration-0.99), &
        0.0_default)
    if (avg_chi2 < eps * g%sum_chi2) avg_chi2 = 0
```

```

        else
            integral = 0
            std_dev = 0
            avg_chi2 = 0
        end if
    end subroutine vamp_average_iterations_grids
121a <Declaration of vamp procedures 76b>+≡
    private :: vamp_average_iterations_grids
121b <Interfaces of vamp procedures 93b>+≡
    interface vamp_average_iterations
        module procedure vamp_average_iterations_grids
    end interface

```

$$\alpha_i \rightarrow \alpha_i \sqrt{V_i} \quad (5.31)$$

```

121c <Implementation of vamp procedures 76d>+≡
    pure subroutine vamp_refine_weights (g, power)
        type(vamp_grids), intent(inout) :: g
        real(kind=default), intent(in), optional :: power
        real(kind=default) :: local_power
        real(kind=default), parameter :: DEFAULT_POWER = 0.5_default
        if (present (power)) then
            local_power = power
        else
            local_power = DEFAULT_POWER
        end if
        call vamp_update_weights &
            (g, g%weights * vamp_get_variance (g%grids) ** local_power)
    end subroutine vamp_refine_weights
121d <Trace results of vamp_sample_grids 121d>≡
    if (present (history)) then
        if (iteration <= size (history)) then
            call vamp_get_history &
                (history(iteration), g, local_integral, local_std_dev, &
                    local_avg_chi2)
        else
            call raise_exception (exc, EXC_WARN, FN, "history too short")
        end if
        call vamp_terminate_history (history(iteration+1:))
    end if
121e <Declaration of vamp procedures 76b>+≡
    private :: vamp_get_history_multi

```

122a *⟨Interfaces of vamp procedures 93b⟩*+≡

```
interface vamp_get_history
  module procedure vamp_get_history_multi
end interface
```

122b *⟨Implementation of vamp procedures 76d⟩*+≡

```
pure subroutine vamp_get_history_multi (h, g, integral, std_dev, avg_chi2)
  type(vamp_history), intent(inout) :: h
  type(vamp_grids), intent(in) :: g
  real(kind=default), intent(in) :: integral, std_dev, avg_chi2
  h%calls = sum (g%grids%calls)
  h%stratified = all (g%grids%all_stratified)
  h%integral = 0.0
  h%std_dev = 0.0
  h%avg_integral = integral
  h%avg_std_dev = std_dev
  h%avg_chi2 = avg_chi2
  h%f_min = 0.0
  h%f_max = huge (h%f_max)
  if (h%verbose) then
    h%verbose = .false.
    if (associated (h%div)) then
      deallocate (h%div)
    end if
  end if
end subroutine vamp_get_history_multi
```



122c *⟨Declaration of vamp procedures 76b⟩*+≡

```
public :: vamp_sum_channels
```

122d *⟨Implementation of vamp procedures 76d⟩*+≡

```
pure function vamp_sum_channels (x, weights, func, prc_index, grids) result (g)
  real(kind=default), dimension(:), intent(in) :: x, weights
  integer, intent(in) :: prc_index
  type(vamp_grid), dimension(:), intent(in), optional :: grids
  interface
    function func (xi, prc_index, weights, channel, grids) result (f)
      use kinds
      use vamp_grid_type !NODEP!
      real(kind=default), dimension(:), intent(in) :: xi
      integer, intent(in) :: prc_index
      real(kind=default), dimension(:), intent(in), optional :: weights
      integer, intent(in), optional :: channel
    end function func
  end interface
```

```

        type(vamp_grid), dimension(:), intent(in), optional :: grids
        real(kind=default) :: f
    end function func
end interface
real(kind=default) :: g
integer :: ch
g = 0.0
do ch = 1, size (weights)
    g = g + weights(ch) * func (x, prc_index, weights, ch, grids)
end do
end function vamp_sum_channels

```

5.2.7 Mapping

 This section is still under construction. The basic algorithm is in place, but the heuristics have not been developed yet.

The most naive approach is to use the rotation matrix R that diagonalizes the covariance C :

$$R_{ij} = (v_j)_i \quad (5.32)$$

where

$$Cv_j = \lambda_j v_j \quad (5.33)$$

with the eigenvalues $\{\lambda_j\}$ and eigenvectors $\{v_j\}$. Then

$$R^T C R = \text{diag}(\lambda_1, \dots) \quad (5.34)$$

After call `diagonalize_real_symmetric (cov, evals, evects)`, we have `evals(j) = λ_j` and `evects(:,j) = v_j` . This is equivalent with `evects(i,j) = R_{ij}` .

This approach will not work in high dimensions, however. In general, R will *not* leave most of the axes invariant, even if the covariance matrix is almost isotropic in these directions. In this case the benefit from the rotation is rather small and offset by the negative effects from the misalignment of the integration region.

A better strategy is to find the axis of the original coordinate system around which a rotation is most beneficial. There are two extreme cases:

- “pancake”: one eigenvalue much smaller than the others
- “cigar”: one eigenvalue much larger than the others

Actually, instead of rotating around a specific axis, we can as well diagonalize in a subspace. Empirically, rotation around an axis is better than diagonalizing in a two-dimensional subspace, but diagonalizing in a three-dimensional subspace can be even better.

```

124a  <Declaration of vamp procedures 76b>+≡
      public :: select_rotation_axis
      public :: select_rotation_subspace

124b  <Set iv to the index of the optimal eigenvector 124b>≡
      if (num_pancake > 0) then
        print *, "FORCED PANCAKE: ", num_pancake
        iv = sum (minloc (evals))
      else if (num_cigar > 0) then
        print *, "FORCED CIGAR: ", num_cigar
        iv = sum (maxloc (evals))
      else
        call more_pancake_than_cigar (evals, like_pancake)
        if (like_pancake) then
          iv = sum (minloc (evals))
        else
          iv = sum (maxloc (evals))
        end if
      end if

124c  <Implementation of vamp procedures 76d>+≡
      subroutine more_pancake_than_cigar (eval, yes_or_no)
        real(kind=default), dimension(:), intent(in) :: eval
        logical, intent(out) :: yes_or_no
        integer, parameter :: N_CL = 2
        real(kind=default), dimension(size(eval)) :: evals
        real(kind=default), dimension(N_CL) :: cluster_pos
        integer, dimension(N_CL,2) :: clusters
        evals = eval
        call sort (evals)
        call condense (evals, cluster_pos, clusters)
        print *, clusters(1,2) - clusters(1,1) + 1, "small EVs: ", &
          evals(clusters(1,1):clusters(1,2))
        print *, clusters(2,2) - clusters(2,1) + 1, "large EVs: ", &
          evals(clusters(2,1):clusters(2,2))
        if ((clusters(1,2) - clusters(1,1)) &
          < (clusters(2,2) - clusters(2,1))) then
          print *, " => PANCAKE!"
          yes_or_no = .true.
        else

```

```

        print *, " => CIGAR!"
        yes_or_no = .false.
    end if
end subroutine more_pancake_than_cigar

```

125a \langle Declaration of `vamp` procedures 76b $\rangle + \equiv$
`private :: more_pancake_than_cigar`

In both cases, we can rotate in the plane P_{ij} closest to eigenvector corresponding to the singled out eigenvalue. This plane is given by

$$\max_{i \neq i'} \sqrt{(v_j)_i^2 + (v_j)_{i'}^2} \quad (5.35)$$

which is simply found by looking for the two largest $|(v_j)_i|$:³

125b \langle Set `i(1)`, `i(2)` to the axes of the optimal plane 125b $\rangle \equiv$
`abs_evec = abs (evecs(:,iv))`
`i(1) = sum (maxloc (abs_evec))`
`abs_evec(i(1)) = -1.0`
`i(2) = sum (maxloc (abs_evec))`

The following is cute, but unfortunately broken, since it fails for degenerate eigenvalues:

125c \langle Set `i(1)`, `i(2)` to the axes of the optimal plane (broken!) 125c $\rangle \equiv$
`abs_evec = abs (evecs(:,iv))`
`i(1) = sum (maxloc (abs_evec))`
`i(2) = sum (maxloc (abs_evec, mask = abs_evec < abs_evec(i(1))))`

125d \langle Set `i(1)`, `i(2)` to the axes of the optimal plane 125b $\rangle + \equiv$
`print *, iv, evals(iv), " => ", evecs(:,iv)`
`print *, i(1), abs_evec(i(1)), ", ", i(2), abs_evec(i(2))`
`print *, i(1), evecs(i(1),iv), ", ", i(2), evecs(i(2),iv)`

125e \langle Get $\cos \theta$ and $\sin \theta$ from `evecs` 125e $\rangle \equiv$
`cos_theta = evecs(i(1),iv)`
`sin_theta = evecs(i(2),iv)`
`norm = 1.0 / sqrt (cos_theta**2 + sin_theta**2)`
`cos_theta = cos_theta * norm`
`sin_theta = sin_theta * norm`

³The `sum` intrinsic is a convenient Fortran90 trick for turning the rank-one array with one element returned by `maxloc` into its value. It has no semantic significance.

$$\hat{R}(\theta; i, j) = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \cos \theta & \cdots & -\sin \theta & \\ & & \vdots & 1 & \vdots & \\ & & \sin \theta & \cdots & \cos \theta & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix} \quad (5.36)$$

126a \langle Construct $\hat{R}(\theta; i, j)$ 126a $\rangle \equiv$

```
call unit (r)
r(i(1),i) = (/ cos_theta, - sin_theta /)
r(i(2),i) = (/ sin_theta, cos_theta /)
```

126b \langle Implementation of vamp procedures 76d $\rangle + \equiv$

```
subroutine select_rotation_axis (cov, r, pancake, cigar)
  real(kind=default), dimension(:, :), intent(in) :: cov
  real(kind=default), dimension(:, :), intent(out) :: r
  integer, intent(in), optional :: pancake, cigar
  integer :: num_pancake, num_cigar
  logical :: like_pancake
  real(kind=default), dimension(size(cov,dim=1),size(cov,dim=2)) :: evecs
  real(kind=default), dimension(size(cov,dim=1)) :: evals, abs_evec
  integer :: iv
  integer, dimension(2) :: i
  real(kind=default) :: cos_theta, sin_theta, norm
   $\langle$ Handle optional pancake and cigar 126c $\rangle$ 
  call diagonalize_real_symmetric (cov, evals, evecs)
   $\langle$ Set iv to the index of the optimal eigenvector 124b $\rangle$ 
   $\langle$ Set i(1), i(2) to the axes of the optimal plane 125b $\rangle$ 
   $\langle$ Get cos  $\theta$  and sin  $\theta$  from evecs 125e $\rangle$ 
   $\langle$ Construct  $\hat{R}(\theta; i, j)$  126a $\rangle$ 
end subroutine select_rotation_axis
```

126c \langle Handle optional pancake and cigar 126c $\rangle \equiv$

```
if (present (pancake)) then
  num_pancake = pancake
else
  num_pancake = -1
endif
if (present (cigar)) then
  num_cigar = cigar
else
  num_cigar = -1
endif
```

Here's a less efficient version that can be easily generalized to more than two dimension, however:

127a *<Implementation of vamp procedures 76d>+≡*

```
subroutine select_subspace_explicit (cov, r, subspace)
  real(kind=default), dimension(:,:), intent(in) :: cov
  real(kind=default), dimension(:,:), intent(out) :: r
  integer, dimension(:), intent(in) :: subspace
  real(kind=default), dimension(size(subspace)) :: eval_sub
  real(kind=default), dimension(size(subspace),size(subspace)) :: &
    cov_sub, evec_sub
  cov_sub = cov(subspace,subspace)
  call diagonalize_real_symmetric (cov_sub, eval_sub, evec_sub)
  call unit (r)
  r(subspace,subspace) = evec_sub
end subroutine select_subspace_explicit
```

127b *<Implementation of vamp procedures 76d>+≡*

```
subroutine select_subspace_guess (cov, r, ndim, pancake, cigar)
  real(kind=default), dimension(:,:), intent(in) :: cov
  real(kind=default), dimension(:,:), intent(out) :: r
  integer, intent(in) :: ndim
  integer, intent(in), optional :: pancake, cigar
  integer :: num_pancake, num_cigar
  logical :: like_pancake
  real(kind=default), dimension(size(cov,dim=1),size(cov,dim=2)) :: evecs
  real(kind=default), dimension(size(cov,dim=1)) :: evals, abs_evec
  integer :: iv, i
  integer, dimension(ndim) :: subspace
  <Handle optional pancake and cigar 126c>
  call diagonalize_real_symmetric (cov, evals, evecs)
  <Set iv to the index of the optimal eigenvector 124b>
  <Set subspace to the axes of the optimal plane 127c>
  call select_subspace_explicit (cov, r, subspace)
end subroutine select_subspace_guess
```

127c *<Set subspace to the axes of the optimal plane 127c>≡*

```
abs_evec = abs (evecs(:,iv))
subspace(1) = sum (maxloc (abs_evec))
do i = 2, ndim
  abs_evec(subspace(i-1)) = -1.0
  subspace(i) = sum (maxloc (abs_evec))
end do
```

127d *<Interfaces of vamp procedures 93b>+≡*

```
interface select_rotation_subspace
```



```

        module procedure select_subspace_explicit, select_subspace_guess
    end interface

128a  <Declaration of vamp procedures 76b>+≡
        private :: select_subspace_explicit
        private :: select_subspace_guess

128b  <Declaration of vamp procedures 76b>+≡
        public :: vamp_print_covariance

128c  <Implementation of vamp procedures 76d>+≡
        subroutine vamp_print_covariance (cov)
            real(kind=default), dimension(:,,:), intent(in) :: cov
            real(kind=default), dimension(size(cov,dim=1)) :: &
                evals, abs_evals, tmp
            real(kind=default), dimension(size(cov,dim=1),size(cov,dim=2)) :: &
                evecs, abs_evecs
            integer, dimension(size(cov,dim=1)) :: idx
            integer :: i, i_max, j
            i_max = size (evals)
            call diagonalize_real_symmetric (cov, evals, evecs)
            call sort (evals, evecs)
            abs_evals = abs (evals)
            abs_evecs = abs (evecs)
            print "(1X,A78)", repeat ("-", 78)
            print "(1X,A)", "Eigenvalues and eigenvectors:"
            print "(1X,A78)", repeat ("-", 78)
            do i = 1, i_max
                print "(1X,I2,A1,1X,E11.4,1X,A1,10(1X,F5.2)/,18X))", &
                    i, ":", evals(i), "|", evecs(:,i)
            end do
            print "(1X,A78)", repeat ("-", 78)
            print "(1X,A)", "Approximate subspaces:"
            print "(1X,A78)", repeat ("-", 78)
            do i = 1, i_max
                idx = (/ (j, j=1,i_max) /)
                tmp = abs_evecs(:,i)
                call sort (tmp, idx, reverse = .true.)
                print "(1X,I2,A1,1X,E11.4,1X,A1,10(1X,I5))", &
                    i, ":", evals(i), "|", idx(1:min(10,size(idx)))
                print "(17X,A1,10(1X,F5.2))", &
                    "|", evecs(idx(1:min(10,size(idx))),i)
            end do
            print "(1X,A78)", repeat ("-", 78)
        end subroutine vamp_print_covariance


```

Condensing Eigenvalues

In order to decide whether we have a “pancake” or a “cigar”, we have to classify the eigenvalues of the covariance matrix. We do this by condensing the n_{dim} eigenvalues into $n_{\text{cl}} \ll n_{\text{dim}}$ clusters.

129a \langle Declaration of **vamp** procedures 76b $\rangle \equiv$
`! private :: condense`
`public :: condense`

The rough description is as follows: in each step, combine the nearest neighbours (according to an appropriate metric) to form a smaller set. This is an extremely simplified, discretized modeling of molecules condensing under the influence of some potential.

 If there’s not a clean separation, this algorithm is certainly chaotic and we need to apply some form of damping!

129b \langle Initialize clusters 129b $\rangle \equiv$
`cl_pos = x`
`cl_num = size (cl_pos)`
`cl = spread ((/ (i, i=1,cl_num) /), dim = 2, ncopies = 2)`

It appears that the logarithmic metric

$$d_0(x, y) = \left| \log \left(\frac{x}{y} \right) \right| \quad (5.37a)$$


performs better than the linear metric

$$d_1(x, y) = |x - y| \quad (5.37b)$$

since the latter won’t separate very small eigenvalues from the bulk. Another option is

$$d_\alpha(x, y) = |x^\alpha - y^\alpha| \quad (5.37c)$$

with $\alpha \neq 1$, in particular $\alpha \approx -1$. I haven’t studied it yet, though.

 but I should perform more empirical studies to determine whether the logarithmic or the linear metric is more appropriate in realistic cases.

129c \langle Join closest clusters 129c $\rangle \equiv$
`if (linear_metric) then`
`gap = sum (minloc (cl_pos(2:cl_num) - cl_pos(1:cl_num-1)))`
`else`

```

        gap = sum (minloc (cl_pos(2:cl_num) / cl_pos(1:cl_num-1)))
    end if
    wgt0 = cl(gap,2) - cl(gap,1) + 1
    wgt1 = cl(gap+1,2) - cl(gap+1,1) + 1
    cl_pos(gap) = (wgt0 * cl_pos(gap) + wgt1 * cl_pos(gap+1)) / (wgt0 + wgt1)
    cl(gap,2) = cl(gap+1,2)

```

130a *Join closest clusters 129c*+≡

```

    cl_pos(gap+1:cl_num-1) = cl_pos(gap+2:cl_num)
    cl(gap+1:cl_num-1,:) = cl(gap+2:cl_num,:)

```

130b *Implementation of vamp procedures 76d*+≡

```

subroutine condense (x, cluster_pos, clusters, linear)
    real(kind=default), dimension(:), intent(in) :: x
    real(kind=default), dimension(:), intent(out) :: cluster_pos
    integer, dimension(:,:), intent(out) :: clusters
    logical, intent(in), optional :: linear
    logical :: linear_metric
    real(kind=default), dimension(size(x)) :: cl_pos
    real(kind=default) :: wgt0, wgt1
    integer :: cl_num
    integer, dimension(size(x),2) :: cl
    integer :: i, gap
    linear_metric = .false.
    if (present (linear)) then
        linear_metric = linear
    end if
    Initialize clusters 129b
    do cl_num = size (cl_pos), size (cluster_pos) + 1, -1
        Join closest clusters 129c
        print *, cl_num, ": action = ", condense_action (x, cl)
    end do
    cluster_pos = cl_pos(1:cl_num)
    clusters = cl(1:cl_num,:)
end subroutine condense

```

130c *Declaration of vamp procedures 76b*+≡

```

! private :: condense_action
public :: condense_action

```

$$S = \sum_{c \in \text{clusters}} \text{var}^{\frac{\alpha}{2}}(c) \quad (5.38)$$

130d *Implementation of vamp procedures 76d*+≡

```

function condense_action (positions, clusters) result (s)
    real(kind=default), dimension(:), intent(in) :: positions

```

```

integer, dimension(:,,:), intent(in) :: clusters
real(kind=default) :: s
integer :: i
integer, parameter :: POWER = 2
s = 0
do i = 1, size (clusters, dim = 1)
    s = s + standard_deviation (positions(clusters(i,1) &
                                         :clusters(i,2))) ** POWER
end do
end function condense_action
131 <ctest.f90 131>≡
program ctest
    use kinds
    use utils
    use vamp_stat
    use tao_random_numbers
    use vamp
    implicit none
    integer, parameter :: N = 16, NC = 2
    real(kind=default), dimension(N) :: eval
    real(kind=default), dimension(NC) :: cluster_pos
    integer, dimension(NC,2) :: clusters
    integer :: i
    call tao_random_number (eval)
    call sort (eval)
    print *, eval
    eval(1:N/2) = 0.95*eval(1:N/2)
    eval(N/2+1:N) = 1.0 - 0.95*(1.0 - eval(N/2+1:N))
    print *, eval
    call condense (eval, cluster_pos, clusters, linear=.true.)
    do i = 1, NC
        print "(I2,A,F5.2,A,I2,A,I2,A,A,F5.2,A,F5.2,A,32F5.2)", &
            i, ":", cluster_pos(i), &
            " [", clusters(i,1), "-", clusters(i,2), "]", &
            " [", eval(clusters(i,1)), " - ", eval(clusters(i,2)), "]", &
            eval(clusters(i,1)+1:clusters(i,2)) &
            - eval(clusters(i,1):clusters(i,2)-1)
        print *, average (eval(clusters(i,1):clusters(i,2))), "+/-", &
            standard_deviation (eval(clusters(i,1):clusters(i,2)))
    end do
end program ctest

```

5.2.8 Event Generation

Automagically adaptive tools are not always appropriate for unweighted event generation, but we can give it a try.

132a *⟨Declaration of vamp procedures 76b⟩*+≡
`public :: vamp_next_event`

132b *⟨Interfaces of vamp procedures 93b⟩*+≡
`interface vamp_next_event`
`module procedure vamp_next_event_single, vamp_next_event_multi`
`end interface`

132c *⟨Declaration of vamp procedures 76b⟩*+≡
`private :: vamp_next_event_single, vamp_next_event_multi`

Both event generation routines operate in two modes, depending on whether the optional argument `weight` is present.

132d *⟨Implementation of vamp procedures 76d⟩*+≡
`pure subroutine vamp_next_event_single &`
`(x, rng, g, func, prc_index, &`
`weight, channel, weights, grids, exc)`
`real(kind=default), dimension(:), intent(out) :: x`
`type(tao_random_state), intent(inout) :: rng`
`type(vamp_grid), intent(inout) :: g`
`real(kind=default), intent(out), optional :: weight`
`integer, intent(in) :: prc_index`
`integer, intent(in), optional :: channel`
`real(kind=default), dimension(:), intent(in), optional :: weights`
`type(vamp_grid), dimension(:), intent(in), optional :: grids`
`type(exception), intent(inout), optional :: exc`
⟨Interface declaration for func 22⟩
`character(len=*), parameter :: FN = "vamp_next_event_single"`
`real(kind=default), dimension(size(g%div)):: wgts`
`real(kind=default), dimension(size(g%div)):: r`
`integer, dimension(size(g%div)):: ia`
`real(kind=default) :: f, wgt`
`real(kind=default) :: r0`
`rejection: do`
`⟨Choose a x and calculate f(x) 133a⟩`
`if (present (weight)) then`
`⟨Unconditionally accept weighted event 133b⟩`
`else`
`⟨Maybe accept unweighted event 133c⟩`
`end if`
`end do rejection`

```

end subroutine vamp_next_event_single

133a <Choose a x and calculate f(x) 133a>≡
  call tao_random_number (rng, r)
  call inject_division_short (g%div, real(r, kind=default), x, ia, wgts)
  wgt = g%jacobi * product (wgts)
  wgt = g%calls * wgt ! the calling procedure will divide by #calls
  if (associated (g%map)) then
    x = matmul (g%map, x)
  end if
  <f = wgt * func (x, weights, channel), iff x inside true_domain 87a>
  ! call record_efficiency (g%div, ia, f/g%f_max)

133b <Unconditionally accept weighted event 133b>≡
  weight = f
  exit rejection

133c <Maybe accept unweighted event 133c>≡
  if (f > g%f_max) then
    g%f_max = f
    call raise_exception (exc, EXC_WARN, FN, "weight > 1")
    exit rejection
  end if
  call tao_random_number (rng, r0)
  if (r0 * g%f_max <= f) then
    exit rejection
  end if

```

We know that `g%weights` are normalized: `sum (g%weights) == 1.0`. The basic formula for multi channel sampling is

$$f(x) = \sum_i \alpha_i g_i(x) w(x) \quad (5.39)$$

with $w(x) = f(x)/g(x) = f(x)/\sum_i \alpha_i g_i(x)$ and $\sum_i \alpha_i = 1$. The non-trivial problem is that the adaptive grid is different in each channel, so we can't just reject on $w(x)$.

```

133d <Implementation of vamp procedures 76d>+≡
  pure subroutine vamp_next_event_multi &
    (x, rng, g, func, prc_index, phi, weight, excess, exc)
    real(kind=default), dimension(:), intent(out) :: x
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grids), intent(inout) :: g
    integer, intent(in) :: prc_index
    real(kind=default), intent(out), optional :: weight
    real(kind=default), intent(out), optional :: excess

```

```

type(exception), intent(inout), optional :: exc
<Interface declaration for func 22>
<Interface declaration for phi 31a>
character(len=*), parameter :: FN = "vamp_next_event_multi"
real(kind=default), dimension(size(x)) :: xi
real(kind=default) :: r, wgt
real(kind=default), dimension(size(g%weights)) :: weights
integer :: channel
<weights:  $\alpha_i \rightarrow w_{\max,i}\alpha_i$  134a>
rejection: do
  <Select channel from weights 134b>
  call vamp_next_event_single &
    (xi, rng, g%grids(channel), func, prc_index, wgt, &
     channel, g%weights, g%grids, exc)
  if (present (weight)) then
    <Unconditionally accept weighted multi channel event 135a>
  else
    <Maybe accept unweighted multi channel event 135b>
  end if
end do rejection
x = phi (xi, channel)
end subroutine vamp_next_event_multi

```

We can either reject with the weights

$$\frac{w_i(x)}{\max_i \max_x w_i(x)} \quad (5.40)$$

after using the apriori weights α_i to select a channel i or we can reject with the weights

$$\frac{w_i(x)}{\max_x w_i(x)} \quad (5.41)$$

after using the apriori weights $\alpha_i(\max_x w_i(x))/(\max_i \max_x w_i(x))$. The latter method is more efficient if the $\max_x w_i(x)$ have a wide spread.

```

134a <weights:  $\alpha_i \rightarrow w_{\max,i}\alpha_i$  134a>≡
      if (any (g%grids%f_max > 0)) then
        weights = g%weights * g%grids%f_max
      else
        weights = g%weights
      end if
      weights = weights / sum (weights)
134b <Select channel from weights 134b>≡
      call tao_random_number (rng, r)

```

```

select_channel: do channel = 1, size (g%weights)
  r = r - weights(channel)
  if (r <= 0.0) then
    exit select_channel
  end if
end do select_channel
channel = min (channel, size (g%weights)) ! for  $r = 1$  and rounding errors

135a <Unconditionally accept weighted multi channel event 135a>≡
  weight = wgt * g%weights(channel) / weights(channel)
  exit rejection

135b <Maybe accept unweighted multi channel event 135b>≡
  if (wgt > g%grids(channel)%f_max) then
    if (present(excess)) then
      excess = wgt/g%grids(channel)%f_max - 1
    else
      ! call raise_exception (exc, EXC_WARN, FN, "weight > 1")
      print *, "weight > 1 (", wgt/g%grids(channel)%f_max, &
        & ") in channel ", channel

      end if
    ! exit rejection
  else
    if (present(excess)) excess = 0
  end if
  call tao_random_number (rng, r)
  if (r * g%grids(channel)%f_max <= wgt) then
    exit rejection
  end if

135c <Maybe accept unweighted multi channel event (old version) 135c>≡
  if (wgt > g%grids(channel)%f_max) then
    g%grids(channel)%f_max = wgt
    <weights:  $\alpha_i \rightarrow w_{\max,i}\alpha_i$  134a>
    call raise_exception (exc, EXC_WARN, FN, "weight > 1")
    exit rejection
  end if
  call tao_random_number (rng, r)
  if (r * g%grids(channel)%f_max <= wgt) then
    exit rejection
  end if

```

Using `vamp_sample_grid (g, ...)` to warm up the grid `g` has a somewhat subtle problem: the minimum and maximum weights `g%f_min` and `g%f_max` refer to the grid *before* the final refinement. One could require an additional

vamp_sample_grid0 (g, ...), but users are likely to forget such technical details. A better solution is a wrapper vamp_warmup_grid (g, ...) that drops the final refinement transparently.

136a *<Declaration of vamp procedures 76b>+≡*

```
public :: vamp_warmup_grid, vamp_warmup_grids
```

136b *<Implementation of vamp procedures 76d>+≡*

```
pure subroutine vamp_warmup_grid &
  (rng, g, func, prc_index, iterations, exc, history)
  type(tao_random_state), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: prc_index
  integer, intent(in) :: iterations
  type(exception), intent(inout), optional :: exc
  type(vamp_history), dimension(:), intent(inout), optional :: history
  <Interface declaration for func 22>
  call vamp_sample_grid &
    (rng, g, func, prc_index, &
      iterations - 1, exc = exc, history = history)
  call vamp_sample_grid0 (rng, g, func, prc_index, exc = exc)
end subroutine vamp_warmup_grid
```



WHERE ... END WHERE alert!

136c *<Implementation of vamp procedures 76d>+≡*

```
pure subroutine vamp_warmup_grids &
  (rng, g, func, prc_index, iterations, history, histories, exc)
  type(tao_random_state), intent(inout) :: rng
  type(vamp_grids), intent(inout) :: g
  integer, intent(in) :: prc_index
  integer, intent(in) :: iterations
  type(vamp_history), dimension(:), intent(inout), optional :: history
  type(vamp_history), dimension(:, :), intent(inout), optional :: histories
  type(exception), intent(inout), optional :: exc
  <Interface declaration for func 22>
  integer :: ch
  logical, dimension(size(g%grids)) :: active
  real(kind=default), dimension(size(g%grids)) :: weights
  active = (g%num_calls >= 2)
  where (active)
    weights = g%num_calls
  elsewhere
    weights = 0.0
  end where
```

```

weights = weights / sum (weights)
call vamp_sample_grids (rng, g, func, prc_index, iterations - 1, &
                        exc = exc, history = history, histories = histories)
do ch = 1, size (g%grids)
  if (g%grids(ch)%num_calls >= 2) then
    call vamp_sample_grid0 &
      (rng, g%grids(ch), func, prc_index, &
       ch, weights, g%grids, exc = exc)
  end if
end do
end subroutine vamp_warmup_grids

```

5.2.9 Convenience Routines

- 137a *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_integrate
private :: vamp_integrate_grid, vamp_integrate_region

```
- 137b *<Interfaces of vamp procedures 93b>+≡*

```

interface vamp_integrate
  module procedure vamp_integrate_grid, vamp_integrate_region
end interface

```
- 137c *<Implementation of vamp procedures 76d>+≡*

```

pure subroutine vamp_integrate_grid &
  (rng, g, func, prc_index, calls, integral, std_dev, avg_chi2, num_div, &
   stratified, quadrupole, accuracy, exc, history)
type(tao_random_state), intent(inout) :: rng
type(vamp_grid), intent(inout) :: g
integer, intent(in) :: prc_index
integer, dimension(:,:), intent(in) :: calls
real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
real(kind=default), intent(in), optional :: accuracy
type(exception), intent(inout), optional :: exc
type(vamp_history), dimension(:), intent(inout), optional :: history
<Interface declaration for func 22>
character(len=*), parameter :: FN = "vamp_integrate_grid"
integer :: step, last_step, it
last_step = size (calls, dim = 2)
it = 1
do step = 1, last_step - 1
  call vamp_discard_integral (g, calls(2,step), num_div, &

```

```

                                stratified, quadrupole, exc = exc)
    call vamp_sample_grid (rng, g, func, prc_index, calls(1,step), &
                           exc = exc, history = history(it:))
    <Bail out if exception exc raised 96d>
    it = it + calls(1,step)
end do
call vamp_discard_integral (g, calls(2,last_step), exc = exc)
call vamp_sample_grid (rng, g, func, prc_index, calls(1,last_step), &
                       integral, std_dev, avg_chi2, accuracy, exc = exc, &
                       history = history(it:))
end subroutine vamp_integrate_grid
138 <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_integrate_region &
    (rng, region, func, prc_index, calls, &
     integral, std_dev, avg_chi2, num_div, &
     stratified, quadrupole, accuracy, map, covariance, exc, history)
type(tao_random_state), intent(inout) :: rng
real(kind=default), dimension(:,:), intent(in) :: region
integer, intent(in) :: prc_index
integer, dimension(:,:), intent(in) :: calls
real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
real(kind=default), intent(in), optional :: accuracy
real(kind=default), dimension(:,:), intent(in), optional :: map
real(kind=default), dimension(:,:), intent(out), optional :: covariance
type(exception), intent(inout), optional :: exc
type(vamp_history), dimension(:), intent(inout), optional :: history
<Interface declaration for func 22>
character(len=*), parameter :: FN = "vamp_integrate_region"
type(vamp_grid) :: g
call vamp_create_grid &
    (g, region, calls(2,1), num_div, &
     stratified, quadrupole, present (covariance), map, exc)
call vamp_integrate_grid &
    (rng, g, func, prc_index, calls, &
     integral, std_dev, avg_chi2, num_div, &
     accuracy = accuracy, exc = exc, history = history)
if (present (covariance)) then
    covariance = vamp_get_covariance (g)
end if
call vamp_delete_grid (g)
end subroutine vamp_integrate_region

```

```

139a <Declaration of vamp procedures 76b>+≡
      public :: vamp_integratex
      private :: vamp_integratex_region

139b <Interfaces of vamp procedures 93b>+≡
      interface vamp_integratex
        module procedure vamp_integratex_region
      end interface

139c <Implementation of vamp procedures 76d>+≡
      subroutine vamp_integratex_region &
        (rng, region, func, prc_index, calls, integral, std_dev, avg_chi2, &
         num_div, stratified, quadrupole, accuracy, pancake, cigar, &
         exc, history)
        type(tao_random_state), intent(inout) :: rng
        real(kind=default), dimension(:,:), intent(in) :: region
        integer, intent(in) :: prc_index
        integer, dimension(:,:,:), intent(in) :: calls
        real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
        integer, dimension(:), intent(in), optional :: num_div
        logical, intent(in), optional :: stratified, quadrupole
        real(kind=default), intent(in), optional :: accuracy
        integer, intent(in), optional :: pancake, cigar
        type(exception), intent(inout), optional :: exc
        type(vamp_history), dimension(:), intent(inout), optional :: history
        <Interface declaration for func 22>
        real(kind=default), dimension(size(region,dim=2)) :: eval
        real(kind=default), dimension(size(region,dim=2),size(region,dim=2)) :: evec
        type(vamp_grid) :: g
        integer :: step, last_step, it
        it = 1
        call vamp_create_grid &
          (g, region, calls(2,1,1), num_div, &
           stratified, quadrupole, covariance = .true., exc = exc)
        call vamp_integrate_grid &
          (rng, g, func, prc_index, calls(:,:,:), num_div = num_div, &
           exc = exc, history = history(it:))
        <Bail out if exception exc raised 96d>
        it = it + sum (calls(1,:,:), 1)
        last_step = size (calls, dim = 3)
        do step = 2, last_step - 1
          call diagonalize_real_symmetric (vamp_get_covariance(g), eval, evec)
          call sort (eval, evec)
          call select_rotation_axis (vamp_get_covariance(g), evec, pancake, cigar)
          call vamp_delete_grid (g)
        end do
      end subroutine

```

```

    call vamp_create_grid &
      (g, region, calls(2,1,step), num_div, stratified, quadrupole, &
        covariance = .true., map = evec, exc = exc)
    call vamp_integrate_grid &
      (rng, g, func, prc_index, calls(:, :, step), num_div = num_div, &
        exc = exc, history = history(it:))
    <Bail out if exception exc raised 96d>
    it = it + sum (calls(1, :, step))
  end do
  call diagonalize_real_symmetric (vamp_get_covariance(g), eval, evec)
  call sort (eval, evec)
  call select_rotation_axis (vamp_get_covariance(g), evec, pancake, cigar)
  call vamp_delete_grid (g)
  call vamp_create_grid &
    (g, region, calls(2,1,last_step), num_div, stratified, quadrupole, &
      covariance = .true., map = evec, exc = exc)
  call vamp_integrate_grid &
    (rng, g, func, prc_index, calls(:, :, last_step), &
      integral, std_dev, avg_chi2, &
      num_div = num_div, exc = exc, history = history(it:))
  call vamp_delete_grid (g)
end subroutine vamp_integratex_region

```

5.2.10 I/O

140a *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_write_grid
private :: write_grid_unit, write_grid_name
public :: vamp_read_grid
private :: read_grid_unit, read_grid_name
public :: vamp_write_grids
private :: write_grids_unit, write_grids_name
public :: vamp_read_grids
private :: read_grids_unit, read_grids_name

```

140b *<Declaration of vamp procedures 76b>+≡*

```

public :: vamp_read_grids_raw
private :: read_grids_raw_unit, read_grids_raw_name
public :: vamp_read_grid_raw
private :: read_grid_raw_unit, read_grid_raw_name
public :: vamp_write_grids_raw
private :: write_grids_raw_unit, write_grids_raw_name
public :: vamp_write_grid_raw
private :: write_grid_raw_unit, write_grid_raw_name

```

141a *<Interfaces of vamp procedures 93b>+≡*

```
interface vamp_write_grid
  module procedure write_grid_unit, write_grid_name
end interface
interface vamp_read_grid
  module procedure read_grid_unit, read_grid_name
end interface
interface vamp_write_grids
  module procedure write_grids_unit, write_grids_name
end interface
interface vamp_read_grids
  module procedure read_grids_unit, read_grids_name
end interface
```

141b *<Interfaces of vamp procedures 93b>+≡*

```
interface vamp_write_grid_raw
  module procedure write_grid_raw_unit, write_grid_raw_name
end interface
interface vamp_read_grid_raw
  module procedure read_grid_raw_unit, read_grid_raw_name
end interface
interface vamp_write_grids_raw
  module procedure write_grids_raw_unit, write_grids_raw_name
end interface
interface vamp_read_grids_raw
  module procedure read_grids_raw_unit, read_grids_raw_name
end interface
```

141c *<Implementation of vamp procedures 76d>+≡*

```
subroutine write_grid_unit (g, unit, write_integrals)
  type(vamp_grid), intent(in) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: write_integrals
  integer :: i, j
  write (unit = unit, fmt = descr_fmt) "begin type(vamp_grid) :: g"
  write (unit = unit, fmt = integer_fmt) "size (g%div) = ", size (g%div)
  write (unit = unit, fmt = integer_fmt) "num_calls = ", g%num_calls
  write (unit = unit, fmt = integer_fmt) "calls_per_cell = ", g%calls_per_cell
  write (unit = unit, fmt = logical_fmt) "stratified = ", g%stratified
  write (unit = unit, fmt = logical_fmt) "all_stratified = ", g%all_stratified
  write (unit = unit, fmt = logical_fmt) "quadrupole = ", g%quadrupole
  write (unit = unit, fmt = double_fmt) "mu(1) = ", g%mu(1)
  write (unit = unit, fmt = double_fmt) "mu(2) = ", g%mu(2)
  write (unit = unit, fmt = double_fmt) "sum_integral = ", g%sum_integral
  write (unit = unit, fmt = double_fmt) "sum_weights = ", g%sum_weights
```

```

write (unit = unit, fmt = double_fmt) "sum_chi2 = ", g%sum_chi2
write (unit = unit, fmt = double_fmt) "calls = ", g%calls
write (unit = unit, fmt = double_fmt) "dv2g = ", g%dv2g
write (unit = unit, fmt = double_fmt) "jacobi = ", g%jacobi
write (unit = unit, fmt = double_fmt) "f_min = ", g%f_min
write (unit = unit, fmt = double_fmt) "f_max = ", g%f_max
write (unit = unit, fmt = double_fmt) "mu_gi = ", g%mu_gi
write (unit = unit, fmt = double_fmt) "sum_mu_gi = ", g%sum_mu_gi
write (unit = unit, fmt = descr_fmt) "begin g%num_div"
do i = 1, size (g%div)
    write (unit = unit, fmt = integer_array_fmt) i, g%num_div(i)
end do
write (unit = unit, fmt = descr_fmt) "end g%num_div"
write (unit = unit, fmt = descr_fmt) "begin g%div"
do i = 1, size (g%div)
    call write_division (g%div(i), unit, write_integrals)
end do
write (unit = unit, fmt = descr_fmt) "end g%div"
if (associated (g%map)) then
    write (unit = unit, fmt = descr_fmt) "begin g%map"
    do i = 1, size (g%div)
        do j = 1, size (g%div)
            write (unit = unit, fmt = double_array2_fmt) i, j, g%map(i,j)
        end do
    end do
    write (unit = unit, fmt = descr_fmt) "end g%map"
else
    write (unit = unit, fmt = descr_fmt) "empty g%map"
end if
if (associated (g%mu_x)) then
    write (unit = unit, fmt = descr_fmt) "begin g%mu_x"
    do i = 1, size (g%div)
        write (unit = unit, fmt = double_array_fmt) i, g%mu_x(i)
        write (unit = unit, fmt = double_array_fmt) i, g%sum_mu_x(i)
        do j = 1, size (g%div)
            write (unit = unit, fmt = double_array2_fmt) i, j, g%mu_xx(i,j)
            write (unit = unit, fmt = double_array2_fmt) i, j, g%sum_mu_xx(i,j)
        end do
    end do
    write (unit = unit, fmt = descr_fmt) "end g%mu_x"
else
    write (unit = unit, fmt = descr_fmt) "empty g%mu_x"
end if

```

```

        write (unit = unit, fmt = descr_fmt) "end type(vamp_grid)"
    end subroutine write_grid_unit

143a  <Variables in vamp 78a>+≡
    character(len=*), parameter, private :: &
        descr_fmt =          "(1x,a)", &
        integer_fmt =        "(1x,a17,1x,i15)", &
        integer_array_fmt =  "(1x,i17,1x,i15)", &
        logical_fmt =        "(1x,a17,1x,l1)", &
        double_fmt =          "(1x,a17,1x,e30.22)", &
        double_array_fmt =   "(1x,i17,1x,e30.22)", &
        double_array2_fmt =   "(2(1x,i8),1x,e30.22)"

143b  <Implementation of vamp procedures 76d>+≡
    subroutine read_grid_unit (g, unit, read_integrals)
        type(vamp_grid), intent(inout) :: g
        integer, intent(in) :: unit
        logical, intent(in), optional :: read_integrals
        character(len=*), parameter :: FN = "vamp_read_grid"
        character(len=80) :: chdum
        integer :: ndim, i, j, idum, jdum
        read (unit = unit, fmt = descr_fmt) chdum
        read (unit = unit, fmt = integer_fmt) chdum, ndim
        <Insure that size (g%div) == ndim 144>
        call create_array_pointer (g%num_div, ndim)
        read (unit = unit, fmt = integer_fmt) chdum, g%num_calls
        read (unit = unit, fmt = integer_fmt) chdum, g%calls_per_cell
        read (unit = unit, fmt = logical_fmt) chdum, g%stratified
        read (unit = unit, fmt = logical_fmt) chdum, g%all_stratified
        read (unit = unit, fmt = logical_fmt) chdum, g%quadrupole
        read (unit = unit, fmt = double_fmt) chdum, g%mu(1)
        read (unit = unit, fmt = double_fmt) chdum, g%mu(2)
        read (unit = unit, fmt = double_fmt) chdum, g%sum_integral
        read (unit = unit, fmt = double_fmt) chdum, g%sum_weights
        read (unit = unit, fmt = double_fmt) chdum, g%sum_chi2
        read (unit = unit, fmt = double_fmt) chdum, g%calls
        read (unit = unit, fmt = double_fmt) chdum, g%dv2g
        read (unit = unit, fmt = double_fmt) chdum, g%jacobi
        read (unit = unit, fmt = double_fmt) chdum, g%f_min
        read (unit = unit, fmt = double_fmt) chdum, g%f_max
        read (unit = unit, fmt = double_fmt) chdum, g%mu_gi
        read (unit = unit, fmt = double_fmt) chdum, g%sum_mu_gi
        read (unit = unit, fmt = descr_fmt) chdum
        do i = 1, size (g%div)
            read (unit = unit, fmt = integer_array_fmt) idum, g%num_div(i)

```



```

end do
read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = descr_fmt) chdum
do i = 1, size (g%div)
    call read_division (g%div(i), unit, read_integrals)
end do
read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = descr_fmt) chdum
if (chdum == "begin g%map") then
    call create_array_pointer (g%map, (/ ndim, ndim /))
    do i = 1, size (g%div)
        do j = 1, size (g%div)
            read (unit = unit, fmt = double_array2_fmt) idum, jdum, g%map(i,j)
        end do
    end do
    read (unit = unit, fmt = descr_fmt) chdum
else
    <Insure that associated (g%map) == .false. 145a>
end if
read (unit = unit, fmt = descr_fmt) chdum
if (chdum == "begin g%mu_x") then
    call create_array_pointer (g%mu_x, ndim )
    call create_array_pointer (g%sum_mu_x, ndim)
    call create_array_pointer (g%mu_xx, (/ ndim, ndim /))
    call create_array_pointer (g%sum_mu_xx, (/ ndim, ndim /))
    do i = 1, size (g%div)
        read (unit = unit, fmt = double_array_fmt) idum, jdum, g%mu_x(i)
        read (unit = unit, fmt = double_array_fmt) idum, jdum, g%sum_mu_x(i)
        do j = 1, size (g%div)
            read (unit = unit, fmt = double_array2_fmt) &
                idum, jdum, g%mu_xx(i,j)
            read (unit = unit, fmt = double_array2_fmt) &
                idum, jdum, g%sum_mu_xx(i,j)
        end do
    end do
    read (unit = unit, fmt = descr_fmt) chdum
else
    <Insure that associated (g%mu_x) == .false. 145b>
end if
read (unit = unit, fmt = descr_fmt) chdum
end subroutine read_grid_unit

144 <Insure that size (g%div) == ndim 144>≡
    if (associated (g%div)) then

```

```

        if (size (g%div) /= ndim) then
            call delete_division (g%div)
            deallocate (g%div)
            allocate (g%div(ndim))
            call create_empty_division (g%div)
        end if
    else
        allocate (g%div(ndim))
        call create_empty_division (g%div)
    end if

145a  <Insure that associated (g%map) == .false. 145a>≡
        if (associated (g%map)) then
            deallocate (g%map)
        end if

145b  <Insure that associated (g%mu_x) == .false. 145b>≡
        if (associated (g%mu_x)) then
            deallocate (g%mu_x)
        end if
        if (associated (g%mu_xx)) then
            deallocate (g%mu_xx)
        end if
        if (associated (g%sum_mu_x)) then
            deallocate (g%sum_mu_x)
        end if
        if (associated (g%sum_mu_xx)) then
            deallocate (g%sum_mu_xx)
        end if

145c  <Implementation of vamp procedures 76d>+≡
        subroutine write_grid_name (g, name, write_integrals)
            type(vamp_grid), intent(inout) :: g
            character(len=*), intent(in) :: name
            logical, intent(in), optional :: write_integrals
            integer :: unit
            call find_free_unit (unit)
            open (unit = unit, action = "write", status = "replace", file = name)
            call write_grid_unit (g, unit, write_integrals)
            close (unit = unit)
        end subroutine write_grid_name

145d  <Implementation of vamp procedures 76d>+≡
        subroutine read_grid_name (g, name, read_integrals)
            type(vamp_grid), intent(inout) :: g
            character(len=*), intent(in) :: name

```

```

logical, intent(in), optional :: read_integrals
integer :: unit
call find_free_unit (unit)
open (unit = unit, action = "read", status = "old", file = name)
call read_grid_unit (g, unit, read_integrals)
close (unit = unit)
end subroutine read_grid_name

```

146a *<Implementation of vamp procedures 76d>+≡*

```

subroutine write_grids_unit (g, unit, write_integrals)
  type(vamp_grids), intent(in) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: write_integrals
  integer :: i
  write (unit = unit, fmt = descr_fmt) "begin type(vamp_grids) :: g"
  write (unit = unit, fmt = integer_fmt) "size (g%grids) = ", size (g%grids)
  write (unit = unit, fmt = double_fmt) "sum_integral = ", g%sum_integral
  write (unit = unit, fmt = double_fmt) "sum_weights = ", g%sum_weights
  write (unit = unit, fmt = double_fmt) "sum_chi2 = ", g%sum_chi2
  write (unit = unit, fmt = descr_fmt) "begin g%weights"
  do i = 1, size (g%grids)
    write (unit = unit, fmt = double_array_fmt) i, g%weights(i)
  end do
  write (unit = unit, fmt = descr_fmt) "end g%weights"
  write (unit = unit, fmt = descr_fmt) "begin g%num_calls"
  do i = 1, size (g%grids)
    write (unit = unit, fmt = integer_array_fmt) i, g%num_calls(i)
  end do
  write (unit = unit, fmt = descr_fmt) "end g%num_calls"
  write (unit = unit, fmt = descr_fmt) "begin g%grids"
  do i = 1, size (g%grids)
    call write_grid_unit (g%grids(i), unit, write_integrals)
  end do
  write (unit = unit, fmt = descr_fmt) "end g%grids"
  write (unit = unit, fmt = descr_fmt) "end type(vamp_grids)"
end subroutine write_grids_unit

```

146b *<Implementation of vamp procedures 76d>+≡*

```

subroutine read_grids_unit (g, unit, read_integrals)
  type(vamp_grids), intent(inout) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: read_integrals
  character(len=*), parameter :: FN = "vamp_read_grids"
  character(len=80) :: chdum
  integer :: i, nch, idum

```

```

read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = integer_fmt) chdum, nch
if (associated (g%grids)) then
  if (size (g%grids) /= nch) then
    call vamp_delete_grid (g%grids)
    deallocate (g%grids, g%weights, g%num_calls)
    allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
    call vamp_create_empty_grid (g%grids)
  end if
else
  allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
  call vamp_create_empty_grid (g%grids)
end if
read (unit = unit, fmt = double_fmt) chdum, g%sum_integral
read (unit = unit, fmt = double_fmt) chdum, g%sum_weights
read (unit = unit, fmt = double_fmt) chdum, g%sum_chi2
read (unit = unit, fmt = descr_fmt) chdum
do i = 1, nch
  read (unit = unit, fmt = double_array_fmt) idum, g%weights(i)
end do
read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = descr_fmt) chdum
do i = 1, nch
  read (unit = unit, fmt = integer_array_fmt) idum, g%num_calls(i)
end do
read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = descr_fmt) chdum
do i = 1, nch
  call read_grid_unit (g%grids(i), unit, read_integrals)
end do
read (unit = unit, fmt = descr_fmt) chdum
read (unit = unit, fmt = descr_fmt) chdum
end subroutine read_grids_unit

```

147 *⟨Implementation of vamp procedures 76d⟩+≡*

```

subroutine write_grids_name (g, name, write_integrals)
  type(vamp_grids), intent(inout) :: g
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: write_integrals
  integer :: unit
  call find_free_unit (unit)
  open (unit = unit, action = "write", status = "replace", file = name)
  call write_grids_unit (g, unit, write_integrals)
  close (unit = unit)

```

```

end subroutine write_grids_name

148a <Implementation of vamp procedures 76d>+≡
subroutine read_grids_name (g, name, read_integrals)
  type(vamp_grids), intent(inout) :: g
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: read_integrals
  integer :: unit
  call find_free_unit (unit)
  open (unit = unit, action = "read", status = "old", file = name)
  call read_grids_unit (g, unit, read_integrals)
  close (unit = unit)
end subroutine read_grids_name

148b <Implementation of vamp procedures 76d>+≡
subroutine write_grid_raw_unit (g, unit, write_integrals)
  type(vamp_grid), intent(in) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: write_integrals
  integer :: i, j
  write (unit = unit) MAGIC_GRID_BEGIN
  write (unit = unit) size (g%div)
  write (unit = unit) g%num_calls
  write (unit = unit) g%calls_per_cell
  write (unit = unit) g%stratified
  write (unit = unit) g%all_stratified
  write (unit = unit) g%quadrupole
  write (unit = unit) g%mu(1)
  write (unit = unit) g%mu(2)
  write (unit = unit) g%sum_integral
  write (unit = unit) g%sum_weights
  write (unit = unit) g%sum_chi2
  write (unit = unit) g%calls
  write (unit = unit) g%dv2g
  write (unit = unit) g%jacobi
  write (unit = unit) g%f_min
  write (unit = unit) g%f_max
  write (unit = unit) g%mu_gi
  write (unit = unit) g%sum_mu_gi
  do i = 1, size (g%div)
    write (unit = unit) g%num_div(i)
  end do
  do i = 1, size (g%div)
    call write_division_raw (g%div(i), unit, write_integrals)
  end do

```

```

if (associated (g%map)) then
  write (unit = unit) MAGIC_GRID_MAP
  do i = 1, size (g%div)
    do j = 1, size (g%div)
      write (unit = unit) g%map(i,j)
    end do
  end do
else
  write (unit = unit) MAGIC_GRID_EMPTY
end if
if (associated (g%mu_x)) then
  write (unit = unit) MAGIC_GRID_MU_X
  do i = 1, size (g%div)
    write (unit = unit) g%mu_x(i)
    write (unit = unit) g%sum_mu_x(i)
    do j = 1, size (g%div)
      write (unit = unit) g%mu_xx(i,j)
      write (unit = unit) g%sum_mu_xx(i,j)
    end do
  end do
else
  write (unit = unit) MAGIC_GRID_EMPTY
end if
write (unit = unit) MAGIC_GRID_END
end subroutine write_grid_raw_unit

```

149a *⟨Constants in vamp 149a⟩*≡

```

integer, parameter, private :: MAGIC_GRID = 22222222
integer, parameter, private :: MAGIC_GRID_BEGIN = MAGIC_GRID + 1
integer, parameter, private :: MAGIC_GRID_END = MAGIC_GRID + 2
integer, parameter, private :: MAGIC_GRID_EMPTY = MAGIC_GRID + 3
integer, parameter, private :: MAGIC_GRID_MAP = MAGIC_GRID + 4
integer, parameter, private :: MAGIC_GRID_MU_X = MAGIC_GRID + 5

```

149b *⟨Implementation of vamp procedures 76d⟩*+≡

```

subroutine read_grid_raw_unit (g, unit, read_integrals)
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: read_integrals
  character(len=*), parameter :: FN = "vamp_read_raw_grid"
  integer :: ndim, i, j, magic
  read (unit = unit) magic
  if (magic /= MAGIC_GRID_BEGIN) then
    print *, FN, " fatal: expecting magic ", MAGIC_GRID_BEGIN, &
      ", found ", magic
  end if
end subroutine read_grid_raw_unit

```

```

        stop
    end if
    read (unit = unit) ndim
    <Insure that size (g%div) == ndim 144>
    call create_array_pointer (g%num_div, ndim)
    read (unit = unit) g%num_calls
    read (unit = unit) g%calls_per_cell
    read (unit = unit) g%stratified
    read (unit = unit) g%all_stratified
    read (unit = unit) g%quadrupole
    read (unit = unit) g%mu(1)
    read (unit = unit) g%mu(2)
    read (unit = unit) g%sum_integral
    read (unit = unit) g%sum_weights
    read (unit = unit) g%sum_chi2
    read (unit = unit) g%calls
    read (unit = unit) g%dv2g
    read (unit = unit) g%jacobi
    read (unit = unit) g%f_min
    read (unit = unit) g%f_max
    read (unit = unit) g%mu_gi
    read (unit = unit) g%sum_mu_gi
    do i = 1, size (g%div)
        read (unit = unit) g%num_div(i)
    end do
    do i = 1, size (g%div)
        call read_division_raw (g%div(i), unit, read_integrals)
    end do
    read (unit = unit) magic
    if (magic == MAGIC_GRID_MAP) then
        call create_array_pointer (g%map, (/ ndim, ndim /))
        do i = 1, size (g%div)
            do j = 1, size (g%div)
                read (unit = unit) g%map(i,j)
            end do
        end do
    else if (magic == MAGIC_GRID_EMPTY) then
        <Insure that associated (g%map) == .false. 145a>
    else
        print *, FN, " fatal: expecting magic ", MAGIC_GRID_EMPTY, &
            " or ", MAGIC_GRID_MAP, ", found ", magic
        stop
    end if
end if

```

```

read (unit = unit) magic
if (magic == MAGIC_GRID_MU_X) then
  call create_array_pointer (g%mu_x, ndim )
  call create_array_pointer (g%sum_mu_x, ndim)
  call create_array_pointer (g%mu_xx, (/ ndim, ndim /))
  call create_array_pointer (g%sum_mu_xx, (/ ndim, ndim /))
  do i = 1, size (g%div)
    read (unit = unit) g%mu_x(i)
    read (unit = unit) g%sum_mu_x(i)
    do j = 1, size (g%div)
      read (unit = unit) g%mu_xx(i,j)
      read (unit = unit) g%sum_mu_xx(i,j)
    end do
  end do
else if (magic == MAGIC_GRID_EMPTY) then
  <Insure that associated (g%mu_x) == .false. 145b>
else
  print *, FN, " fatal: expecting magic ", MAGIC_GRID_EMPTY, &
    " or ", MAGIC_GRID_MU_X, ", found ", magic
  stop
end if
read (unit = unit) magic
if (magic /= MAGIC_GRID_END) then
  print *, FN, " fatal: expecting magic ", MAGIC_GRID_END, &
    " found ", magic
  stop
end if
end subroutine read_grid_raw_unit

```

151a *<Implementation of vamp procedures 76d>+≡*

```

subroutine write_grid_raw_name (g, name, write_integrals)
  type(vamp_grid), intent(inout) :: g
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: write_integrals
  integer :: unit
  call find_free_unit (unit)
  open (unit = unit, action = "write", status = "replace", &
    form = "unformatted", file = name)
  call write_grid_raw_unit (g, unit, write_integrals)
  close (unit = unit)
end subroutine write_grid_raw_name

```

151b *<Implementation of vamp procedures 76d>+≡*

```

subroutine read_grid_raw_name (g, name, read_integrals)
  type(vamp_grid), intent(inout) :: g

```



```

character(len=*), intent(in) :: name
logical, intent(in), optional :: read_integrals
integer :: unit
call find_free_unit (unit)
open (unit = unit, action = "read", status = "old", &
      form = "unformatted", file = name)
call read_grid_raw_unit (g, unit, read_integrals)
close (unit = unit)
end subroutine read_grid_raw_name

```

152a *⟨Implementation of vamp procedures 76d⟩*+≡

```

subroutine write_grids_raw_unit (g, unit, write_integrals)
  type(vamp_grids), intent(in) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: write_integrals
  integer :: i
  write (unit = unit) MAGIC_GRIDS_BEGIN
  write (unit = unit) size (g%grids)
  write (unit = unit) g%sum_integral
  write (unit = unit) g%sum_weights
  write (unit = unit) g%sum_chi2
  do i = 1, size (g%grids)
    write (unit = unit) g%weights(i)
  end do
  do i = 1, size (g%grids)
    write (unit = unit) g%num_calls(i)
  end do
  do i = 1, size (g%grids)
    call write_grid_raw_unit (g%grids(i), unit, write_integrals)
  end do
  write (unit = unit) MAGIC_GRIDS_END
end subroutine write_grids_raw_unit

```

152b *⟨Constants in vamp 149a⟩*+≡

```

integer, parameter, private :: MAGIC_GRIDS = 33333333
integer, parameter, private :: MAGIC_GRIDS_BEGIN = MAGIC_GRIDS + 1
integer, parameter, private :: MAGIC_GRIDS_END = MAGIC_GRIDS + 2

```

152c *⟨Implementation of vamp procedures 76d⟩*+≡

```

subroutine read_grids_raw_unit (g, unit, read_integrals)
  type(vamp_grids), intent(inout) :: g
  integer, intent(in) :: unit
  logical, intent(in), optional :: read_integrals
  character(len=*), parameter :: FN = "vamp_read_grids_raw"
  integer :: i, nch, magic

```

```

read (unit = unit) magic
if (magic /= MAGIC_GRIDS_BEGIN) then
    print *, FN, " fatal: expecting magic ", MAGIC_GRIDS_BEGIN, &
        " found ", magic
    stop
end if
read (unit = unit) nch
if (associated (g%grids)) then
    if (size (g%grids) /= nch) then
        call vamp_delete_grid (g%grids)
        deallocate (g%grids, g%weights, g%num_calls)
        allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
        call vamp_create_empty_grid (g%grids)
    end if
else
    allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
    call vamp_create_empty_grid (g%grids)
end if
read (unit = unit) g%sum_integral
read (unit = unit) g%sum_weights
read (unit = unit) g%sum_chi2
do i = 1, nch
    read (unit = unit) g%weights(i)
end do
do i = 1, nch
    read (unit = unit) g%num_calls(i)
end do
do i = 1, nch
    call read_grid_raw_unit (g%grids(i), unit, read_integrals)
end do
read (unit = unit) magic
if (magic /= MAGIC_GRIDS_END) then
    print *, FN, " fatal: expecting magic ", MAGIC_GRIDS_END, &
        " found ", magic
    stop
end if
end subroutine read_grids_raw_unit

```

153 *⟨Implementation of vamp procedures 76d⟩*+≡

```

subroutine write_grids_raw_name (g, name, write_integrals)
    type(vamp_grids), intent(inout) :: g
    character(len=*), intent(in) :: name
    logical, intent(in), optional :: write_integrals
    integer :: unit

```

```

    call find_free_unit (unit)
    open (unit = unit, action = "write", status = "replace", &
          form = "unformatted", file = name)
    call write_grids_raw_unit (g, unit, write_integrals)
    close (unit = unit)
end subroutine write_grids_raw_name

```

154a *⟨Implementation of vamp procedures 76d⟩+≡*

```

subroutine read_grids_raw_name (g, name, read_integrals)
  type(vamp_grids), intent(inout) :: g
  character(len=*), intent(in) :: name
  logical, intent(in), optional :: read_integrals
  integer :: unit
  call find_free_unit (unit)
  open (unit = unit, action = "read", status = "old", &
        form = "unformatted", file = name)
  call read_grids_raw_unit (g, unit, read_integrals)
  close (unit = unit)
end subroutine read_grids_raw_name

```

5.2.11 Marshaling

154b *⟨Declaration of vamp procedures 76b⟩+≡*

```

public :: vamp_marshall_grid_size, vamp_marshall_grid, vamp_unmarshal_grid

```

154c *⟨Implementation of vamp procedures 76d⟩+≡*

```

pure subroutine vamp_marshall_grid (g, ibuf, dbuf)
  type(vamp_grid), intent(in) :: g
  integer, dimension(:), intent(inout) :: ibuf
  real(kind=default), dimension(:), intent(inout) :: dbuf
  integer :: i, iwords, dwords, iidx, didx, ndim
  ndim = size (g%div)
  ibuf(1) = g%num_calls
  ibuf(2) = g%calls_per_cell
  ibuf(3) = ndim
  if (g%stratified) then
    ibuf(4) = 1
  else
    ibuf(4) = 0
  end if
  if (g%all_stratified) then
    ibuf(5) = 1
  else
    ibuf(5) = 0
  end if

```

```

end if
if (g%quadrupole) then
    ibuf(6) = 1
else
    ibuf(6) = 0
end if
dbuf(1:2) = g%mu
dbuf(3) = g%sum_integral
dbuf(4) = g%sum_weights
dbuf(5) = g%sum_chi2
dbuf(6) = g%calls
dbuf(7) = g%dv2g
dbuf(8) = g%jacobi
dbuf(9) = g%f_min
dbuf(10) = g%f_max
dbuf(11) = g%mu_gi
dbuf(12) = g%sum_mu_gi
ibuf(7:6+ndim) = g%num_div
iidx = 7 + ndim
didx = 13
do i = 1, ndim
    call marshal_division_size (g%div(i), iwords, dwords)
    ibuf(iidx) = iwords
    ibuf(iidx+1) = dwords
    iidx = iidx + 2
    call marshal_division (g%div(i), ibuf(iidx:iidx-1+iwords), &
                           dbuf(didx:didx-1+dwords))

    iidx = iidx + iwords
    didx = didx + dwords
end do
if (associated (g%map)) then
    ibuf(iidx) = 1
    dbuf(didx:didx-1+ndim**2) = reshape (g%map, (/ ndim**2 /))
    didx = didx + ndim**2
else
    ibuf(iidx) = 0
end if
iidx = iidx + 1
if (associated (g%mu_x)) then
    ibuf(iidx) = 1
    dbuf(didx:didx-1+ndim) = g%mu_x
    didx = didx + ndim
    dbuf(didx:didx-1+ndim) = g%sum_mu_x

```

```

        didx = didx + ndim
        dbuf(didx:didx-1+ndim**2) = reshape (g%mu_xx, (/ ndim**2 /))
        didx = didx + ndim**2
        dbuf(didx:didx-1+ndim**2) = reshape (g%sum_mu_xx, (/ ndim**2 /))
        didx = didx + ndim**2
    else
        ibuf(iidx) = 0
    end if
    iidx = iidx + 1
end subroutine vamp_marshall_grid

```

156a *⟨Implementation of vamp procedures 76d⟩+≡*

```

pure subroutine vamp_marshall_grid_size (g, iwords, dwords)
    type(vamp_grid), intent(in) :: g
    integer, intent(out) :: iwords, dwords
    integer :: i, ndim, iw, dw
    ndim = size (g%div)
    iwords = 6 + ndim
    dwords = 12
    do i = 1, ndim
        call marshal_division_size (g%div(i), iw, dw)
        iwords = iwords + 2 + iw
        dwords = dwords + dw
    end do
    iwords = iwords + 1
    if (associated (g%map)) then
        dwords = dwords + ndim**2
    end if
    iwords = iwords + 1
    if (associated (g%mu_x)) then
        dwords = dwords + 2 * (ndim + ndim**2)
    end if
end subroutine vamp_marshall_grid_size

```

156b *⟨Implementation of vamp procedures 76d⟩+≡*

```

pure subroutine vamp_unmarshal_grid (g, ibuf, dbuf)
    type(vamp_grid), intent(inout) :: g
    integer, dimension(:), intent(in) :: ibuf
    real(kind=default), dimension(:), intent(in) :: dbuf
    integer :: i, iwords, dwords, iidx, didx, ndim
    g%num_calls = ibuf(1)
    g%calls_per_cell = ibuf(2)
    ndim = ibuf(3)
    g%stratified = ibuf(4) /= 0
    g%all_stratified = ibuf(5) /= 0

```

```

g%quadrupole = ibuf(6) /= 0
g%mu = dbuf(1:2)
g%sum_integral = dbuf(3)
g%sum_weights = dbuf(4)
g%sum_chi2 = dbuf(5)
g%calls = dbuf(6)
g%dv2g = dbuf(7)
g%jacobi = dbuf(8)
g%f_min = dbuf(9)
g%f_max = dbuf(10)
g%mu_gi = dbuf(11)
g%sum_mu_gi = dbuf(12)
call copy_array_pointer (g%num_div, ibuf(7:6+ndim))
<Insure that size (g%div) == ndim 144>
iidx = 7 + ndim
didx = 13
do i = 1, ndim
    iwords = ibuf(iidx)
    dwords = ibuf(iidx+1)
    iidx = iidx + 2
    call unmarshal_division (g%div(i), ibuf(iidx:iidx-1+iwords), &
                             dbuf(didx:didx-1+dwords))

    iidx = iidx + iwords
    didx = didx + dwords
end do
if (ibuf(iidx) > 0) then
    call copy_array_pointer &
        (g%map, reshape (dbuf(didx:didx-1+ibuf(iidx)), (/ ndim, ndim /)))
    didx = didx + ibuf(iidx)
else
    <Insure that associated (g%map) == .false. 145a>
end if
iidx = iidx + 1
if (ibuf(iidx) > 0) then
    call copy_array_pointer (g%mu_x, dbuf(didx:didx-1+ndim))
    didx = didx + ndim
    call copy_array_pointer (g%sum_mu_x, dbuf(didx:didx-1+ndim))
    didx = didx + ndim
    call copy_array_pointer &
        (g%mu_xx, reshape (dbuf(didx:didx-1+ndim**2), (/ ndim, ndim /)))
    didx = didx + ndim**2
    call copy_array_pointer &
        (g%sum_mu_xx, reshape (dbuf(didx:didx-1+ndim**2), (/ ndim, ndim /)))

```

```

        didx = didx + ndim**2
    else
        <Insure that associated (g%mu_x) == .false. 145b>
    end if
    iidx = iidx + 1
end subroutine vamp_unmarshal_grid

158a <Declaration of vamp procedures 76b>+≡
    public :: vamp_marshal_history_size, vamp_marshal_history
    public :: vamp_unmarshal_history

158b <Implementation of vamp procedures 76d>+≡
    pure subroutine vamp_marshal_history (h, ibuf, dbuf)
        type(vamp_history), intent(in) :: h
        integer, dimension(:), intent(inout) :: ibuf
        real(kind=default), dimension(:), intent(inout) :: dbuf
        integer :: j, ndim, iidx, didx, iwords, dwords
        if (h%verbose .and. (associated (h%div))) then
            ndim = size (h%div)
        else
            ndim = 0
        end if
        ibuf(1) = ndim
        ibuf(2) = h%calls
        if (h%stratified) then
            ibuf(3) = 1
        else
            ibuf(3) = 0
        end if
        dbuf(1) = h%integral
        dbuf(2) = h%std_dev
        dbuf(3) = h%avg_integral
        dbuf(4) = h%avg_std_dev
        dbuf(5) = h%avg_chi2
        dbuf(6) = h%f_min
        dbuf(7) = h%f_max
        iidx = 4
        didx = 8
        do j = 1, ndim
            call marshal_div_history_size (h%div(j), iwords, dwords)
            ibuf(iidx) = iwords
            ibuf(iidx+1) = dwords
            iidx = iidx + 2
            call marshal_div_history (h%div(j), ibuf(iidx:iidx-1+iwords), &
                                     dbuf(didx:didx-1+dwords))
        end do
    end subroutine

```

```

        iidx = iidx + iwords
        didx = didx + dwords
    end do
end subroutine vamp_marshall_history

159a <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_marshall_history_size (h, iwords, dwords)
    type(vamp_history), intent(in) :: h
    integer, intent(out) :: iwords, dwords
    integer :: i, ndim, iw, dw
    if (h%verbose .and. (associated (h%div))) then
        ndim = size (h%div)
    else
        ndim = 0
    end if
    iwords = 3
    dwords = 7
    do i = 1, ndim
        call marshal_div_history_size (h%div(i), iw, dw)
        iwords = iwords + 2 + iw
        dwords = dwords + dw
    end do
end subroutine vamp_marshall_history_size

159b <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_unmarshal_history (h, ibuf, dbuf)
    type(vamp_history), intent(inout) :: h
    integer, dimension(:), intent(in) :: ibuf
    real(kind=default), dimension(:), intent(in) :: dbuf
    integer :: j, ndim, iidx, didx, iwords, dwords
    ndim = ibuf(1)
    h%calls = ibuf(2)
    h%stratified = ibuf(3) /= 0
    h%integral = dbuf(1)
    h%std_dev = dbuf(2)
    h%avg_integral = dbuf(3)
    h%avg_std_dev = dbuf(4)
    h%avg_chi2 = dbuf(5)
    h%f_min = dbuf(6)
    h%f_max = dbuf(7)
    if (ndim > 0) then
        if (associated (h%div)) then
            if (size (h%div) /= ndim) then
                deallocate (h%div)
                allocate (h%div(ndim))
            end if
        end if
    end if

```



```

        end if
    else
        allocate (h%div(ndim))
    end if
    iidx = 4
    didx = 8
    do j = 1, ndim
        iwords = ibuf(iidx)
        dwords = ibuf(iidx+1)
        iidx = iidx + 2
        call unmarshal_div_history (h%div(j), ibuf(iidx:iidx-1+iwords), &
                                   dbuf(didx:didx-1+dwords))

        iidx = iidx + iwords
        didx = didx + dwords
    end do
end if
end subroutine vamp_unmarshal_history

```

5.2.12 Boring Copying and Deleting of Objects

160 *⟨Implementation of vamp procedures 76d⟩*+≡

```

elemental subroutine vamp_copy_grid (lhs, rhs)
    type(vamp_grid), intent(inout) :: lhs
    type(vamp_grid), intent(in) :: rhs
    integer :: ndim
    ndim = size (rhs%div)
    lhs%mu = rhs%mu
    lhs%sum_integral = rhs%sum_integral
    lhs%sum_weights = rhs%sum_weights
    lhs%sum_chi2 = rhs%sum_chi2
    lhs%calls = rhs%calls
    lhs%num_calls = rhs%num_calls
    call copy_array_pointer (lhs%num_div, rhs%num_div)
    lhs%dv2g = rhs%dv2g
    lhs%jacobi = rhs%jacobi
    lhs%f_min = rhs%f_min
    lhs%f_max = rhs%f_max
    lhs%mu_gi = rhs%mu_gi
    lhs%sum_mu_gi = rhs%sum_mu_gi
    lhs%calls_per_cell = rhs%calls_per_cell
    lhs%stratified = rhs%stratified
    lhs%all_stratified = rhs%all_stratified
    lhs%quadrupole = rhs%quadrupole

```

```

if (associated (lhs%div)) then
  if (size (lhs%div) /= ndim) then
    call delete_division (lhs%div)
    deallocate (lhs%div)
    allocate (lhs%div(ndim))
  end if
else
  allocate (lhs%div(ndim))
end if
call copy_division (lhs%div, rhs%div)
if (associated (rhs%map)) then
  call copy_array_pointer (lhs%map, rhs%map)
else if (associated (lhs%map)) then
  deallocate (lhs%map)
end if
if (associated (rhs%mu_x)) then
  call copy_array_pointer (lhs%mu_x, rhs%mu_x)
  call copy_array_pointer (lhs%mu_xx, rhs%mu_xx)
  call copy_array_pointer (lhs%sum_mu_x, rhs%sum_mu_x)
  call copy_array_pointer (lhs%sum_mu_xx, rhs%sum_mu_xx)
else if (associated (lhs%mu_x)) then
  deallocate (lhs%mu_x, lhs%mu_xx, lhs%sum_mu_x, lhs%sum_mu_xx)
end if
end subroutine vamp_copy_grid

```

161a *⟨Implementation of vamp procedures 76d⟩*+≡

```

elemental subroutine vamp_delete_grid (g)
  type(vamp_grid), intent(inout) :: g
  if (associated (g%div)) then
    call delete_division (g%div)
    deallocate (g%div, g%num_div)
  end if
  if (associated (g%map)) then
    deallocate (g%map)
  end if
  if (associated (g%mu_x)) then
    deallocate (g%mu_x, g%mu_xx, g%sum_mu_x, g%sum_mu_xx)
  end if
end subroutine vamp_delete_grid

```

161b *⟨Implementation of vamp procedures 76d⟩*+≡

```

elemental subroutine vamp_copy_grids (lhs, rhs)
  type(vamp_grids), intent(inout) :: lhs
  type(vamp_grids), intent(in) :: rhs

```

```

integer :: nch
nch = size (rhs%grids)
lhs%sum_integral = rhs%sum_integral
lhs%sum_chi2 = rhs%sum_chi2
lhs%sum_weights = rhs%sum_weights
if (associated (lhs%grids)) then
  if (size (lhs%grids) /= nch) then
    deallocate (lhs%grids)
    allocate (lhs%grids(nch))
    call vamp_create_empty_grid (lhs%grids(nch))
  end if
else
  allocate (lhs%grids(nch))
  call vamp_create_empty_grid (lhs%grids(nch))
end if
call vamp_copy_grid (lhs%grids, rhs%grids)
call copy_array_pointer (lhs%weights, rhs%weights)
call copy_array_pointer (lhs%num_calls, rhs%num_calls)
end subroutine vamp_copy_grids

```

162a *⟨Implementation of vamp procedures 76d⟩*+≡

```

elemental subroutine vamp_delete_grids (g)
  type(vamp_grids), intent(inout) :: g
  if (associated (g%grids)) then
    call vamp_delete_grid (g%grids)
    deallocate (g%weights, g%grids, g%num_calls)
  end if
end subroutine vamp_delete_grids

```

162b *⟨Implementation of vamp procedures 76d⟩*+≡

```

elemental subroutine vamp_copy_history (lhs, rhs)
  type(vamp_history), intent(inout) :: lhs
  type(vamp_history), intent(in) :: rhs
  lhs%calls = rhs%calls
  lhs%stratified = rhs%stratified
  lhs%verbose = rhs%verbose
  lhs%integral = rhs%integral
  lhs%std_dev = rhs%std_dev
  lhs%avg_integral = rhs%avg_integral
  lhs%avg_std_dev = rhs%avg_std_dev
  lhs%avg_chi2 = rhs%avg_chi2
  lhs%f_min = rhs%f_min
  lhs%f_max = rhs%f_max
  if (rhs%verbose) then
    if (associated (lhs%div)) then

```

```

        if (size (lhs%div) /= size (rhs%div)) then
            deallocate (lhs%div)
            allocate (lhs%div(size(rhs%div)))
        end if
    else
        allocate (lhs%div(size(rhs%div)))
    end if
    call copy_history (lhs%div, rhs%div)
end if
end subroutine vamp_copy_history
163a <Implementation of vamp procedures 76d>+≡
    elemental subroutine vamp_delete_history (h)
        type(vamp_history), intent(inout) :: h
        if (associated (h%div)) then
            deallocate (h%div)
        end if
    end subroutine vamp_delete_history

```

5.3 Interface to MPI

The module `vamp` makes no specific assumptions about the hardware and software supporting parallel execution. In this section, we present a specific example of a parallel implementation of multi channel sampling using the message passing paradigm.

The modules `vamp_serial_mpi` and `vamp_parallel_mpi` are not intended to be used directly by application programs. For this purpose, the module `vampi` is provided. `vamp_serial_mpi` is identical to `vamp`, but some types, procedures and variables are renamed so that `vamp_parallel_mpi` can redefine them:

```

163b <vampi.f90 163b>≡
    ! vampi.f90 --
    <Copyleft notice 1>
    module vamp_serial_mpi
        use vamp, &
            <vamp0_* => vamp_* 165a>
            VAMPO_RCS_ID => VAMP_RCS_ID
        public
    end module vamp_serial_mpi

```

`vamp_parallel_mpi` contains the non trivial MPI code and will be discussed in detail below.

```

163c  <vampi.f90 163b>+≡
      module vamp_parallel_mpi
        use kinds
        use utils
        use tao_random_numbers
        use exceptions
        use mpi90
        use divisions
        use vamp_serial_mpi !NODEP!
        use iso_fortran_env
        implicit none
        private
        <Declaration of vampi procedures 164b>
        <Interfaces of vampi procedures 169b>
        <Parameters in vampi 165c>
        <Declaration of vampi types 169e>
        character(len=*), public, parameter :: VAMPI_RCS_ID = &
            "$Id: vampi.nw 314 2010-04-17 20:32:33Z ohl $"
        contains
        <Implementation of vampi procedures 165b>
      end module vamp_parallel_mpi

vampi is now a plug-in replacement for vamp and must not be used together
with vamp:

164a  <vampi.f90 163b>+≡
      module vampi
        use vamp_serial_mpi !NODEP!
        use vamp_parallel_mpi !NODEP!
        public
      end module vampi

```

5.3.1 Parallel Execution

Single Channel

```

164b  <Declaration of vampi procedures 164b>≡
      public :: vamp_create_grid
      public :: vamp_discard_integral
      public :: vamp_reshape_grid
      public :: vamp_sample_grid
      public :: vamp_delete_grid

```

```

165a  <vamp0_* => vamp_* 165a>≡
      vamp0_create_grid => vamp_create_grid, &
      vamp0_discard_integral => vamp_discard_integral, &
      vamp0_reshape_grid => vamp_reshape_grid, &
      vamp0_sample_grid => vamp_sample_grid, &
      vamp0_delete_grid => vamp_delete_grid, &

165b  <Implementation of vampi procedures 165b>≡
      subroutine vamp_create_grid &
        (g, domain, num_calls, num_div, &
         stratified, quadrupole, covariance, map, exc)
        type(vamp_grid), intent(inout) :: g
        real(kind=default), dimension(:,:), intent(in) :: domain
        integer, intent(in) :: num_calls
        integer, dimension(:), intent(in), optional :: num_div
        logical, intent(in), optional :: stratified, quadrupole, covariance
        real(kind=default), dimension(:,:), intent(in), optional :: map
        type(exception), intent(inout), optional :: exc
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
          call vamp0_create_grid &
            (g, domain, num_calls, num_div, &
             stratified, quadrupole, covariance, map, exc)
        else
          call vamp_create_empty_grid (g)
        end if
      end subroutine vamp_create_grid

165c  <Parameters in vampi 165c>≡
      integer, public, parameter :: VAMP_ROOT = 0

165d  <Implementation of vampi procedures 165b>+≡
      subroutine vamp_discard_integral &
        (g, num_calls, num_div, stratified, quadrupole, covariance, exc)
        type(vamp_grid), intent(inout) :: g
        integer, intent(in), optional :: num_calls
        integer, dimension(:), intent(in), optional :: num_div
        logical, intent(in), optional :: stratified, quadrupole, covariance
        type(exception), intent(inout), optional :: exc
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
          call vamp0_discard_integral &
            (g, num_calls, num_div, stratified, quadrupole, covariance, exc)

```

```

    end if
end subroutine vamp_discard_integral

```

166a *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_reshape_grid &
    (g, num_calls, num_div, stratified, quadrupole, covariance, exc)
    type(vamp_grid), intent(inout) :: g
    integer, intent(in), optional :: num_calls
    integer, dimension(:), intent(in), optional :: num_div
    logical, intent(in), optional :: stratified, quadrupole, covariance
    type(exception), intent(inout), optional :: exc
    integer :: proc_id
    call mpi90_rank (proc_id)
    if (proc_id == VAMP_ROOT) then
        call vamp0_reshape_grid &
            (g, num_calls, num_div, stratified, quadrupole, covariance, exc)
    end if
end subroutine vamp_reshape_grid

```

NB: grids has to have intent(inout) because we will call vamp_broadcast_grid on it.

166b *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_sample_grid &
    (rng, g, func, iterations, integral, std_dev, avg_chi2, accuracy, &
    channel, weights, grids, exc, history)
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grid), intent(inout) :: g
    integer, intent(in) :: iterations
    real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
    real(kind=default), intent(in), optional :: accuracy
    integer, intent(in), optional :: channel
    real(kind=default), dimension(:), intent(in), optional :: weights
    type(vamp_grid), dimension(:), intent(inout), optional :: grids
    type(exception), intent(inout), optional :: exc
    type(vamp_history), dimension(:), intent(inout), optional :: history
<Interface declaration for func 22>
    character(len=*), parameter :: FN = "vamp_sample_grid"
    real(kind=default) :: local_integral, local_std_dev, local_avg_chi2
    type(vamp_grid), dimension(:), allocatable :: gs, gx
    integer, dimension(:,:), pointer :: d
    integer :: iteration, i
    integer :: num_proc, proc_id, num_workers
    nullify (d)
    call mpi90_size (num_proc)

```

```

call mpi90_rank (proc_id)
iterate: do iteration = 1, iterations
  if (proc_id == VAMP_ROOT) then
    call vamp_distribute_work (num_proc, vamp_rigid_divisions (g), d)
    num_workers = max (1, product (d(2,:)))
  end if
  call mpi90_broadcast (num_workers, VAMP_ROOT)
  if ((present (grids)) .and. (num_workers > 1)) then
    call vamp_broadcast_grid (grids, VAMP_ROOT)
  end if
  if (proc_id == VAMP_ROOT) then
    allocate (gs(num_workers), gx(vamp_fork_grid_joints (d)))
    call vamp_create_empty_grid (gs)
    call vamp_fork_grid (g, gs, gx, d, exc)
    do i = 2, num_workers
      call vamp_send_grid (gs(i), i-1, 0)
    end do
  else if (proc_id < num_workers) then
    call vamp_receive_grid (g, VAMP_ROOT, 0)
  end if
  if (proc_id == VAMP_ROOT) then
    if (num_workers > 1) then
      call vamp_sample_grid0 &
        (rng, gs(1), func, channel, weights, grids, exc)
    else
      call vamp_sample_grid0 &
        (rng, g, func, channel, weights, grids, exc)
    end if
  else if (proc_id < num_workers) then
    call vamp_sample_grid0 &
      (rng, g, func, channel, weights, grids, exc)
  end if
  if (proc_id == VAMP_ROOT) then
    do i = 2, num_workers
      call vamp_receive_grid (gs(i), i-1, 0)
    end do
    call vamp_join_grid (g, gs, gx, d, exc)
    call vamp0_delete_grid (gs)
    deallocate (gs, gx)
    call vamp_refine_grid (g)
    call vamp_average_iterations &
      (g, iteration, local_integral, local_std_dev, local_avg_chi2)
    if (present (history)) then

```



```

        if (iteration <= size (history)) then
            call vamp_get_history &
                (history(iteration), g, &
                 local_integral, local_std_dev, local_avg_chi2)
        else
            call raise_exception (exc, EXC_WARN, FN, "history too short")
        end if
        call vamp_terminate_history (history(iteration+1:))
    end if
    if (present (accuracy)) then
        if (local_std_dev <= accuracy * local_integral) then
            call raise_exception (exc, EXC_INFO, FN, &
                "requested accuracy reached")
            exit iterate
        end if
    end if
else if (proc_id < num_workers) then
    call vamp_send_grid (g, VAMP_ROOT, 0)
end if
end do iterate
if (proc_id == VAMP_ROOT) then
    deallocate (d)
    if (present (integral)) then
        integral = local_integral
    end if
    if (present (std_dev)) then
        std_dev = local_std_dev
    end if
    if (present (avg_chi2)) then
        avg_chi2 = local_avg_chi2
    end if
end if
end subroutine vamp_sample_grid

```

168a *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_delete_grid (g)
    type(vamp_grid), intent(inout) :: g
    integer :: proc_id
    call mpi90_rank (proc_id)
    if (proc_id == VAMP_ROOT) then
        call vamp0_reshape_grid (g)
    end if
end subroutine vamp_delete_grid

```

168b *<Declaration of vampi procedures 164b>+≡*

```

    public :: vamp_print_history
    private :: vamp_print_one_history, vamp_print_histories

169a <vamp0_* => vamp_* 165a>+≡
    vamp0_print_history => vamp_print_history, &

169b <Interfaces of vampi procedures 169b>≡
    interface vamp_print_history
        module procedure vamp_print_one_history, vamp_print_histories
    end interface

169c <Implementation of vampi procedures 165b>+≡
    subroutine vamp_print_one_history (h, tag)
        type(vamp_history), dimension(:), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
            call vamp0_print_history (h, tag)
        end if
    end subroutine vamp_print_one_history

169d <Implementation of vampi procedures 165b>+≡
    subroutine vamp_print_histories (h, tag)
        type(vamp_history), dimension(:, :), intent(in) :: h
        character(len=*), intent(in), optional :: tag
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
            call vamp0_print_history (h, tag)
        end if
    end subroutine vamp_print_histories

```

Multi Channel

```

169e <Declaration of vampi types 169e>≡
    type, public :: vamp_grids
    !!! private
        type(vamp0_grids) :: g0
        logical, dimension(:), pointer :: active
        integer, dimension(:), pointer :: proc
        real(kind=default), dimension(:), pointer :: integrals, std_devs
    end type vamp_grids

169f <vamp0_* => vamp_* 165a>+≡
    vamp0_grids => vamp_grids, &

```

Partially duplicate the API of vamp:

170a *<Declaration of vampi procedures 164b>+≡*

```
public :: vamp_create_grids
public :: vamp_discard_integrals
public :: vamp_update_weights
public :: vamp_refine_weights
public :: vamp_delete_grids
public :: vamp_sample_grids
```

170b *<vamp0_* => vamp_* 165a>+≡*

```
vamp0_create_grids => vamp_create_grids, &
vamp0_discard_integrals => vamp_discard_integrals, &
vamp0_update_weights => vamp_update_weights, &
vamp0_refine_weights => vamp_refine_weights, &
vamp0_delete_grids => vamp_delete_grids, &
vamp0_sample_grids => vamp_sample_grids, &
```

Call vamp_create_grids just like the serial version. It will create the actual grids on the root processor and create stubs on the other processors

170c *<Implementation of vampi procedures 165b>+≡*

```
subroutine vamp_create_grids (g, domain, num_calls, weights, maps, &
                             num_div, stratified, quadrupole, exc)
    type(vamp_grids), intent(inout) :: g
    real(kind=default), dimension(:,,:), intent(in) :: domain
    integer, intent(in) :: num_calls
    real(kind=default), dimension(:), intent(in) :: weights
    real(kind=default), dimension(:, :, :), intent(in), optional :: maps
    integer, dimension(:), intent(in), optional :: num_div
    logical, intent(in), optional :: stratified, quadrupole
    type(exception), intent(inout), optional :: exc
    integer :: proc_id, nch
    call mpi90_rank (proc_id)
    nch = size (weights)
    allocate (g%active(nch), g%proc(nch), g%integrals(nch), g%std_devs(nch))
    if (proc_id == VAMP_ROOT) then
        call vamp0_create_grids (g%g0, domain, num_calls, weights, maps, &
                                num_div, stratified, quadrupole, exc)
    else
        allocate (g%g0%grids(nch), g%g0%weights(nch), g%g0%num_calls(nch))
        call vamp_create_empty_grid (g%g0%grids)
    end if
end subroutine vamp_create_grids
```

170d *<Implementation of vampi procedures 165b>+≡*

```
subroutine vamp_discard_integrals &
```

```

        (g, num_calls, num_div, stratified, quadrupole, exc)
type(vamp_grids), intent(inout) :: g
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
type(exception), intent(inout), optional :: exc
integer :: proc_id
call mpi90_rank (proc_id)
if (proc_id == VAMP_ROOT) then
    call vamp0_discard_integrals &
        (g%g0, num_calls, num_div, stratified, quadrupole, exc)
end if
end subroutine vamp_discard_integrals

```

171a *⟨Implementation of vampi procedures 165b⟩*+≡

```

subroutine vamp_update_weights &
    (g, weights, num_calls, num_div, stratified, quadrupole, exc)
type(vamp_grids), intent(inout) :: g
real(kind=default), dimension(:), intent(in) :: weights
integer, intent(in), optional :: num_calls
integer, dimension(:), intent(in), optional :: num_div
logical, intent(in), optional :: stratified, quadrupole
type(exception), intent(inout), optional :: exc
integer :: proc_id
call mpi90_rank (proc_id)
if (proc_id == VAMP_ROOT) then
    call vamp0_update_weights &
        (g%g0, weights, num_calls, num_div, stratified, quadrupole, exc)
end if
end subroutine vamp_update_weights

```

171b *⟨Implementation of vampi procedures 165b⟩*+≡

```

subroutine vamp_refine_weights (g, power)
type(vamp_grids), intent(inout) :: g
real(kind=default), intent(in), optional :: power
integer :: proc_id
call mpi90_rank (proc_id)
if (proc_id == VAMP_ROOT) then
    call vamp0_refine_weights (g%g0, power)
end if
end subroutine vamp_refine_weights

```

171c *⟨Implementation of vampi procedures 165b⟩*+≡

```

subroutine vamp_delete_grids (g)
type(vamp_grids), intent(inout) :: g

```

```

character(len=*), parameter :: FN = "vamp_delete_grids"
deallocate (g%active, g%proc, g%integrals, g%std_devs)
call vamp0_delete_grids (g%g0)
end subroutine vamp_delete_grids

```

Call `vamp_sample_grids` just like `vamp0_sample_grids`.

172 *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_sample_grids &
  (rng, g, func, iterations, integral, std_dev, avg_chi2, &
   accuracy, history, histories, exc)
type(tao_random_state), intent(inout) :: rng
type(vamp_grids), intent(inout) :: g
integer, intent(in) :: iterations
real(kind=default), intent(out), optional :: integral, std_dev, avg_chi2
real(kind=default), intent(in), optional :: accuracy
type(vamp_history), dimension(:), intent(inout), optional :: history
type(vamp_history), dimension(:, :), intent(inout), optional :: histories
type(exception), intent(inout), optional :: exc
<Interface declaration for func 22>
character(len=*), parameter :: FN = "vamp_sample_grids"
integer :: num_proc, proc_id, nch, ch, iteration
real(kind=default), dimension(size(g%g0%weights)) :: weights
real(kind=default) :: local_integral, local_std_dev, local_avg_chi2
real(kind=default) :: current_accuracy, waste
logical :: distribute_complete_grids
call mpi90_size (num_proc)
call mpi90_rank (proc_id)
nch = size (g%g0%weights)
if (proc_id == VAMP_ROOT) then
  g%active = (g%g0%num_calls >= 2)
  where (g%active)
    weights = g%g0%num_calls
  elsewhere
    weights = 0.0
  endwhere
  weights = weights / sum (weights)
  call schedule (weights, num_proc, g%proc, waste)
  distribute_complete_grids = (waste <= VAMP_MAX_WASTE)
end if
call mpi90_broadcast (weights, VAMP_ROOT)
call mpi90_broadcast (g%active, VAMP_ROOT)
call mpi90_broadcast (distribute_complete_grids, VAMP_ROOT)
if (distribute_complete_grids) then
  call mpi90_broadcast (g%proc, VAMP_ROOT)

```

```

end if
iterate: do iteration = 1, iterations
  if (distribute_complete_grids) then
    call vamp_broadcast_grid (g%g0%grids, VAMP_ROOT)
    <Distribute complete grids among processes 173b>
  else
    <Distribute each grid among processes 177b>
  end if
  <Exit iterate if accuracy has been reached (MPI) 176a>
end do iterate
<Copy results of vamp_sample_grids to dummy variables 175f>
end subroutine vamp_sample_grids

```

Setting VAMP_MAX_WASTE to 1 disables the splitting of grids, which doesn't work yet.

```

173a <Parameters in vampi 165c>+≡
  real(kind=default), private, parameter :: VAMP_MAX_WASTE = 1.0
  ! real(kind=default), private, parameter :: VAMP_MAX_WASTE = 0.3

```

```

173b <Distribute complete grids among processes 173b>≡
  do ch = 1, nch
    if (g%active(ch)) then
      if (proc_id == g%proc(ch)) then
        call vamp0_discard_integral (g%g0%grids(ch))
        <Sample g%g0%grids(ch) 173d>
      end if
    else
      call vamp_nullify_variance (g%g0%grids(ch))
      call vamp_nullify_covariance (g%g0%grids(ch))
    end if
  end do

```

Refine the grids after *all* grids have been sampled:

```

173c <Distribute complete grids among processes 173b>+≡
  do ch = 1, nch
    if (g%active(ch) .and. (proc_id == g%proc(ch))) then
      call vamp_refine_grid (g%g0%grids(ch))
      if (proc_id /= VAMP_ROOT) then
        <Ship the result for channel #ch back to the root 175a>
      end if
    end if
  end do

```

therefore we use vamp_sample_grid0 instead of vamp0_sample_grid:

```

173d <Sample g%g0%grids(ch) 173d>≡

```

```

call vamp_sample_grid0 &
  (rng, g%g0%grids(ch), func, ch, weights, g%g0%grids, exc)
call vamp_average_iterations &
  (g%g0%grids(ch), iteration, g%integrals(ch), g%std_devs(ch), local_avg_chi2)
if (present (histories)) then
  if (iteration <= ubound (histories, dim=1)) then
    call vamp_get_history &
      (histories(iteration,ch), g%g0%grids(ch), &
       g%integrals(ch), g%std_devs(ch), local_avg_chi2)
  else
    call raise_exception (exc, EXC_WARN, FN, "history too short")
  end if
  call vamp_terminate_history (histories(iteration+1:,ch))
end if

```

174a *⟨Distribute complete grids among processes 173b⟩+≡*

```

if (proc_id == VAMP_ROOT) then
  do ch = 1, nch
    if (g%active(ch) .and. (g%proc(ch) /= proc_id)) then
      ⟨Receive the result for channel #ch at the root 175b⟩
    end if
  end do
  call vamp_reduce_channels (g%g0, g%integrals, g%std_devs, g%active)
  call vamp_average_iterations &
    (g%g0, iteration, local_integral, local_std_dev, local_avg_chi2)
  if (present (history)) then
    if (iteration <= size (history)) then
      call vamp_get_history &
        (history(iteration), g%g0, local_integral, local_std_dev, &
         local_avg_chi2)
    else
      call raise_exception (exc, EXC_WARN, FN, "history too short")
    end if
    call vamp_terminate_history (history(iteration+1:))
  end if
end if

```

This would be cheaper than `vamp_broadcast_grid`, but we need the latter to support the adaptive multi channel sampling:

174b *⟨Ship g%g0%grids from the root to the assigned processor 174b⟩≡*

```

do ch = 1, nch
  if (g%active(ch) .and. (g%proc(ch) /= VAMP_ROOT)) then
    if (proc_id == VAMP_ROOT) then
      call vamp_send_grid &

```

```

        (g%g0%grids(ch), g%proc(ch), object (ch, TAG_GRID))
    else if (proc_id == g%proc(ch)) then
        call vamp_receive_grid &
            (g%g0%grids(ch), VAMP_ROOT, object (ch, TAG_GRID))
    end if
end if
end do

175a <Ship the result for channel #ch back to the root 175a>≡
    call mpi90_send (g%integrals(ch), VAMP_ROOT, object (ch, TAG_INTEGRAL))
    call mpi90_send (g%std_devs(ch), VAMP_ROOT, object (ch, TAG_STD_DEV))
    call vamp_send_grid (g%g0%grids(ch), VAMP_ROOT, object (ch, TAG_GRID))
    if (present (histories)) then
        call vamp_send_history &
            (histories(iteration,ch), VAMP_ROOT, object (ch, TAG_HISTORY))
    end if

175b <Receive the result for channel #ch at the root 175b>≡
    call mpi90_receive (g%integrals(ch), g%proc(ch), object (ch, TAG_INTEGRAL))
    call mpi90_receive (g%std_devs(ch), g%proc(ch), object (ch, TAG_STD_DEV))
    call vamp_receive_grid (g%g0%grids(ch), g%proc(ch), object (ch, TAG_GRID))
    if (present (histories)) then
        call vamp_receive_history &
            (histories(iteration,ch), g%proc(ch), object (ch, TAG_HISTORY))
    end if

175c <Declaration of vampi procedures 164b>+≡
    private :: object

175d <Implementation of vampi procedures 165b>+≡
    pure function object (ch, obj) result (tag)
        integer, intent(in) :: ch, obj
        integer :: tag
        tag = 100 * ch + obj
    end function object

175e <Parameters in vampi 165c>+≡
    integer, public, parameter :: &
        TAG_INTEGRAL = 1, &
        TAG_STD_DEV = 2, &
        TAG_GRID = 3, &
        TAG_HISTORY = 6, &
        TAG_NEXT_FREE = 9

175f <Copy results of vamp_sample_grids to dummy variables 175f>≡
    if (present (integral)) then
        call mpi90_broadcast (local_integral, VAMP_ROOT)

```



```

        integral = local_integral
    end if
    if (present (std_dev)) then
        call mpi90_broadcast (local_std_dev, VAMP_ROOT)
        std_dev = local_std_dev
    end if
    if (present (avg_chi2)) then
        call mpi90_broadcast (local_avg_chi2, VAMP_ROOT)
        avg_chi2 = local_avg_chi2
    end if

```

176a *<Exit iterate if accuracy has been reached (MPI) 176a>≡*

```

    if (present (accuracy)) then
        if (proc_id == VAMP_ROOT) then
            current_accuracy = local_std_dev / local_integral
        end if
        call mpi90_broadcast (current_accuracy, VAMP_ROOT)
        if (current_accuracy <= accuracy) then
            call raise_exception (exc, EXC_INFO, FN, &
                "requested accuracy reached")
            exit iterate
        end if
    end if

```

A very simple minded scheduler: maximizes processor utilization and, does not pay attention to communication costs.

176b *<Declaration of vampi procedures 164b>+≡*

```

    private :: schedule

```

We disfavor the root process a little bit (by starting up with a fake filling ratio of 10%) so that it is likely to be ready to answer all communication requests.

176c *<Implementation of vampi procedures 165b>+≡*

```

    pure subroutine schedule (jobs, num_procs, assign, waste)
        real(kind=default), dimension(:), intent(in) :: jobs
        integer, intent(in) :: num_procs
        integer, dimension(:), intent(out) :: assign
        real(kind=default), intent(out), optional :: waste
        integer, dimension(size(jobs)) :: idx
        real(kind=default), dimension(size(jobs)) :: sjobs
        real(kind=default), dimension(num_procs) :: fill
        integer :: job, proc
        sjobs = jobs / sum (jobs) * num_procs
        idx = (/ (job, job = 1, size(jobs)) /)
        call sort (sjobs, idx, reverse = .true.)

```

```

fill = 0.0
fill(VAMP_ROOT+1) = 0.1
do job = 1, size (sjobs)
  proc = sum (minloc (fill))
  fill(proc) = fill(proc) + sjobs(job)
  assign(idx(job)) = proc - 1
end do
<Estimate waste of processor time 177a>
end subroutine schedule

```

Assuming equivalent processors and uniform computation costs, the waste is given by the fraction of the time that it spent by the other processors waiting for the processor with the biggest assignment:

```

177a <Estimate waste of processor time 177a>≡
  if (present (waste)) then
    waste = 1.0 - sum (fill) / (num_procs * maxval (fill))
  end if

```

Accordingly, if the waste caused by distributing only complete grids, we switch to splitting the grids, just like in single channel sampling. This is *not* the default, because the communication costs are measurably higher for many grids and many processors.

 This version is broken!

```

177b <Distribute each grid among processes 177b>≡
  do ch = 1, size (g%g0%grids)
    if (g%active(ch)) then
      call vamp_discard_integral (g%g0%grids(ch))
      if (present (histories)) then
        call vamp_sample_grid &
          (rng, g%g0%grids(ch), func, 1, g%integrals(ch), g%std_devs(ch), &
           channel = ch, weights = weights, grids = g%g0%grids, &
           history = histories(iteration:iteration,ch))
      else
        call vamp_sample_grid &
          (rng, g%g0%grids(ch), func, 1, g%integrals(ch), g%std_devs(ch), &
           channel = ch, weights = weights, grids = g%g0%grids)
      end if
    else
      if (proc_id == VAMP_ROOT) then
        call vamp_nullify_variance (g%g0%grids(ch))
        call vamp_nullify_covariance (g%g0%grids(ch))
      end if
    end if
  end do

```

```

end do
if (proc_id == VAMP_ROOT) then
  call vamp_reduce_channels (g%g0, g%integrals, g%std_devs, g%active)
  call vamp_average_iterations &
    (g%g0, iteration, local_integral, local_std_dev, local_avg_chi2)
  if (present (history)) then
    if (iteration <= size (history)) then
      call vamp_get_history &
        (history(iteration), g%g0, local_integral, local_std_dev, &
          local_avg_chi2)
    else
      call raise_exception (exc, EXC_WARN, FN, "history too short")
    end if
    call vamp_terminate_history (history(iteration+1:))
  end if
end if

```

5.3.2 Event Generation

This is currently only a syntactical translation ...

- 178a \langle Declaration of vampi procedures 164b $\rangle + \equiv$
- ```

public :: vamp_warmup_grid
public :: vamp_warmup_grids
public :: vamp_next_event
private :: vamp_next_event_single, vamp_next_event_multi

```
- 178b  $\langle$ vamp0\_\* => vamp\_\* 165a $\rangle + \equiv$
- ```

vamp0_warmup_grid => vamp_warmup_grid, &
vamp0_warmup_grids => vamp_warmup_grids, &
vamp0_next_event => vamp_next_event, &

```
- 178c \langle Interfaces of vampi procedures 169b $\rangle + \equiv$
- ```

interface vamp_next_event
 module procedure vamp_next_event_single, vamp_next_event_multi
end interface

```
- 178d  $\langle$ Implementation of vampi procedures 165b $\rangle + \equiv$
- ```

subroutine vamp_next_event_single &
  (x, rng, g, func, weight, channel, weights, grids, exc)
  real(kind=default), dimension(:), intent(out) :: x
  type(tao_random_state), intent(inout) :: rng
  type(vamp_grid), intent(inout) :: g
  real(kind=default), intent(out), optional :: weight
  integer, intent(in), optional :: channel

```

```

real(kind=default), dimension(:), intent(in), optional :: weights
type(vamp_grid), dimension(:), intent(in), optional :: grids
type(exception), intent(inout), optional :: exc
<Interface declaration for func 22>
integer :: proc_id
call mpi90_rank (proc_id)
if (proc_id == VAMP_ROOT) then
    call vamp0_next_event &
        (x, rng, g, func, weight, channel, weights, grids, exc)
end if
end subroutine vamp_next_event_single

```

179a <Implementation of vampi procedures 165b>+≡

```

subroutine vamp_next_event_multi (x, rng, g, func, phi, weight, exc)
    real(kind=default), dimension(:), intent(out) :: x
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grids), intent(inout) :: g
    real(kind=default), intent(out), optional :: weight
    type(exception), intent(inout), optional :: exc
    <Interface declaration for func 22>
    <Interface declaration for phi 31a>
    integer :: proc_id
    call mpi90_rank (proc_id)
    if (proc_id == VAMP_ROOT) then
        call vamp0_next_event (x, rng, g%g0, func, phi, weight, exc)
    end if
end subroutine vamp_next_event_multi

```

179b <Implementation of vampi procedures 165b>+≡

```

subroutine vamp_warmup_grid (rng, g, func, iterations, exc, history)
    type(tao_random_state), intent(inout) :: rng
    type(vamp_grid), intent(inout) :: g
    integer, intent(in) :: iterations
    type(exception), intent(inout), optional :: exc
    type(vamp_history), dimension(:), intent(inout), optional :: history
    <Interface declaration for func 22>
    call vamp_sample_grid &
        (rng, g, func, iterations - 1, exc = exc, history = history)
    call vamp_sample_grid0 (rng, g, func, exc = exc)
end subroutine vamp_warmup_grid

```

179c <Implementation of vampi procedures 165b>+≡

```

subroutine vamp_warmup_grids &
    (rng, g, func, iterations, history, histories, exc)
    type(tao_random_state), intent(inout) :: rng

```

```

type(vamp_grids), intent(inout) :: g
integer, intent(in) :: iterations
type(vamp_history), dimension(:), intent(inout), optional :: history
type(vamp_history), dimension(:, :), intent(inout), optional :: histories
type(exception), intent(inout), optional :: exc
<Interface declaration for func 22>
integer :: ch
call vamp0_sample_grids (rng, g%g0, func, iterations - 1, exc = exc, &
                        history = history, histories = histories)
do ch = 1, size (g%g0%grids)
  ! if (g%g0%grids(ch)%num_calls >= 2) then
    call vamp_sample_grid0 (rng, g%g0%grids(ch), func, exc = exc)
  ! end if
end do
end subroutine vamp_warmup_grids

```

5.3.3 I/O

- 180a *<Declaration of vampi procedures 164b>+≡*

```

public :: vamp_write_grid, vamp_read_grid
private :: write_grid_unit, write_grid_name
private :: read_grid_unit, read_grid_name

```
- 180b *<vamp0_* => vamp_* 165a>+≡*

```

vamp0_write_grid => vamp_write_grid, &
vamp0_read_grid => vamp_read_grid, &

```
- 180c *<Interfaces of vampi procedures 169b>+≡*

```

interface vamp_write_grid
  module procedure write_grid_unit, write_grid_name
end interface
interface vamp_read_grid
  module procedure read_grid_unit, read_grid_name
end interface

```
- 180d *<Implementation of vampi procedures 165b>+≡*

```

subroutine write_grid_unit (g, unit)
  type(vamp_grid), intent(in) :: g
  integer, intent(in) :: unit
  integer :: proc_id
  call mpi90_rank (proc_id)
  if (proc_id == VAMP_ROOT) then
    call vamp0_write_grid (g, unit)
  end if
end subroutine write_grid_unit

```

```

181a  <Implementation of vampi procedures 165b>+≡
      subroutine read_grid_unit (g, unit)
        type(vamp_grid), intent(inout) :: g
        integer, intent(in) :: unit
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
          call vamp0_read_grid (g, unit)
        end if
      end subroutine read_grid_unit

181b  <Implementation of vampi procedures 165b>+≡
      subroutine write_grid_name (g, name)
        type(vamp_grid), intent(inout) :: g
        character(len=*), intent(in) :: name
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
          call vamp0_write_grid (g, name)
        end if
      end subroutine write_grid_name

181c  <Implementation of vampi procedures 165b>+≡
      subroutine read_grid_name (g, name)
        type(vamp_grid), intent(inout) :: g
        character(len=*), intent(in) :: name
        integer :: proc_id
        call mpi90_rank (proc_id)
        if (proc_id == VAMP_ROOT) then
          call vamp0_read_grid (g, name)
        end if
      end subroutine read_grid_name

181d  <Declaration of vampi procedures 164b>+≡
      public :: vamp_write_grids, vamp_read_grids
      private :: write_grids_unit, write_grids_name
      private :: read_grids_unit, read_grids_name

181e  <vamp0_* => vamp_* 165a>+≡
      vamp0_write_grids => vamp_write_grids, &
      vamp0_read_grids => vamp_read_grids, &

181f  <Interfaces of vampi procedures 169b>+≡
      interface vamp_write_grids
        module procedure write_grids_unit, write_grids_name
      end interface

```

```

interface vamp_read_grids
  module procedure read_grids_unit, read_grids_name
end interface

```

182a *<Implementation of vampi procedures 165b>+≡*

```

subroutine write_grids_unit (g, unit)
  type(vamp_grids), intent(in) :: g
  integer, intent(in) :: unit
  integer :: proc_id
  call mpi90_rank (proc_id)
  if (proc_id == VAMP_ROOT) then
    call vamp0_write_grids (g%g0, unit)
  end if
end subroutine write_grids_unit

```

182b *<Implementation of vampi procedures 165b>+≡*

```

subroutine read_grids_unit (g, unit)
  type(vamp_grids), intent(inout) :: g
  integer, intent(in) :: unit
  integer :: proc_id
  call mpi90_rank (proc_id)
  if (proc_id == VAMP_ROOT) then
    call vamp0_read_grids (g%g0, unit)
  end if
end subroutine read_grids_unit

```

182c *<Implementation of vampi procedures 165b>+≡*

```

subroutine write_grids_name (g, name)
  type(vamp_grids), intent(inout) :: g
  character(len=*), intent(in) :: name
  integer :: proc_id
  call mpi90_rank (proc_id)
  if (proc_id == VAMP_ROOT) then
    call vamp0_write_grids (g%g0, name)
  end if
end subroutine write_grids_name

```

182d *<Implementation of vampi procedures 165b>+≡*

```

subroutine read_grids_name (g, name)
  type(vamp_grids), intent(inout) :: g
  character(len=*), intent(in) :: name
  integer :: proc_id
  call mpi90_rank (proc_id)
  if (proc_id == VAMP_ROOT) then
    call vamp0_read_grids (g%g0, name)
  end if

```

```
end subroutine read_grids_name
```

5.3.4 Communicating Grids

183a *<Declaration of vampi procedures 164b>+≡*

```
public :: vamp_send_grid
public :: vamp_receive_grid
public :: vamp_broadcast_grid
public :: vamp_broadcast_grids
```



The next two are still kludged. Nicer implementations with one message less per call below, but MPICH does funny things during `mpi_get_count`, which is called by `mpi90_receive_pointer`.

Caveat: this `vamp_send_grid` uses *three* tags: `tag`, `tag+1` and `tag+2`:

183b *<Implementation of vampi procedures 165b>+≡*

```
subroutine vamp_send_grid (g, target, tag, domain, error)
  type(vamp_grid), intent(in) :: g
  integer, intent(in) :: target, tag
  integer, intent(in), optional :: domain
  integer, intent(out), optional :: error
  integer, dimension(2) :: words
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
  call vamp_marshall_grid_size (g, words(1), words(2))
  allocate (ibuf(words(1)), dbuf(words(2)))
  call vamp_marshall_grid (g, ibuf, dbuf)
  call mpi90_send (words, target, tag, domain, error)
  call mpi90_send (ibuf, target, tag+1, domain, error)
  call mpi90_send (dbuf, target, tag+2, domain, error)
  deallocate (ibuf, dbuf)
end subroutine vamp_send_grid
```

183c *<Implementation of vampi procedures 165b>+≡*

```
subroutine vamp_receive_grid (g, source, tag, domain, status, error)
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: source, tag
  integer, intent(in), optional :: domain
  type(mpi90_status), intent(out), optional :: status
  integer, intent(out), optional :: error
  integer, dimension(2) :: words
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
```



```

    call mpi90_receive (words, source, tag, domain, status, error)
    allocate (ibuf(words(1)), dbuf(words(2)))
    call mpi90_receive (ibuf, source, tag+1, domain, status, error)
    call mpi90_receive (dbuf, source, tag+2, domain, status, error)
    call vamp_unmarshal_grid (g, ibuf, dbuf)
    deallocate (ibuf, dbuf)
end subroutine vamp_receive_grid

```

Caveat: the real `vamp_send_grid` uses *two* tags: `tag` and `tag+1`:

184a *⟨Implementation of vampi procedures (doesn't work with MPICH yet) 184a⟩≡*

```

subroutine vamp_send_grid (g, target, tag, domain, error)
  type(vamp_grid), intent(in) :: g
  integer, intent(in) :: target, tag
  integer, intent(in), optional :: domain
  integer, intent(out), optional :: error
  integer :: iwords, dwords
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
  call vamp_marshall_grid_size (g, iwords, dwords)
  allocate (ibuf(iwords), dbuf(dwords))
  call vamp_marshall_grid (g, ibuf, dbuf)
  call mpi90_send (ibuf, target, tag, domain, error)
  call mpi90_send (dbuf, target, tag+1, domain, error)
  deallocate (ibuf, dbuf)
end subroutine vamp_send_grid

```



There's something wrong with MPICH: if I call `mpi90_receive_pointer` in the opposite order, the low level call to `mpi_get_count` bombs for no apparent reason!



There are also funky things going on with `tag`: `mpi90_receive_pointer` should leave it alone, but ...

184b *⟨Implementation of vampi procedures (doesn't work with MPICH yet) 184a⟩+≡*

```

subroutine vamp_receive_grid (g, source, tag, domain, status, error)
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: source, tag
  integer, intent(in), optional :: domain
  type(mpi90_status), intent(out), optional :: status
  integer, intent(out), optional :: error
  integer, dimension(:), pointer :: ibuf
  real(kind=default), dimension(:), pointer :: dbuf
  nullify (ibuf, dbuf)
  call mpi90_receive_pointer (dbuf, source, tag+1, domain, status, error)

```

```

    call mpi90_receive_pointer (ibuf, source, tag, domain, status, error)
    call vamp_unmarshal_grid (g, ibuf, dbuf)
    deallocate (ibuf, dbuf)
end subroutine vamp_receive_grid

```

This if not a good idea, with respect to communication costs. For SMP machines, it appears to be negligible however.

185a *<Interfaces of vampi procedures 169b>+≡*

```

interface vamp_broadcast_grid
  module procedure &
    vamp_broadcast_one_grid, vamp_broadcast_many_grids
end interface

```

185b *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_broadcast_one_grid (g, root, domain, error)
  type(vamp_grid), intent(inout) :: g
  integer, intent(in) :: root
  integer, intent(in), optional :: domain
  integer, intent(out), optional :: error
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
  integer :: iwords, dwords, me
  call mpi90_rank (me)
  if (me == root) then
    call vamp_marshal_grid_size (g, iwords, dwords)
  end if
  call mpi90_broadcast (iwords, root, domain, error)
  call mpi90_broadcast (dwords, root, domain, error)
  allocate (ibuf(iwords), dbuf(dwords))
  if (me == root) then
    call vamp_marshal_grid (g, ibuf, dbuf)
  end if
  call mpi90_broadcast (ibuf, root, domain, error)
  call mpi90_broadcast (dbuf, root, domain, error)
  if (me /= root) then
    call vamp_unmarshal_grid (g, ibuf, dbuf)
  end if
  deallocate (ibuf, dbuf)
end subroutine vamp_broadcast_one_grid

```

185c *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_broadcast_many_grids (g, root, domain, error)
  type(vamp_grid), dimension(:), intent(inout) :: g
  integer, intent(in) :: root
  integer, intent(in), optional :: domain

```

```

integer, intent(out), optional :: error
integer :: i
do i = 1, size(g)
    call vamp_broadcast_one_grid (g(i), root, domain, error)
end do
end subroutine vamp_broadcast_many_grids

```

186a *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_broadcast_grids (g, root, domain, error)
    type(vamp0_grids), intent(inout) :: g
    integer, intent(in) :: root
    integer, intent(in), optional :: domain
    integer, intent(out), optional :: error
    integer :: nch, me
    call mpi90_broadcast (g%sum_chi2, root, domain, error)
    call mpi90_broadcast (g%sum_integral, root, domain, error)
    call mpi90_broadcast (g%sum_weights, root, domain, error)
    call mpi90_rank (me)
    if (me == root) then
        nch = size (g%grids)
    end if
    call mpi90_broadcast (nch, root, domain, error)
    if (me /= root) then
        if (associated (g%grids)) then
            if (size (g%grids) /= nch) then
                call vamp0_delete_grid (g%grids)
                deallocate (g%grids, g%weights, g%num_calls)
                allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
                call vamp_create_empty_grid (g%grids)
            end if
        else
            allocate (g%grids(nch), g%weights(nch), g%num_calls(nch))
            call vamp_create_empty_grid (g%grids)
        end if
    end if
    call vamp_broadcast_grid (g%grids, root, domain, error)
    call mpi90_broadcast (g%weights, root, domain, error)
    call mpi90_broadcast (g%num_calls, root, domain, error)
end subroutine vamp_broadcast_grids

```

186b *<Declaration of vampi procedures 164b>+≡*

```

public :: vamp_send_history
public :: vamp_receive_history

```

186c *<Implementation of vampi procedures 165b>+≡*

```

subroutine vamp_send_history (g, target, tag, domain, error)
  type(vamp_history), intent(in) :: g
  integer, intent(in) :: target, tag
  integer, intent(in), optional :: domain
  integer, intent(out), optional :: error
  integer, dimension(2) :: words
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
  call vamp_marshall_history_size (g, words(1), words(2))
  allocate (ibuf(words(1)), dbuf(words(2)))
  call vamp_marshall_history (g, ibuf, dbuf)
  call mpi90_send (words, target, tag, domain, error)
  call mpi90_send (ibuf, target, tag+1, domain, error)
  call mpi90_send (dbuf, target, tag+2, domain, error)
  deallocate (ibuf, dbuf)
end subroutine vamp_send_history

```

187 *⟨Implementation of vampi procedures 165b⟩+≡*

```

subroutine vamp_receive_history (g, source, tag, domain, status, error)
  type(vamp_history), intent(inout) :: g
  integer, intent(in) :: source, tag
  integer, intent(in), optional :: domain
  type(mpi90_status), intent(out), optional :: status
  integer, intent(out), optional :: error
  integer, dimension(2) :: words
  integer, dimension(:), allocatable :: ibuf
  real(kind=default), dimension(:), allocatable :: dbuf
  call mpi90_receive (words, source, tag, domain, status, error)
  allocate (ibuf(words(1)), dbuf(words(2)))
  call mpi90_receive (ibuf, source, tag+1, domain, status, error)
  call mpi90_receive (dbuf, source, tag+2, domain, status, error)
  call vamp_unmarshal_history (g, ibuf, dbuf)
  deallocate (ibuf, dbuf)
end subroutine vamp_receive_history

```

—6—

SELF TEST

6.1 No Mapping Mode

In this chapter we perform a test of the major features of Vamp. A function with many peaks is integrated with the traditional Vegas algorithm, using a multi-channel approach and in parallel. The function is constructed to have a known analytical integral (which is chosen to be one) in order to be able to gauge the accuracy of the result and error estimate.

6.1.1 Serial Test

```

188a <vamp_test.f90 188a>≡
      ! vamp_test.f90 --
      <Copyleft notice 1>
      <Module vamp_test_functions 188b>
      <Module vamp_tests 192b>

188b <Module vamp_test_functions 188b>≡
      module vamp_test_functions
        use kinds
        use constants, only: PI
        use coordinates
        use vamp, only: vamp_grid, vamp_multi_channel
        implicit none
        private
        public :: f, j, phi, ihp, w
        public :: lorentzian
        private :: lorentzian_normalized
        real(kind=default), public :: width
      contains
        <Implementation of vamp_test_functions procedures 189a>
      end module vamp_test_functions

```

$$\int_{x_1}^{x_2} dx \frac{1}{(x - x_0)^2 + a^2} = \frac{1}{a} \left(\text{atan} \left(\frac{x_2 - x_0}{a} \right) - \text{atan} \left(\frac{x_1 - x_0}{a} \right) \right) = N(x_0, x_1, x_2, a) \quad (6.1)$$

189a *⟨Implementation of vamp_test_functions procedures 189a⟩*≡

```

pure function lorentzian_normalized (x, x0, x1, x2, a) result (f)
  real(kind=default), intent(in) :: x, x0, x1, x2, a
  real(kind=default) :: f
  if (x1 <= x .and. x <= x2) then
    f = 1 / ((x - x0)**2 + a**2) &
      * a / (atan2 (x2 - x0, a) - atan2 (x1 - x0, a))
  else
    f = 0
  end if
end function lorentzian_normalized

```

$$\int d^n x f(x) = \int d\Omega_n r^{n-1} dr f(x) = 1 \quad (6.2)$$

189b *⟨Implementation of vamp_test_functions procedures 189a⟩*+≡

```

pure function lorentzian (x, x0, x1, x2, r0, a) result (f)
  real(kind=default), dimension(:), intent(in) :: x, x0, x1, x2
  real(kind=default), intent(in) :: r0, a
  real(kind=default) :: f
  real(kind=default) :: r, r1, r2
  integer :: n
  n = size (x)
  if (n > 1) then
    r = sqrt (dot_product (x-x0, x-x0))
    r1 = 0.4_default
    r2 = min (minval (x2-x0), minval (x0-x1))
    if (r1 <= r .and. r <= r2) then
      f = lorentzian_normalized (r, r0, r1, r2, a) * r**(1-n) / surface (n)
    else
      f = 0
    end if
  else
    f = lorentzian_normalized (x(1), x0(1), x1(1), x2(1), a)
  endif
end function lorentzian

```

189c *⟨Implementation of vamp_test_functions procedures 189a⟩*+≡

```

pure function f (x, prc_index, weights, channel, grids) result (f_x)
  real(kind=default), dimension(:), intent(in) :: x
  integer, intent(in) :: prc_index
  real(kind=default), dimension(:), intent(in), optional :: weights

```

```

integer, intent(in), optional :: channel
type(vamp_grid), dimension(:), intent(in), optional :: grids
real(kind=default) :: f_x
real(kind=default), dimension(size(x)) :: minus_one, plus_one, zero, w_i, f_i
integer :: n, i
n = size(x)
minus_one = -1
zero = 0
plus_one = 1
w_i = 1
do i = 1, n
  if (all (abs (x(i+1:)) <= 1)) then
    f_i = lorentzian (x(1:i), zero(1:i), minus_one(1:i), plus_one(1:i), &
      0.7_default, width) &
      / 2.0_default**(n-i)
  else
    f_i = 0
  end if
end do
f_x = dot_product (w_i, f_i) / sum (w_i)
end function f

```

190 *⟨Implementation of vamp_test_functions procedures 189a⟩+≡*

```

pure function phi (xi, channel) result (x)
  real(kind=default), dimension(:), intent(in) :: xi
  integer, intent(in) :: channel
  real(kind=default), dimension(size(xi)) :: x
  real(kind=default) :: r
  real(kind=default), dimension(0) :: dummy
  integer :: n
  n = size(x)
  if (channel == 1) then
    x = xi
  else if (channel == 2) then
    r = (xi(1) + 1) / 2 * sqrt (2.0_default)
    x(1:2) = spherical_cos_to_cartesian (r, PI * xi(2), dummy)
    x(3:) = xi(3:)
  else if (channel < n) then
    r = (xi(1) + 1) / 2 * sqrt (real (channel, kind=default))
    x(1:channel) = spherical_cos_to_cartesian (r, PI * xi(2), xi(3:channel))
    x(channel+1:) = xi(channel+1:)
  else if (channel == n) then
    r = (xi(1) + 1) / 2 * sqrt (real (channel, kind=default))
    x = spherical_cos_to_cartesian (r, PI * xi(2), xi(3:))
  end if
end function phi

```

```

else
  x = 0
end if
end function phi

```

191a *⟨Implementation of vamp_test_functions procedures 189a⟩* \equiv

```

pure function ihp (x, channel) result (xi)
  real(kind=default), dimension(:), intent(in) :: x
  integer, intent(in) :: channel
  real(kind=default), dimension(size(x)) :: xi
  real(kind=default) :: r, phi
  integer :: n
  n = size(x)
  if (channel == 1) then
    xi = x
  else if (channel == 2) then
    call cartesian_to_spherical_cos (x(1:2), r, phi)
    xi(1) = 2 * r / sqrt (2.0_default) - 1
    xi(2) = phi / PI
    xi(3:) = x(3:)
  else if (channel < n) then
    call cartesian_to_spherical_cos (x(1:channel), r, phi, xi(3:channel))
    xi(1) = 2 * r / sqrt (real (channel, kind=default)) - 1
    xi(2) = phi / PI
    xi(channel+1:) = x(channel+1:)
  else if (channel == n) then
    call cartesian_to_spherical_cos (x, r, phi, xi(3:))
    xi(1) = 2 * r / sqrt (real (channel, kind=default)) - 1
    xi(2) = phi / PI
  else
    xi = 0
  end if
end function ihp

```

191b *⟨Implementation of vamp_test_functions procedures 189a⟩* \equiv

```

pure function j (x, prc_index, channel) result (j_x)
  real(kind=default), dimension(:), intent(in) :: x
  integer, intent(in) :: prc_index
  integer, intent(in) :: channel
  real(kind=default) :: j_x
  if (channel == 1) then
    j_x = 1
  else if (channel > 1) then
    j_x = 2 / sqrt (real (channel, kind=default)) ! 1/|dr/dξ1|
    j_x = j_x / PI ! 1/|dφ/dξ2|
  end if
end function j

```



```

        j_x = j_x * cartesian_to_spherical_cos_j (x(1:channel))
    else
        j_x = 0
    end if
end function j

```

192a *⟨Implementation of vamp_test_functions procedures 189a⟩* \equiv

```

pure function w (x, prc_index, weights, channel, grids) result (w_x)
    real(kind=default), dimension(:), intent(in) :: x
    integer, intent(in) :: prc_index
    real(kind=default), dimension(:), intent(in), optional :: weights
    integer, intent(in), optional :: channel
    type(vamp_grid), dimension(:), intent(in), optional :: grids
    real(kind=default) :: w_x
    w_x = vamp_multi_channel (f, prc_index, phi, ihp, j, x, weights, channel, grids)
end function w

```

192b *⟨Module vamp_tests 192b⟩* \equiv

```

module vamp_tests
    use kinds
    use exceptions
    use histograms
    use tao_random_numbers
    use coordinates
    use vamp
    use vamp_test_functions !NODEP!
    implicit none
    private
    ⟨Declaration of procedures in vamp_tests 192c⟩
contains
    ⟨Implementation of procedures in vamp_tests 193a⟩
end module vamp_tests

```

Verification

192c *⟨Declaration of procedures in vamp_tests 192c⟩* \equiv

```

! public :: check_jacobians, check_inverses, check_inverses3
public :: check_inverses, check_inverses3

```

192d *⟨Implementation of procedures in vamp_tests (broken?) 192d⟩* \equiv

```

subroutine check_jacobians (rng, region, weights, samples)
    type( tao_random_state ), intent(inout) :: rng
    real(kind=default), dimension(:, :), intent(in) :: region
    real(kind=default), dimension(:), intent(in) :: weights
    integer, intent(in) :: samples

```

```

real(kind=default), dimension(size(region,dim=2)) :: x
real(kind=default) :: d
integer :: ch
integer, parameter :: prc_index = 1
do ch = 1, size(weights)
    call vamp_check_jacobian (rng, samples, j, prc_index, phi, ch, region, d, x)
    print *, "channel", ch, ": delta(j)/j=", real(d), ", @x=", real (x)
end do
end subroutine check_jacobians

```

193a *⟨Implementation of procedures in vamp_tests 193a⟩*≡

```

subroutine check_inverses (rng, region, weights, samples)
    type(tao_random_state), intent(inout) :: rng
    real(kind=default), dimension(:,,:), intent(in) :: region
    real(kind=default), dimension(:), intent(in) :: weights
    integer, intent(in) :: samples
    real(kind=default), dimension(size(region,dim=2)) :: x1, x2, x_dx
    real(kind=default) :: dx, dx_max
    integer :: ch, i
    dx_max = 0
    x_dx = 0
    do ch = 1, size(weights)
        do i = 1, samples
            call tao_random_number (rng, x1)
            x2 = ihp (phi (x1, ch), ch)
            dx = sqrt (dot_product (x1-x2, x1-x2))
            if (dx > dx_max) then
                dx_max = dx
                x_dx = x1
            end if
        end do
        print *, "channel", ch, ": |x-x|=", real(dx), ", @x=", real (x_dx)
    end do
end subroutine check_inverses

```

193b *⟨Implementation of procedures in vamp_tests 193a⟩*+≡

```

subroutine check_inverses3 (rng, region, samples)
    type(tao_random_state), intent(inout) :: rng
    real(kind=default), dimension(:,,:), intent(in) :: region
    integer, intent(in) :: samples
    real(kind=default), dimension(size(region,dim=2)) :: x1, x2, x_dx, x_dj
    real(kind=default) :: r, phi, jac, caj, dx, dx_max, dj, dj_max
    real(kind=default), dimension(size(x1)-2) :: cos_theta
    integer :: i
    dx_max = 0

```

```

x_dx = 0
dj_max = 0
x_dj = 0
do i = 1, samples
  call tao_random_number (rng, x1)
  call cartesian_to_spherical_cos_2 (x1, r, phi, cos_theta, jac)
  call spherical_cos_to_cartesian_2 (r, phi, cos_theta, x2, caj)
  dx = sqrt (dot_product (x1-x2, x1-x2))
  dj = jac*caj - 1
  if (dx > dx_max) then
    dx_max = dx
    x_dx = x1
  end if
  if (dj > dj_max) then
    dj_max = dj
    x_dj = x1
  end if
end do
print *, "channel 3 : j*j-1=", real(dj), ", @x=", real (x_dj)
print *, "channel 3 : |x-x|=", real(dx), ", @x=", real (x_dx)
end subroutine check_inverses3

```

Integration

194a *<Declaration of procedures in vamp_tests 192c>+≡*
 public :: single_channel, multi_channel

194b *<Implementation of procedures in vamp_tests 193a>+≡*
 subroutine single_channel (rng, region, samples, iterations, &
 integral, standard_dev, chi_squared)
 type(tao_random_state), intent(inout) :: rng
 real(kind=default), dimension(:,:), intent(in) :: region
 integer, dimension(:), intent(in) :: samples, iterations
 real(kind=default), intent(out) :: integral, standard_dev, chi_squared
 type(vamp_grid) :: gr
 type(vamp_history), dimension(iterations(1)+iterations(2)) :: history
 integer, parameter :: PRC_INDEX = 1
 call vamp_create_history (history)
 call vamp_create_grid (gr, region, samples(1))
 call vamp_sample_grid (rng, gr, f, PRC_INDEX, iterations(1), history = history)
 call vamp_discard_integral (gr, samples(2))
 call vamp_sample_grid &
 (rng, gr, f, PRC_INDEX, iterations(2), &
 integral, standard_dev, chi_squared, &

```

        history = history(iterations(1)+1:)
    call vamp_write_grid (gr, "vamp_test.grid")
    call vamp_delete_grid (gr)
    call vamp_print_history (history, "single")
    call vamp_delete_history (history)
end subroutine single_channel

195a  ⟨Implementation of procedures in vamp_tests 193a⟩+≡
    subroutine multi_channel (rng, region, weights, samples, iterations, powers, &
        integral, standard_dev, chi_squared)
        type(tao_random_state), intent(inout) :: rng
        real(kind=default), dimension(:,:), intent(in) :: region
        real(kind=default), dimension(:), intent(inout) :: weights
        integer, dimension(:), intent(in) :: samples, iterations
        real(kind=default), dimension(:), intent(in) :: powers
        real(kind=default), intent(out) :: integral, standard_dev, chi_squared
        type(vamp_grids) :: grs
        ⟨Body of multi_channel 195b⟩
    end subroutine multi_channel

195b  ⟨Body of multi_channel 195b⟩≡
    type(vamp_history), dimension(iterations(1)+iterations(2)+size(powers)-1) :: &
        history
    type(vamp_history), dimension(size(history),size(weights)) :: histories
    integer :: it, nit
    integer, parameter :: PRC_INDEX = 1
    nit = size (powers)
    call vamp_create_history (history)
    call vamp_create_history (histories)
    call vamp_create_grids (grs, region, samples(1), weights)
    call vamp_sample_grids (rng, grs, w, PRC_INDEX, iterations(1) - 1, &
        history = history, histories = histories)
    call vamp_print_history (history, "multi")
    call vamp_print_history (histories, "multi")
    do it = 1, nit
        call vamp_sample_grids (rng, grs, w, PRC_INDEX, 1, &
            history = history(iterations(1)+it-1:), &
            histories = histories(iterations(1)+it-1:,:))
        call vamp_print_history (history(iterations(1)+it-1:), "multi")
        call vamp_print_history (histories(iterations(1)+it-1:,:), "multi")
        call vamp_refine_weights (grs, powers(it))
    end do
    call vamp_discard_integrals (grs, samples(2))
    call vamp_sample_grids &
        (rng, grs, w, PRC_INDEX, iterations(2), &

```

```

        integral, standard_dev, chi_squared, &
        history = history(iterations(1)+nit:), &
        histories = histories(iterations(1)+nit:,:))
call vamp_print_history (history(iterations(1)+nit:), "multi")
call vamp_print_history (histories(iterations(1)+nit:,:), "multi")
call vamp_write_grids (grs, "vamp_test.grids")
call vamp_delete_grids (grs)
call vamp_print_history (history, "multi")
call vamp_print_history (histories, "multi")
call vamp_delete_history (history)
call vamp_delete_history (histories)

```

Input/Output

196a *<Declaration of procedures in vamp_tests 192c>+≡*

```
public :: print_results
```

196b *<Implementation of procedures in vamp_tests 193a>+≡*

```

subroutine print_results (prefix, prev_ticks, &
    integral, std_dev, chi2, acceptable, failures)
    character(len=*), intent(in) :: prefix
    integer, intent(in) :: prev_ticks
    real(kind=default), intent(in) :: integral, std_dev, chi2, acceptable
    integer, intent(inout) :: failures
    integer :: ticks, ticks_per_second
    real(kind=default) :: pull
    call system_clock (ticks, ticks_per_second)
    pull = (integral - 1) / std_dev
    print "(1X,A,A,F6.2,A)", prefix, &
        ": time = ", real (ticks - prev_ticks) / ticks_per_second, " secs"
    print *, prefix, ":    int, err, chi2: ", &
        real (integral), real (std_dev), real (chi2)
    if (abs (pull) > acceptable) then
        failures = failures + 1
        print *, prefix, ": unacceptable pull:", real (pull)
    else
        print *, prefix, ":    acceptable pull:", real (pull)
    end if
end subroutine print_results

```

Main Program

196c *<vamp_test.f90 188a>+≡*

```
program vamp_test
```

```

use kinds
use tao_random_numbers
use coordinates
use divisions, only: DIVISIONS_RCS_ID
use vamp
use vamp_test_functions !NODEP!
use vamp_tests !NODEP!
implicit none
integer :: start_ticks
integer, dimension(2) :: iterations, samples
real(kind=default), dimension(2,5) :: region
real(kind=default), dimension(5) :: weight_vector
real(kind=default), dimension(10) :: powers
real(kind=default) :: single_integral, single_standard_dev, single_chi_squared
real(kind=default) :: multi_integral, multi_standard_dev, multi_chi_squared
type(tao_random_state) :: rng
real(kind=default), parameter :: ACCEPTABLE = 4
integer :: failures
failures = 0
call tao_random_create (rng, 0)
call system_clock (start_ticks)
call tao_random_seed (rng, start_ticks)
iterations = (/ 4, 3 /)
samples = (/ 20000, 200000 /)
region(1,:) = -1.0
region(2,:) = 1.0
width = 0.0001
print *, "Starting VAMP 1.0 self test..."
print *, "serial code"
print *, VAMP_RCS_ID
print *, DIVISIONS_RCS_ID
call system_clock (start_ticks)
call single_channel (rng, region, samples, iterations, &
    single_integral, single_standard_dev, single_chi_squared)
call print_results ("SINGLE", start_ticks, &
    single_integral, single_standard_dev, single_chi_squared, &
    10*ACCEPTABLE, failures)
weight_vector = 1
powers = 0.25_default
call system_clock (start_ticks)
call multi_channel (rng, region, weight_vector, samples, iterations, &
    powers, multi_integral, multi_standard_dev, multi_chi_squared)
call print_results ("MULTI", start_ticks, &

```

```

        multi_integral, multi_standard_dev, multi_chi_squared, &
        ACCEPTABLE, failures)
    call system_clock (start_ticks)
! call check_jacobians (rng, region, weight_vector, samples(1))
    call check_inverses (rng, region, weight_vector, samples(1))
    call check_inverses3 (rng, region, samples(1))
    if (failures == 0) then
        stop 0
    else if (failures == 1) then
        stop 1
    else
        stop 2
    end if
end program vamp_test

```

6.1.2 Parallel Test

198a \langle vampi_test.f90 198a $\rangle \equiv$
! vampi_test.f90 --
 \langle Copyleft notice 1 \rangle
 \langle Module vamp_test_functions 188b \rangle

The following is identical to `vamp_tests`, except for use `vampi`:

198b \langle vampi_test.f90 198a $\rangle + \equiv$
module vampi_tests
 use kinds
 use exceptions
 use histograms
 use tao_random_numbers
 use coordinates
 use vampi
 use vamp_test_functions !NODEP!
 implicit none
 private
 \langle Declaration of procedures in vamp_tests 192c \rangle
contains
 \langle Implementation of procedures in vamp_tests 193a \rangle
end module vampi_tests

198c \langle vampi_test.f90 198a $\rangle + \equiv$
program vampi_test
 use kinds
 use tao_random_numbers
 use coordinates

```

use divisions, only: DIVISIONS_RCS_ID
use vamp, only: VAMP_RCS_ID
use vampi
use mpi90
use vamp_test_functions !NODEP!
use vampi_tests !NODEP!
implicit none
integer :: num_proc, proc_id, start_ticks
logical :: perform_io
integer, dimension(2) :: iterations, samples
real(kind=default), dimension(2,5) :: region
real(kind=default), dimension(5) :: weight_vector
real(kind=default), dimension(10) :: powers
real(kind=default) :: single_integral, single_standard_dev, single_chi_squared
real(kind=default) :: multi_integral, multi_standard_dev, multi_chi_squared
type(tao_random_state) :: rng
integer :: iostat, command
character(len=72) :: command_line
integer, parameter :: &
    CMD_ERROR = -1, CMD_END = 0, &
    CMD_NOP = 1, CMD_SINGLE = 2, CMD_MULTI = 3, CMD_CHECK = 4
call tao_random_create (rng, 0)
call mpi90_init ()
call mpi90_size (num_proc)
call mpi90_rank (proc_id)
perform_io = (proc_id == 0)
call system_clock (start_ticks)
call tao_random_seed (rng, start_ticks + proc_id)
iterations = (/ 4, 3 /)
samples = (/ 20000, 200000 /)
samples = (/ 200000, 2000000 /)
region(1,:) = -1.0
region(2,:) = 1.0
width = 0.0001
if (perform_io) then
    print *, "Starting VAMP 1.0 self test..."
    if (num_proc > 1) then
        print *, "parallel code running on ", num_proc, " processors"
    else
        print *, "parallel code running serially"
    end if
    print *, VAMP_RCS_ID
    print *, VAMPI_RCS_ID

```



```

        print *, DIVISIONS_RCS_ID
    end if
    command_loop: do
        <Parse the commandline in vamp_test and set command (never defined)>
        call mpi90_broadcast (command, 0)
        call system_clock (start_ticks)
        select case (command)
            <Execute command in vamp_test (never defined)>
            case (CMD_END)
                exit command_loop
            case (CMD_NOP)
                ! do nothing
            case (CMD_ERROR)
                ! do nothing
        end select
    end do command_loop
    call mpi90_finalize ()
end program vampi_test

```

6.1.3 Output

200a *<vamp_test.out 200a>*≡

6.2 Mapped Mode

In this chapter we perform a test of the major features of Vamp. A function with many peaks is integrated with the traditional Vegas algorithm, using a multi-channel approach and in parallel. The function is constructed to have a known analytical integral (which is chosen to be one) in order to be able to gauge the accuracy of the result and error estimate.

6.2.1 Serial Test

200b *<vamp_test0.f90 200b>*≡
 ! vamp_test0.f90 --
<Copyleft notice 1>
<Module vamp_test0_functions 201>

Single Channel

The functions to be integrated are shared by the serial and the parallel incarnation of the code.

```

201 <Module vamp_test0_functions 201>≡
  module vamp_test0_functions
    use kinds
    use vamp, only: vamp_grid, vamp_multi_channel0
    implicit none
    private
    public :: f, g, phi, w
    public :: create_sample, delete_sample
    private :: f0, psi, g0, f_norm
    real(kind=default), dimension(:), allocatable, private :: c, x_min, x_max
    real(kind=default), dimension(:, :, :), allocatable, public :: x0, gamma
  contains
    <Implementation of vamp_test0_functions procedures 202a>
  end module vamp_test0_functions

```

We start from a model of n_p interfering resonances in one variable (cf. section ??)

$$f_0(x|x_{\min}, x_{\max}, x_0, \gamma) = \frac{1}{N(x_{\min}, x_{\max}, x_0, \gamma)} \left| \sum_{p=1}^{n_p} \frac{1}{x - x_{0,p} + i\gamma_p} \right|^2 \quad (6.3)$$

where

$$N(x_{\min}, x_{\max}, x_0, \gamma) = \int_{x_{\min}}^{x_{\max}} dx \left| \sum_{p=1}^{n_p} \frac{1}{x - x_{0,p} + i\gamma_p} \right|^2 \quad (6.4)$$

such that

$$\int_{x_{\min}}^{x_{\max}} dx f_0(x|x_{\min}, x_{\max}, x_0, \gamma) = 1 \quad (6.5)$$

NB: the $N(x_{\min}, x_{\max}, x_0, \gamma)$ should be calculated once and tabulated to save processing time, but we are lazy here.

$$\begin{aligned}
N(x_{\min}, x_{\max}, x_0, \gamma) &= \sum_{p=1}^{n_p} \int_{x_{\min}}^{x_{\max}} dx \left| \frac{1}{x - x_{0,p} + i\gamma_p} \right|^2 \\
&+ 2 \operatorname{Re} \sum_{p=1}^{n_p} \sum_{q=1}^{n_p} \int_{x_{\min}}^{x_{\max}} dx \frac{1}{x - x_{0,p} + i\gamma_p} \frac{1}{x - x_{0,q} - i\gamma_q} \quad (6.6)
\end{aligned}$$

202a \langle Implementation of vamp_test0_functions procedures 202a $\rangle \equiv$

```

pure function f0 (x, x_min, x_max, x0, g) result (f_x)
  real(kind=default), intent(in) :: x, x_min, x_max
  real(kind=default), dimension(:), intent(in) :: x0, g
  real(kind=default) :: f_x
  complex(kind=default) :: amp
  real(kind=default) :: norm
  integer :: i, j
  amp = sum (1.0 / cmplx (x - x0, g, kind=default))
  norm = 0
  do i = 1, size (x0)
    norm = norm + f_norm (x_min, x_max, x0(i), g(i), x0(i), g(i))
    do j = i + 1, size (x0)
      norm = norm + 2 * f_norm (x_min, x_max, x0(i), g(i), x0(j), g(j))
    end do
  end do
  f_x = amp * conjg (amp) / norm
end function f0

```

$$\int_{x_{\min}}^{x_{\max}} dx \frac{1}{x - x_{0,p} + i\gamma_p} \frac{1}{x - x_{0,q} - i\gamma_q} = \frac{1}{x_{0,p} - x_{0,q} - i\gamma_p - i\gamma_q} \left(\ln \left(\frac{x_{\max} - x_{0,p} + i\gamma_p}{x_{\min} - x_{0,p} + i\gamma_p} \right) - \ln \left(\frac{x_{\max} - x_{0,q} - i\gamma_q}{x_{\min} - x_{0,q} - i\gamma_q} \right) \right) \quad (6.7)$$

Don't even think of merging the logarithms: it will screw up the Riemann sheet.

202b \langle Implementation of vamp_test0_functions procedures 202a $\rangle + \equiv$

```

pure function f_norm (x_min, x_max, x0p, gp, x0q, gq) &
  result (norm)
  real(kind=default), intent(in) :: x_min, x_max, x0p, gp, x0q, gq
  real(kind=default) :: norm
  norm = real (( log ( cmplx (x_max - x0p, gp, kind=default) &
                        / cmplx (x_min - x0p, gp, kind=default)) &
                - log ( cmplx (x_max - x0q, - gq, kind=default) &
                        / cmplx (x_min - x0q, - gq, kind=default))) &
                / cmplx (x0p - x0q, - gp - gq, kind=default), &
                kind=default)
end function f_norm

```

Since we want to be able to do the integral of f analytically, it is most

convenient to take a weighted sum of products:

$$f(x_1, \dots, x_{n_d} | x_{\min}, x_{\max}, x_0, \gamma) = \frac{1}{\sum_{i=1}^{n_c} c_i} \sum_{i=1}^{n_c} c_i \prod_{j=1}^{n_d} f_0(x_j | x_{\min,j}, x_{\max,j}, x_{0,ij}, \gamma_{ij}) \quad (6.8)$$

Each summand is factorized and therefore very easily integrated by Vegas. A non-trivial sum is more realistic in this respect.

- 203a *⟨Implementation of vamp_test0_functions procedures 202a⟩+≡*
- ```

pure function f (x, prc_index, weights, channel, grids) result (f_x)
 real(kind=default), dimension(:), intent(in) :: x
 integer, intent(in) :: prc_index
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel
 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: f_x
 real(kind=default) :: fi_x
 integer :: i, j
 f_x = 0.0
 do i = 1, size (c)
 fi_x = 1.0
 do j = 1, size (x)
 if (all (gamma(:,i,j) > 0)) then
 fi_x = fi_x * f0 (x(j), x_min(j), x_max(j), &
 x0(:,i,j), gamma(:,i,j))
 else
 fi_x = fi_x / (x_max(j) - x_min(j))
 end if
 end do
 f_x = f_x + c(i) * fi_x
 end do
 f_x = f_x / sum (c)
end function f

```
- 203b *⟨Implementation of vamp\_test0\_functions procedures 202a⟩+≡*
- ```

subroutine delete_sample ()
  deallocate (c, x_min, x_max, x0, gamma)
end subroutine delete_sample

```
- 203c *⟨Implementation of vamp_test0_functions procedures 202a⟩+≡*
- ```

subroutine create_sample (num_poles, weights, region)
 integer, intent(in) :: num_poles
 real(kind=default), dimension(:), intent(in) :: weights

```

```

real(kind=default), dimension(:,,:), intent(in) :: region
integer :: nd, nc
nd = size (region, dim=2)
nc = size (weights)
allocate (c(nc), x_min(nd), x_max(nd))
allocate (x0(num_poles,nc,nd), gamma(num_poles,nc,nd))
x_min = region(1,:)
x_max = region(2,:)
c = weights
end subroutine create_sample

```

### Multi Channel

We start from the usual mapping for Lorentzian peaks

$$\begin{aligned} \psi(x_{\min}, x_{\max}, x_0, \gamma) : [x_{\min}, x_{\max}] &\rightarrow [x_{\min}, x_{\max}] \\ \xi \mapsto x = \psi(\xi | x_{\min}, x_{\max}, x_0, \gamma) \end{aligned} \quad (6.9)$$

where

$$\begin{aligned} \psi(\xi | x_{\min}, x_{\max}, x_0, \gamma) = &x_0 + \\ &\gamma \cdot \tan \left( \frac{\xi - x_{\min}}{x_{\max} - x_{\min}} \cdot \operatorname{atan} \frac{x_{\max} - x_0}{\gamma} - \frac{x_{\max} - \xi}{x_{\max} - x_{\min}} \cdot \operatorname{atan} \frac{x_0 - x_{\min}}{\gamma} \right) \end{aligned} \quad (6.10)$$

204 *⟨Implementation of vamp\_test0\_functions procedures 202a⟩* +≡

```

pure function psi (xi, x_min, x_max, x0, gamma) result (x)
 real(kind=default), intent(in) :: xi, x_min, x_max, x0, gamma
 real(kind=default) :: x
 x = x0 + gamma &
 * tan (((xi - x_min) * atan ((x_max - x0) / gamma) &
 - (x_max - xi) * atan ((x0 - x_min) / gamma)) &
 / (x_max - x_min))
end function psi

```

The inverse mapping is

$$\begin{aligned} \psi^{-1}(x_{\min}, x_{\max}, x_0, \gamma) : [x_{\min}, x_{\max}] &\rightarrow [x_{\min}, x_{\max}] \\ x \mapsto \xi = \psi^{-1}(x | x_{\min}, x_{\max}, x_0, \gamma) \end{aligned} \quad (6.11)$$

with

$$\begin{aligned} \psi^{-1}(x | x_{\min}, x_{\max}, x_0, \gamma) = & \\ & \frac{x_{\max} \left( \operatorname{atan} \frac{x_0 - x_{\min}}{\gamma} + \operatorname{atan} \frac{x - x_0}{\gamma} \right) + x_{\min} \left( \operatorname{atan} \frac{x_{\max} - x_0}{\gamma} + \operatorname{atan} \frac{x_0 - x}{\gamma} \right)}{\operatorname{atan} \frac{x_{\max} - x_0}{\gamma} + \operatorname{atan} \frac{x_0 - x_{\min}}{\gamma}} \end{aligned} \quad (6.12)$$

with Jacobian

$$\frac{d(\psi^{-1}(x|x_{\min}, x_{\max}, x_0, \gamma))}{dx} = \frac{x_{\max} - x_{\min}}{\operatorname{atan} \frac{x_{\max} - x_0}{\gamma} + \operatorname{atan} \frac{x_0 - x_{\min}}{\gamma}} \frac{\gamma}{(x - x_0)^2 + \gamma^2} \quad (6.13)$$

**205a** *⟨Implementation of vamp\_test0\_functions procedures 202a⟩*  $\equiv$

```

pure function g0 (x, x_min, x_max, x0, gamma) result (g_x)
 real(kind=default), intent(in) :: x, x_min, x_max, x0, gamma
 real(kind=default) :: g_x
 g_x = gamma / (atan ((x_max - x0) / gamma) - atan ((x_min - x0) / gamma)) &
 * (x_max - x_min) / ((x - x0)**2 + gamma**2)
end function g0

```

The function  $f$  has  $n_c n_p^{n_d}$  peaks and we need a channel for each one, plus a constant function for the background. We encode the position on the grid linearly:

**205b** *⟨Decode channel into ch and p(:) 205b⟩*  $\equiv$

```

ch = channel - 1
do j = 1, size (x)
 p(j) = 1 + modulo (ch, np)
 ch = ch / np
end do
ch = ch + 1

```

The map  $\phi$  is the direct product of  $\psi$ s:

**205c** *⟨Implementation of vamp\_test0\_functions procedures 202a⟩*  $\equiv$

```

pure function phi (xi, channel) result (x)
 real(kind=default), dimension(:), intent(in) :: xi
 integer, intent(in) :: channel
 real(kind=default), dimension(size(xi)) :: x
 integer, dimension(size(xi)) :: p
 integer :: j, ch, np, nch, nd, channels
 np = size (x0, dim = 1)
 nch = size (x0, dim = 2)
 nd = size (x0, dim = 3)
 channels = nch * np**nd
 if (channel >= 1 .and. channel <= channels) then
 ⟨Decode channel into ch and p(:) 205b⟩
 do j = 1, size (xi)
 if (all (gamma(:,ch,j) > 0)) then
 x(j) = psi (xi(j), x_min(j), x_max(j), &
 x0(p(j),ch,j), gamma(p(j),ch,j))
 else
 x = xi

```

```

 end if
 end do
 else if (channel == channels + 1) then
 x = xi
 else
 x = 0
 end if
end function phi

```

similarly for the Jacobians:

206a *⟨Implementation of vamp\_test0\_functions procedures 202a⟩*+≡

```

pure recursive function g (x, prc_index, channel) result (g_x)
 real(kind=default), dimension(:), intent(in) :: x
 integer, intent(in) :: prc_index
 integer, intent(in) :: channel
 real(kind=default) :: g_x
 integer, dimension(size(x)) :: p
 integer :: j, ch, np, nch, nd, channels
 np = size (x0, dim = 1)
 nch = size (x0, dim = 2)
 nd = size (x0, dim = 3)
 channels = nch * np**nd
 if (channel >= 1 .and. channel <= channels) then
 ⟨Decode channel into ch and p(:) 205b⟩
 g_x = 1.0
 do j = 1, size (x)
 if (all (gamma(:,ch,j) > 0)) then
 g_x = g_x * g0 (x(j), x_min(j), x_max(j), &
 x0(p(j),ch,j), gamma(p(j),ch,j))
 end if
 end do
 else if (channel == channels + 1) then
 g_x = 1.0
 else
 g_x = 0
 end if
end function g

```

206b *⟨Implementation of vamp\_test0\_functions procedures 202a⟩*+≡

```

pure function w (x, prc_index, weights, channel, grids) result (w_x)
 real(kind=default), dimension(:), intent(in) :: x
 integer, intent(in) :: prc_index
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel

```

```

 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: w_x
 w_x = vamp_multi_channel0 (f, prc_index, phi, g, x, weights, channel)
end function w

```

### *Driver Routines*

**207a**  $\langle \text{vamp\_test0.f90 200b} \rangle + \equiv$

```

module vamp_tests0
 $\langle \text{Modules used by vamp_tests0 207b} \rangle$
 use vamp
 implicit none
 private
 $\langle \text{Declaration of procedures in vamp_tests0 208a} \rangle$
contains
 $\langle \text{Implementation of procedures in vamp_tests0 208b} \rangle$
end module vamp_tests0

```

**207b**  $\langle \text{Modules used by vamp\_tests0 207b} \rangle \equiv$

```

use kinds
use exceptions
use histograms
use tao_random_numbers
use vamp_test0_functions !NODEP!

```

### *Verification*

**207c**  $\langle \text{Declaration of procedures in vamp\_tests0 (broken?) 207c} \rangle \equiv$

```

public :: check_jacobians

```

**207d**  $\langle \text{Implementation of procedures in vamp\_tests0 (broken?) 207d} \rangle \equiv$

```

subroutine check_jacobians (do_print, region, samples, rng)
 logical, intent(in) :: do_print
 real(kind=default), dimension(:,:), intent(in) :: region
 integer, dimension(:), intent(in) :: samples
 type(tao_random_state), intent(inout) :: rng
 real(kind=default), dimension(size(region,dim=2)) :: x
 real(kind=default) :: d
 integer :: ch
 do ch = 1, size(x0,dim=2) * size(x0,dim=1)**size(x0,dim=3) + 1
 call vamp_check_jacobian (rng, samples(1), g, phi, ch, region, d, x)
 if (do_print) then
 print *, ch, ": ", d, ", x = ", real (x)
 end if
 end do
end subroutine check_jacobians

```



```

 end do
 end subroutine check_jacobians

```

### *Integration*

208a *<Declaration of procedures in vamp\_tests0 208a>*≡

```

 public :: single_channel, multi_channel

```

208b *<Implementation of procedures in vamp\_tests0 208b>*≡

```

 subroutine single_channel (do_print, region, iterations, samples, rng, &
 acceptable, failures)
 logical, intent(in) :: do_print
 real(kind=default), dimension(:,:), intent(in) :: region
 integer, dimension(:), intent(in) :: iterations, samples
 type(tao_random_state), intent(inout) :: rng
 real(kind=default), intent(in) :: acceptable
 integer, intent(inout) :: failures
 type(vamp_grid) :: gr
 type(vamp_history), dimension(iterations(1)+iterations(2)) :: history
 real(kind=default) :: integral, standard_dev, chi_squared, pull
 integer, parameter :: PRC_INDEX = 1
 call vamp_create_history (history)
 call vamp_create_grid (gr, region, samples(1))
 call vamp_sample_grid (rng, gr, f, PRC_INDEX, iterations(1), history = history)
 call vamp_discard_integral (gr, samples(2))
 call vamp_sample_grid &
 (rng, gr, f, PRC_INDEX, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(iterations(1)+1:))
 call vamp_write_grid (gr, "vamp_test0.grid")
 call vamp_delete_grid (gr)
 call vamp_print_history (history, "single")
 call vamp_delete_history (history)
 pull = (integral - 1) / standard_dev
 if (do_print) then
 print *, " int, err, chi2:", integral, standard_dev, chi_squared
 end if
 if (abs (pull) > acceptable) then
 failures = failures + 1
 print *, " unacceptable pull:", pull
 else
 print *, " acceptable pull:", pull
 end if
 end subroutine single_channel

```

209a *⟨Implementation of procedures in vamp\_tests0 208b⟩*+≡

```

subroutine multi_channel (do_print, region, iterations, samples, rng, &
 acceptable, failures)
 logical, intent(in) :: do_print
 real(kind=default), dimension(:,:), intent(in) :: region
 integer, dimension(:), intent(in) :: iterations, samples
 type(tao_random_state), intent(inout) :: rng
 real(kind=default), intent(in) :: acceptable
 type(vamp_grids) :: grs
 integer, intent(inout) :: failures
 ⟨Body of multi_channel 195b⟩
end subroutine multi_channel

```

209b *⟨Body of multi\_channel 195b⟩*+≡

```

real(kind=default), &
 dimension(size(x0,dim=2)*size(x0,dim=1)**size(x0,dim=3)+1) :: &
 weight_vector
type(vamp_history), dimension(iterations(1)+iterations(2)+4) :: history
type(vamp_history), dimension(size(history),size(weight_vector)) :: histories
real(kind=default) :: integral, standard_dev, chi_squared, pull
integer :: it
integer, parameter :: PRC_INDEX = 1
weight_vector = 1.0
call vamp_create_history (history)
call vamp_create_history (histories)
call vamp_create_grids (grs, region, samples(1), weight_vector)
call vamp_sample_grids (rng, grs, w, PRC_INDEX, iterations(1) - 1, &
 history = history, histories = histories)

do it = 1, 5
 call vamp_sample_grids (rng, grs, w, PRC_INDEX, 1, &
 history = history(iterations(1)+it-1:), &
 histories = histories(iterations(1)+it-1:,:))
 call vamp_refine_weights (grs)
end do
call vamp_discard_integrals (grs, samples(2))
call vamp_sample_grids &
 (rng, grs, w, PRC_INDEX, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(iterations(1)+5:), &
 histories = histories(iterations(1)+5:,:))
call vamp_write_grids (grs, "vamp_test0.grids")
call vamp_delete_grids (grs)
call vamp_print_history (history, "multi")
call vamp_print_history (histories, "multi")

```

```

call vamp_delete_history (history)
call vamp_delete_history (histories)
if (do_print) then
 print *, integral, standard_dev, chi_squared
end if
pull = (integral - 1) / standard_dev
if (abs (pull) > acceptable) then
 failures = failures + 1
 print *, " unacceptable pull:", pull
else
 print *, " acceptable pull:", pull
end if

```

### *Event Generation*

210a *<Declaration of procedures in vamp\_tests0 208a>+≡*

```

public :: single_channel_generator, multi_channel_generator

```

210b *<Implementation of procedures in vamp\_tests0 208b>+≡*

```

subroutine single_channel_generator (do_print, region, iterations, samples, rng)
 logical, intent(in) :: do_print
 real(kind=default), dimension(:,,:), intent(in) :: region
 integer, dimension(:), intent(in) :: iterations, samples
 type(tao_random_state), intent(inout) :: rng
 type(vamp_grid) :: gr
 type(vamp_history), dimension(iterations(1)+iterations(2)) :: history
 type(histogram) :: unweighted, reweighted, weighted, weights
 type(exception) :: exc
 real(kind=default) :: weight, integral, standard_dev
 integer :: i
 real(kind=default), dimension(size(region,dim=2)) :: x
 integer, parameter :: PRC_INDEX = 1
 call vamp_create_grid (gr, region, samples(1))
 call vamp_sample_grid (rng, gr, f, PRC_INDEX, iterations(1), history = history)
 call vamp_discard_integral (gr, samples(2))
 call vamp_warmup_grid &
 (rng, gr, f, PRC_INDEX, iterations(2), history = history(iterations(1)+1:))
 call vamp_print_history (history, "single")
 call vamp_delete_history (history)
 call create_histogram (unweighted, region(1,1), region(2,1), 100)
 call create_histogram (reweighted, region(1,1), region(2,1), 100)
 call create_histogram (weighted, region(1,1), region(2,1), 100)
 call create_histogram (weights, 0.0_default, 10.0_default, 100)
 ! do i = 1, 1000000

```

```

do i = 1, 100
 call clear_exception (exc)
 call vamp_next_event (x, rng, gr, f, PRC_INDEX, exc = exc)
 call handle_exception (exc)
 call fill_histogram (unweighted, x(1))
 call fill_histogram (reweighted, x(1), 1.0_default / f (x, PRC_INDEX))
end do
integral = 0.0
standard_dev = 0.0
do i = 1, 10000
 call clear_exception (exc)
 call vamp_next_event (x, rng, gr, f, PRC_INDEX, weight, exc = exc)
 call handle_exception (exc)
 call fill_histogram (weighted, x(1), weight / f (x, PRC_INDEX))
 call fill_histogram (weights, x(1), weight)
 integral = integral + weight
 standard_dev = standard_dev + weight**2
end do
if (do_print) then
 print *, integral / (i-1), sqrt (standard_dev) / (i-1)
 call write_histogram (unweighted, "u_s.d")
 call write_histogram (reweighted, "r_s.d")
 call write_histogram (weighted, "w_s.d")
 call write_histogram (weights, "ws_s.d")
end if
call delete_histogram (unweighted)
call delete_histogram (reweighted)
call delete_histogram (weighted)
call delete_histogram (weights)
call vamp_delete_grid (gr)
end subroutine single_channel_generator

```

211 *<Implementation of procedures in vamp\_tests0 208b>+≡*

```

subroutine multi_channel_generator (do_print, region, iterations, samples, rng)
 logical, intent(in) :: do_print
 real(kind=default), dimension(:,,:), intent(in) :: region
 integer, dimension(:), intent(in) :: iterations, samples
 type(tao_random_state), intent(inout) :: rng
 type(vamp_grids) :: grs
 real(kind=default), &
 dimension(size(x0,dim=2)*size(x0,dim=1)**size(x0,dim=3)+1) :: &
 weight_vector
 type(vamp_history), dimension(iterations(1)+iterations(2)+4) :: history
 type(vamp_history), dimension(size(history),size(weight_vector)) :: histories

```

```

type(histogram) :: unweighted, reweighted, weighted, weights
type(exception) :: exc
real(kind=default) :: weight, integral, standard_dev
real(kind=default), dimension(size(region,dim=2)) :: x
character(len=5) :: pfx
integer :: it, i, j
integer, parameter :: PRC_INDEX = 1
weight_vector = 1.0
call vamp_create_history (history)
call vamp_create_history (histories)
call vamp_create_grids (grs, region, samples(1), weight_vector)
call vamp_sample_grids (rng, grs, w, PRC_INDEX, iterations(1) - 1, &
 history = history, histories = histories)
do it = 1, 5
 call vamp_sample_grids (rng, grs, w, PRC_INDEX, 1, &
 history = history(iterations(1)+it-1:), &
 histories = histories(iterations(1)+it-1:,:))
 call vamp_refine_weights (grs)
end do
call vamp_discard_integrals (grs, samples(2))
call vamp_warmup_grids &
 (rng, grs, w, PRC_INDEX, iterations(2), &
 history = history(iterations(1)+5:), &
 histories = histories(iterations(1)+5:,:))
call vamp_print_history (history, "multi")
call vamp_print_history (histories, "multi")
call vamp_delete_history (history)
call vamp_delete_history (histories)
!!! do i = 1, size (grs%grids)
!!! do j = 1, size (grs%grids(i)%div)
!!! write (pfx, "(I2.2,',' ,I2.2)") i, j
!!! call dump_division (grs%grids(i)%div(j), pfx)
!!! end do
!!! end do
call create_histogram (unweighted, region(1,1), region(2,1), 100)
call create_histogram (reweighted, region(1,1), region(2,1), 100)
call create_histogram (weighted, region(1,1), region(2,1), 100)
call create_histogram (weights, 0.0_default, 10.0_default, 100)
! do i = 1, 1000000
do i = 1, 100
 call clear_exception (exc)
 call vamp_next_event (x, rng, grs, f, PRC_INDEX, phi, exc = exc)
 call handle_exception (exc)

```

```

 call fill_histogram (unweighted, x(1))
 call fill_histogram (reweighted, x(1), 1.0_default / f (x, PRC_INDEX))
 end do
 integral = 0.0
 standard_dev = 0.0
 do i = 1, 10000
 call clear_exception (exc)
 call vamp_next_event (x, rng, grs, f, PRC_INDEX, phi, weight, exc = exc)
 call handle_exception (exc)
 call fill_histogram (weighted, x(1), weight / f (x, PRC_INDEX))
 call fill_histogram (weights, x(1), weight)
 integral = integral + weight
 standard_dev = standard_dev + weight**2
 end do
 if (do_print) then
 print *, integral / (i-1), sqrt (standard_dev) / (i-1)
 call write_histogram (unweighted, "u_m.d")
 call write_histogram (reweighted, "r_m.d")
 call write_histogram (weighted, "w_m.d")
 call write_histogram (weights, "ws_m.d")
 end if
 call delete_histogram (unweighted)
 call delete_histogram (reweighted)
 call delete_histogram (weighted)
 call delete_histogram (weights)
 call vamp_delete_grids (grs)
end subroutine multi_channel_generator

```

### *Main Program*

```

213 <vamp_test0.f90 200b>+≡
 program vamp_test0
 <Modules used by vamp_test0 215c>
 implicit none
 <Variables in vamp_test0 215a>
 do_print = .true.
 print *, "Starting VAMP 1.0 self test..."
 print *, "serial code"
 print *, VAMP_RCS_ID
 print *, DIVISIONS_RCS_ID
 call tao_random_create (rng, 0)
 call system_clock (ticks0)
 call tao_random_seed (rng, ticks0)

```

```

 <Set up integrand and region in vamp_test0 215e>
 <Execute tests in vamp_test0 214a>
 <Cleanup in vamp_test0 215f>
 if (failures == 0) then
 stop 0
 else if (failures == 1) then
 stop 1
 else
 stop 2
 end if
end program vamp_test0

214a <Execute tests in vamp_test0 214a>≡
 failures = 0
 call system_clock (ticks0)
 call single_channel (do_print, region, iterations, samples, rng, 10*ACCEPTABLE, failures)
 call system_clock (ticks, ticks_per_second)
 print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"

214b <Execute tests in vamp_test0 214a>+≡
 call system_clock (ticks0)
 call single_channel_generator &
 (do_print, region, iterations, samples, rng)
 call system_clock (ticks, ticks_per_second)
 print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"

214c <Execute tests in vamp_test0 214a>+≡
 call system_clock (ticks0)
 call multi_channel (do_print, region, iterations, samples, rng, ACCEPTABLE, failures)
 call system_clock (ticks, ticks_per_second)
 print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"

214d <Execute tests in vamp_test0 214a>+≡
 call system_clock (ticks0)
 call multi_channel_generator &
 (do_print, region, iterations, samples, rng)
 call system_clock (ticks, ticks_per_second)
 print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"

214e <Execute tests in vamp_test0 214a>+≡
 call system_clock (ticks0)
 ! call check_jacobians (do_print, region, samples, rng)

```

```

call system_clock (ticks, ticks_per_second)
print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"

```

215a *<Variables in vamp\_test0 215a>*≡  
       logical :: do\_print

215b *<Execute command 215b>*≡

215c *<Modules used by vamp\_test0 215c>*≡  
       use kinds  
       use tao\_random\_numbers  
       use divisions, only: DIVISIONS\_RCS\_ID  
       use vamp, only: VAMP\_RCS\_ID  
       use vamp\_test0\_functions !NODEP!  
       use vamp\_tests0 !NODEP!

215d *<Variables in vamp\_test0 215a>*+≡  
       integer :: i, j, ticks, ticks\_per\_second, ticks0  
       integer, dimension(2) :: iterations, samples  
       real(kind=default), dimension(:,,:), allocatable :: region  
       type( tao\_random\_state ) :: rng  
       real(kind=default), parameter :: ACCEPTABLE = 4  
       integer :: failures

215e *<Set up integrand and region in vamp\_test0 215e>*≡  
       iterations = (/ 4, 3 /)  
       samples = (/ 10000, 50000 /)  
       allocate (region(2,2))  
       region(1,:) = -1.0  
       region(2,:) = 2.0  
       call create\_sample &  
           (num\_poles = 2, weights = (/ 1.0\_default, 2.0\_default /), region = region)  
       do i = 1, size (x0, dim=2)  
         do j = 1, size (x0, dim=3)  
           call tao\_random\_number (rng, x0(:,i,j))  
         end do  
       end do  
       gamma = 0.001  
       x0(1,::) = 0.2  
       x0(2,::) = 0.8

215f *<Cleanup in vamp\_test0 215f>*≡  
       call delete\_sample ()  
       deallocate (region)



## 6.2.2 Parallel Test

```
216a <vampi_test0.f90 216a>≡
 ! vampi_test0.f90 --
 <Copyleft notice 1>
 <Module vamp_test0_functions 201>
 module vamp_tests0
 <Modules used by vamp_tests0 207b>
 use vampi
 use mpi90
 implicit none
 private
 <Declaration of procedures in vamp_tests0 208a>
 contains
 <Implementation of procedures in vamp_tests0 208b>
 end module vamp_tests0

216b <vampi_test0.f90 216a>+≡
 program vampi_test0
 <Modules used by vamp_test0 215c>
 use mpi90
 use vampi, only: VAMPI_RCS_ID
 implicit none
 <Variables in vamp_test0 215a>
 integer :: num_proc, proc_id
 call mpi90_init ()
 call mpi90_size (num_proc)
 call mpi90_rank (proc_id)
 if (proc_id == 0) then
 do_print = .true.
 print *, "Starting VAMP 1.0 self test..."
 if (num_proc > 1) then
 print *, "parallel code running on ", num_proc, " processors"
 else
 print *, "parallel code running serially"
 end if
 print *, VAMP_RCS_ID
 print *, VAMPI_RCS_ID
 print *, DIVISIONS_RCS_ID
 else
 do_print = .false.
 end if
 call tao_random_create (rng, 0)
 call system_clock (ticks0)
```

```

call tao_random_seed (rng, ticks0 + proc_id)
<Set up integrand and region in vamp_test0 215e>
call mpi90_broadcast (x0, 0)
call mpi90_broadcast (gamma, 0)
command_loop: do
 if (proc_id == 0) then
 <Read command line and decode it as command (never defined)>
 end if
 call mpi90_broadcast (command, 0)
 call system_clock (ticks0)
 <Execute command 215b>
 call system_clock (ticks, ticks_per_second)
 if (proc_id == 0) then
 print "(1X,A,F6.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"
 end if
end do command_loop
<Cleanup in vamp_test0 215f>
call mpi90_finalize ()
if (proc_id == 0) then
 print *, "bye."
end if
end program vampi_test0

```

### 6.2.3 Output

217 <vamp\_test0.out 217>≡

# —7—

## APPLICATION

### 7.1 *Cross section*

```
218a <application.f90 218a>≡
! application.f90 --
<Copyleft notice 1>
module cross_section
 use kinds
 use constants
 use utils
 use kinematics
 use tao_random_numbers
 use products, only: dot
 use helicity
 use vamp, only: vamp_grid, vamp_probability
 implicit none
 private
 <Declaration of cross_section procedures 219d>
 <Types in cross_section 224b>
 <Variables in cross_section 218b>
contains
 <Implementation of cross_section procedures 220a>
end module cross_section

218b <Variables in cross_section 218b>≡
real(kind=default), private, parameter :: &
 MA_0 = 0.0, &
 MB_0 = 0.0, &
 M1_0 = 0.0, &
 M2_0 = 0.0, &
 M3_0 = 0.0, &
```

```

 S_0 = 200.0 ** 2
219a <XXX Variables in cross_section 219a>≡
 real(kind=default), private, parameter :: &
 MA_0 = 0.01, &
 MB_0 = 0.01, &
 M1_0 = 0.01, &
 M2_0 = 0.01, &
 M3_0 = 0.01, &
 S_0 = 200.0 ** 2
219b <XXX Variables in cross_section 219a>+≡
 real(kind=default), private, parameter :: &
 S1_MIN_0 = 0.0 ** 2, &
 S2_MIN_0 = 0.0 ** 2, &
 S3_MIN_0 = 0.0 ** 2, &
 T1_MIN_0 = 0.0 ** 2, &
 T2_MIN_0 = 0.0 ** 2
219c <Variables in cross_section 218b>+≡
 real(kind=default), private, parameter :: &
 S1_MIN_0 = 1.0 ** 2, &
 S2_MIN_0 = 1.0 ** 2, &
 S3_MIN_0 = 1.0 ** 2, &
 T1_MIN_0 = 10.0 ** 2, &
 T2_MIN_0 = 10.0 ** 2
219d <Declaration of cross_section procedures 219d>≡
 private :: cuts
219e <XXX Implementation of cross_section procedures 219e>≡
 pure function cuts (k1, k2, p1, p2, q) result (inside)
 real(kind=default), dimension(0:), intent(in) :: k1, k2, p1, p2, q
 logical :: inside
 inside = (abs (dot (k1 - q, k1 - q)) >= T1_MIN_0) &
 .and. (abs (dot (k2 - q, k2 - q)) >= T2_MIN_0) &
 .and. (abs (dot (p1 + q, p1 + q)) >= S1_MIN_0) &
 .and. (abs (dot (p2 + q, p2 + q)) >= S2_MIN_0) &
 .and. (abs (dot (p1 + p2, p1 + p2)) >= S3_MIN_0)
 end function cuts
219f <Variables in cross_section 218b>+≡
 real(kind=default), private, parameter :: &
 E_MIN = 1.0, &
 COSTH_SEP_MAX = 0.99, &
 COSTH_BEAM_MAX = 0.99

```

220a  $\langle$ Implementation of cross\_section procedures 220a $\rangle \equiv$

```

pure function cuts (k1, k2, p1, p2, q) result (inside)
 real(kind=default), dimension(0:), intent(in) :: k1, k2, p1, p2, q
 logical :: inside
 real(kind=default), dimension(3) :: p1n, p2n, qn
 inside = .false.
 if ((p1(0) < E_MIN) .or. (p2(0) < E_MIN) .or. (q(0) < E_MIN)) then
 return
 end if
 p1n = p1(1:3) / sqrt (dot_product (p1(1:3), p1(1:3)))
 p2n = p2(1:3) / sqrt (dot_product (p2(1:3), p2(1:3)))
 qn = q(1:3) / sqrt (dot_product (q(1:3), q(1:3)))
 if ((abs (qn(3)) > COSTH_BEAM_MAX) &
 .or. (abs (p1n(3)) > COSTH_BEAM_MAX) &
 .or. (abs (p2n(3)) > COSTH_BEAM_MAX)) then
 return
 end if
 if (dot_product (p1n, qn) > COSTH_SEP_MAX) then
 return
 end if
 if (dot_product (p2n, qn) > COSTH_SEP_MAX) then
 return
 end if
 if (dot_product (p1n, p2n) > COSTH_SEP_MAX) then
 return
 end if
 inside = .true.
end function cuts

```

220b  $\langle$ Implementation of cross\_section procedures 220a $\rangle + \equiv$

```

function xsect (k1, k2, p1, p2, q) result (xs)
 real(kind=default), dimension(0:), intent(in) :: k1, k2, p1, p2, q
 real(kind=default) :: xs
 complex(kind=default), dimension(-1:1,-1:1,-1:1,-1:1,-1:1) :: amp
 !!! xs = 1.0_double / phase_space_volume (3, k1(0) + k2(0))
 !!! xs = 1.0_double / dot (p1 + q, p1 + q) &
 !!! + 1.0_double / dot (p2 + q, p2 + q)
 !!! return
 amp = nneeg (k1, k2, p1, p2, q)
 xs = sum (amp(-1:1:2,-1:1:2,-1:1:2,-1:1:2,-1:1:2) &
 * conjg (amp(-1:1:2,-1:1:2,-1:1:2,-1:1:2,-1:1:2)))
end function xsect

```

220c  $\langle$ Declaration of cross\_section procedures 219d $\rangle + \equiv$

```

private :: xsect

```

$$\begin{aligned}
\phi : [0, 1]^{\otimes 5} &\rightarrow [(m_2 + m_3)^2, (\sqrt{s} - m_1)^2] \otimes [t_1^{\min}(s_2), t_1^{\max}(s_2)] \\
&\otimes [0, 2\pi] \otimes [-1, 1] \otimes [0, 2\pi] \\
(x_1, \dots, x_5) &\mapsto (s_2, t_1, \phi, \cos \theta_3, \phi_3) \\
&= (s_2(x_1), x_2 t_1^{\max}(s_2) + (1 - x_2) t_1^{\min}(s_2), 2\pi x_3, 2x_4 - 1, 2\pi x_5)
\end{aligned} \tag{7.1}$$

where

$$\begin{aligned}
&t_1^{\max/\min}(s_2) \\
&= m_a^2 + m_1^2 - \frac{(s + m_a^2 - m_b^2)(s - s_2 + m_1^2) \mp \sqrt{\lambda(s, m_a^2, m_b^2)\lambda(s, s_2, m_1^2)}}{2s}
\end{aligned} \tag{7.2}$$

**221a**  $\langle \text{Set } (s_2, t_1, \phi, \cos \theta_3, \phi_3) \text{ from } (x_1, \dots, x_5) \text{ 221a} \rangle \equiv$   

```

! s2_min = S1_MIN_0
s2_min = (m2 + m3)**2
s2_max = (sqrt(s) - m1)**2
s2 = s2_max * x(1) + s2_min * (1 - x(1))
t1_min = ma**2 + m1**2 - ((s + ma**2 - mb**2) * (s - s2 + m1**2) &
+ sqrt(lambda(s, ma**2, mb**2) * lambda(s, s2, m1**2))) / (2*s)
t1_max = ma**2 + m1**2 - ((s + ma**2 - mb**2) * (s - s2 + m1**2) &
- sqrt(lambda(s, ma**2, mb**2) * lambda(s, s2, m1**2))) / (2*s)
t1 = t1_max * x(2) + t1_min * (1 - x(2))
phi = 2*PI * x(3)
cos_theta3 = 2 * x(4) - 1
phi3 = 2*PI * x(5)

```

**221b**  $\langle \text{Set } (s_2, t_1, \phi, \cos \theta_3, \phi_3) \text{ from } (x_1, \dots, x_5) \text{ (massless case) 221b} \rangle \equiv$   

```

! s2_min = S1_MIN_0
s2_min = 0
s2_max = s
s2 = s2_max * x(1) + s2_min * (1 - x(1))
t1_min = - (s - s2)
t1_max = 0
t1 = t1_max * x(2) + t1_min * (1 - x(2))
phi = 2*PI * x(3)
cos_theta3 = 2 * x(4) - 1
phi3 = 2*PI * x(5)

```

$$J_\phi(x_1, \dots, x_5) = \begin{vmatrix} \frac{\partial s_2}{\partial x_1} & \frac{\partial t_1}{\partial x_1} \\ \frac{\partial s_2}{\partial x_2} & \frac{\partial t_1}{\partial x_2} \end{vmatrix} \cdot 8\pi^2 \tag{7.3}$$

i.e.

$$J_\phi(x_1, \dots, x_5) = 8\pi^2 \cdot \left| \frac{ds_2}{dx_1} \right| \cdot (t_1^{\max}(s_2) - t_1^{\min}(s_2)) \tag{7.4}$$

```

222a $\langle \text{Adjust Jacobian } 222a \rangle \equiv$
 p%jacobian = p%jacobian &
 * (8.0 * PI**2 * (s2_max - s2_min) * (t1_max - t1_min))

222b $\langle \text{Implementation of cross_section procedures } 220a \rangle + \equiv$
 pure function phase_space (x, channel) result (p)
 real(kind=default), dimension(:), intent(in) :: x
 integer, intent(in) :: channel
 type(LIPS3) :: p
 real(kind=default) :: &
 ma, mb, m1, m2, m3, s, t1, s2, phi, cos_theta3, phi3
 real(kind=default) :: s2_min, s2_max, t1_min, t1_max
 s = S_0
 $\langle m_a \leftrightarrow m_b, m_1 \leftrightarrow m_2 \text{ for channel \#1 } 222c \rangle$
 $\langle \text{Set } (s_2, t_1, \phi, \cos \theta_3, \phi_3) \text{ from } (x_1, \dots, x_5) \text{ } 221a \rangle$
 p = two_to_three (s, t1, s2, phi, cos_theta3, phi3, ma, mb, m1, m2, m3)
 $\langle \text{Adjust Jacobian } 222a \rangle$
 $\langle p_1 \leftrightarrow p_2 \text{ for channel \#2 } 223a \rangle$
 end function phase_space

222c $\langle m_a \leftrightarrow m_b, m_1 \leftrightarrow m_2 \text{ for channel \#1 } 222c \rangle \equiv$
 select case (channel)
 case (1)
 ma = MA_0
 mb = MB_0
 m1 = M1_0
 m2 = M2_0
 m3 = M3_0
 case (2)
 ma = MB_0
 mb = MA_0
 m1 = M2_0
 m2 = M1_0
 m3 = M3_0
 case (3)
 ma = MA_0
 mb = MB_0
 m1 = M3_0
 m2 = M2_0
 m3 = M1_0
 case default
 ma = MA_0
 mb = MB_0
 m1 = M1_0
 m2 = M2_0

```

```

 m3 = M3_0
 end select
223a $\langle p_1 \leftrightarrow p_2 \text{ for channel \#2 223a} \rangle \equiv$
 select case (channel)
 case (1)
 ! OK
 case (2)
 call swap (p%p(1,:), p%p(2,:))
 case (3)
 call swap (p%p(1,:), p%p(3,:))
 case default
 ! OK
 end select
223b $\langle \text{Declaration of cross_section procedures 219d} \rangle + \equiv$
 private :: jacobian
223c $\langle \text{Implementation of cross_section procedures 220a} \rangle + \equiv$
 pure function jacobian (k1, k2, p1, p2, q) result (jac)
 real(kind=default), dimension(0:), intent(in) :: k1, k2, p1, p2, q
 real(kind=default) :: jac
 real(kind=default) :: ma_2, mb_2, m1_2, m2_2, m3_2
 real(kind=default) :: s, s2, s2_min, s2_max, t1_min, t1_max
 ma_2 = max (dot (k1, k1), 0.0_double)
 mb_2 = max (dot (k2, k2), 0.0_double)
 m1_2 = max (dot (p1, p1), 0.0_double)
 m2_2 = max (dot (p2, p2), 0.0_double)
 m3_2 = max (dot (q, q), 0.0_double)
 s = dot (k1 + k2, k1 + k2)
 s2 = dot (p2 + q, p2 + q)
 ! s2_min = S1_MIN_0
 s2_min = (sqrt (m2_2) + sqrt (m3_2))**2
 s2_max = (sqrt (s) - sqrt (m1_2))**2
 t1_min = ma_2 + m1_2 - ((s + ma_2 - mb_2) * (s - s2 + m1_2) &
 + sqrt (lambda (s, ma_2, mb_2) * lambda (s, s2, m1_2))) / (2*s)
 t1_max = ma_2 + m1_2 - ((s + ma_2 - mb_2) * (s - s2 + m1_2) &
 - sqrt (lambda (s, ma_2, mb_2) * lambda (s, s2, m1_2))) / (2*s)
 jac = 1.0 / ((2*PI)**5 * 32 * s2) &
 * sqrt (lambda (s2, m2_2, m3_2) / lambda (s, ma_2, mb_2)) &
 * (8.0 * PI**2 * (s2_max - s2_min) * (t1_max - t1_min))
 end function jacobian
223d $\langle \text{Declaration of cross_section procedures 219d} \rangle + \equiv$
 private :: phase_space, phase_space_massless

```



```

224a <Implementation of cross_section procedures 220a>+≡
 pure function phase_space_massless (x, channel) result (p)
 real(kind=default), dimension(:), intent(in) :: x
 integer, intent(in) :: channel
 type(LIPS3) :: p
 real(kind=default) :: s, t1, s2, phi, cos_theta3, phi3
 real(kind=default) :: s2_min, s2_max, t1_min, t1_max
 s = S_0
 <Set (s2, t1, φ, cos θ3, φ3) from (x1, ..., x5) (massless case) 221b>
 p = two_to_three (s, t1, s2, phi, cos_theta3, phi3)
 <Adjust Jacobian 222a>
 <p1 ↔ p2 for channel #2 223a>
 end function phase_space_massless

224b <Types in cross_section 224b>≡
 type, public :: LIPS3_m5i2a3
 ! private
 real(kind=default) :: ma, mb, m1, m2, m3
 real(kind=default) :: s, s2, t1
 real(kind=default) :: phi, cos_theta3, phi3
 real(kind=default) :: jacobian
 end type LIPS3_m5i2a3

224c <Types in cross_section 224b>+≡
 type, public :: x5
 ! private
 real(kind=default), dimension(5) :: x
 real(kind=default) :: jacobian
 end type x5

224d <Declaration of cross_section procedures 219d>+≡
 private :: invariants_from_p, invariants_to_p
 private :: invariants_from_x, invariants_to_x

224e <Implementation of cross_section procedures 220a>+≡
 pure function invariants_from_p (p, k1, k2) result (q)
 type(LIPS3), intent(in) :: p
 real(kind=default), dimension(0:), intent(in) :: k1, k2
 type(LIPS3_m5i2a3) :: q
 real(kind=default) :: ma_2, mb_2, m1_2, m2_2, m3_2
 real(kind=default), dimension(0:3) :: k1k2, p2p3, k1p1, p3_23
 k1k2 = k1 + k2
 k1p1 = - k1 + p%p(1,:)
 p2p3 = p%p(2,:) + p%p(3,:)
 ma_2 = max (dot (k1, k1), 0.0_double)
 mb_2 = max (dot (k2, k2), 0.0_double)

```

```

m1_2 = max (dot (p%p(1,:), p%p(1,:)), 0.0_double)
m2_2 = max (dot (p%p(2,:), p%p(2,:)), 0.0_double)
m3_2 = max (dot (p%p(3,:), p%p(3,:)), 0.0_double)
q%ma = sqrt (ma_2)
q%mb = sqrt (mb_2)
q%m1 = sqrt (m1_2)
q%m2 = sqrt (m2_2)
q%m3 = sqrt (m3_2)
q%s = dot (k1k2, k1k2)
q%s2 = dot (p2p3, p2p3)
q%t1 = dot (k1p1, k1p1)
q%phi = atan2 (p%p(1,2), p%p(1,1))
if (q%phi < 0) then
 q%phi = q%phi + 2*PI
end if
p3_23 = boost_momentum (p%p(3,:), p2p3)
q%cos_theta3 = p3_23(3) / sqrt (dot_product (p3_23(1:3), p3_23(1:3)))
q%phi3 = atan2 (p3_23(2), p3_23(1))
if (q%phi3 < 0) then
 q%phi3 = q%phi3 + 2*PI
end if
q%jacobian = 1.0 / ((2*PI)**5 * 32 * q%s2) &
 * sqrt (lambda (q%s2, m2_2, m3_2) / lambda (q%s, ma_2, mb_2))
end function invariants_from_p

```

225a *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

pure function invariants_to_p (p) result (q)
 type(LIPS3_m5i2a3), intent(in) :: p
 type(LIPS3) :: q
 q = two_to_three (p%s, p%t1, p%s2, p%phi, p%cos_theta3, p%phi3)
 q%jacobian = q%jacobian * p%jacobian
end function invariants_to_p

```

225b *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

pure function invariants_from_x (x, s, ma, mb, m1, m2, m3) result (p)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), intent(in) :: s, ma, mb, m1, m2, m3
 type(LIPS3_m5i2a3) :: p
 real(kind=default) :: s2_min, s2_max, t1_min, t1_max
 p%ma = ma
 p%mb = mb
 p%m1 = m1
 p%m2 = m2
 p%m3 = m3
 p%s = s

```

```

s2_min = (p%m2 + p%m3)**2
s2_max = (sqrt (p%s) - p%m1)**2
p%s2 = s2_max * x(1) + s2_min * (1 - x(1))
t1_min = p%ma**2 + p%m1**2 &
- ((p%s + p%ma**2 - p%mb**2) * (p%s - p%s2 + p%m1**2) &
+ sqrt (lambda (p%s, p%ma**2, p%mb**2) &
* lambda (p%s, p%s2, p%m1**2))) / (2*p%s)
t1_max = p%ma**2 + p%m1**2 &
- ((p%s + p%ma**2 - p%mb**2) * (p%s - p%s2 + p%m1**2) &
- sqrt (lambda (p%s, p%ma**2, p%mb**2) &
* lambda (p%s, p%s2, p%m1**2))) / (2*p%s)
p%t1 = t1_max * x(2) + t1_min * (1 - x(2))
p%phi = 2*PI * x(3)
p%cos_theta3 = 2 * x(4) - 1
p%phi3 = 2*PI * x(5)
p%jacobian = 8*PI**2 * (s2_max - s2_min) * (t1_max - t1_min)
end function invariants_from_x

```

226a *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

pure function invariants_to_x (p) result (x)
 type(LIPS3_m5i2a3), intent(in) :: p
 type(x5) :: x
 real(kind=default) :: s2_min, s2_max, t1_min, t1_max
 s2_min = (p%m2 + p%m3)**2
 s2_max = (sqrt (p%s) - p%m1)**2
 t1_min = p%ma**2 + p%m1**2 &
- ((p%s + p%ma**2 - p%mb**2) * (p%s - p%s2 + p%m1**2) &
+ sqrt (lambda (p%s, p%ma**2, p%mb**2) &
* lambda (p%s, p%s2, p%m1**2))) / (2*p%s)
 t1_max = p%ma**2 + p%m1**2 &
- ((p%s + p%ma**2 - p%mb**2) * (p%s - p%s2 + p%m1**2) &
- sqrt (lambda (p%s, p%ma**2, p%mb**2) &
* lambda (p%s, p%s2, p%m1**2))) / (2*p%s)
 x%x(1) = (p%s2 - s2_min) / (s2_max - s2_min)
 x%x(2) = (p%t1 - t1_min) / (t1_max - t1_min)
 x%x(3) = p%phi / (2*PI)
 x%x(4) = (p%cos_theta3 + 1) / 2
 x%x(5) = p%phi3 / (2*PI)
 x%jacobian = p%jacobian * 8*PI**2 * (s2_max - s2_min) * (t1_max - t1_min)
end function invariants_to_x

```

226b *⟨Declaration of cross\_section procedures 219d⟩*+≡

```

public :: sigma, sigma_raw, sigma_massless

```

226c *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

function sigma (x, weights, channel, grids) result (xs)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel
 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: xs
 real(kind=default), dimension(2,0:3) :: k
 type(LIPS3) :: p
 k(1,:) = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k(2,:) = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 if (present (channel)) then
 p = phase_space (x, channel)
 else
 p = phase_space (x, 0)
 end if
 if (cuts (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:))) then
 xs = xsect (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:)) &
 * jacobian (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:))
 !!! * p%jacobian
 else
 xs = 0.0
 end if
end function sigma

```

227a *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

function sigma_raw (k1, k2, p1, p2, q) result (xs)
 real(kind=default), dimension(0:), intent(in) :: k1, k2, p1, p2, q
 real(kind=default) :: xs
 if (cuts (k1, k2, p1, p2, q)) then
 xs = xsect (k1, k2, p1, p2, q)
 else
 xs = 0.0
 end if
end function sigma_raw

```

227b *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

function sigma_massless (x, weights, channel, grids) result (xs)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel
 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: xs
 real(kind=default), dimension(2,0:3) :: k
 type(LIPS3) :: p
 k(1,:) = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)

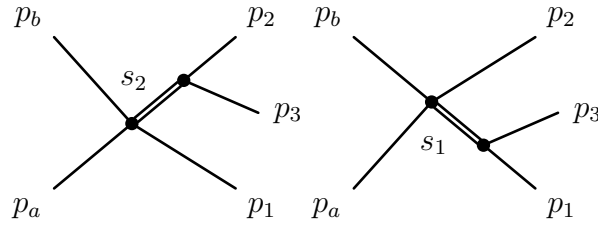
```

```

k(2,:) = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
p = phase_space_massless (x, 0)
if (cuts (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:))) then
 xs = xsect (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:)) &
 * p%jacobian
else
 xs = 0.0
end if
end function sigma_massless

```

228a  $\langle$ Declaration of cross\_section procedures 219d $\rangle + \equiv$   
public :: w



228b  $\langle$ Implementation of cross\_section procedures 220a $\rangle + \equiv$

```

function w (x, weights, channel, grids) result (w_x)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel
 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: w_x
 real(kind=default), dimension(size(weights)) :: g_x
 real(kind=default), dimension(2,0:3) :: k
 type(LIPS3) :: p
 integer :: ch
 if (present (channel)) then
 ch = channel
 else
 ch = 0
 end if
 k(1,:) = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k(2,:) = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 p = phase_space (x, abs (ch))
 g_x(1) = 1.0_double / jacobian (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:))
 g_x(2) = 1.0_double / jacobian (k(1,:), k(2,:), p%p(2,:), p%p(1,:), p%p(3,:))
 g_x(3) = 1.0_double / jacobian (k(1,:), k(2,:), p%p(3,:), p%p(2,:), p%p(1,:))
 if (ch > 0) then
 w_x = sigma_raw (k(1,:), k(2,:), p%p(1,:), p%p(2,:), p%p(3,:)) &

```

```

 / sum (weights * g_x)
else if (ch < 0) then
 w_x = g_x(-ch) / sum (weights * g_x)
else
 w_x = -1
end if
end function w

```

229a *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

function sigma_rambo (x, weights, channel, grids) result (xs)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:), intent(in), optional :: weights
 integer, intent(in), optional :: channel
 type(vamp_grid), dimension(:), intent(in), optional :: grids
 real(kind=default) :: xs
 real(kind=default), dimension(2,0:3) :: k
 real(kind=default), dimension(3,0:3) :: p
 k(1,:) = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k(2,:) = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 p = massless_isotropic_decay (sum (k(:,0)), reshape (x, (/ 3, 4 /)))
 if (cuts (k(1,:), k(2,:), p(1,:), p(2,:), p(3,:))) then
 xs = xsect (k(1,:), k(2,:), p(1,:), p(2,:), p(3,:)) &
 * phase_space_volume (size (p, dim = 1), sum (k(:,0)))
 else
 xs = 0.0
 end if
end function sigma_rambo

```

229b *⟨Declaration of cross\_section procedures 219d⟩*+≡

```

public :: sigma_rambo

```

229c *⟨Declaration of cross\_section procedures 219d⟩*+≡

```

public :: check_kinematics
private :: print_LIPS3_m5i2a3

```

229d *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

subroutine check_kinematics (rng)
 type(tao_random_state), intent(inout) :: rng
 real(kind=default), dimension(5) :: x
 real(kind=default), dimension(0:3) :: k1, k2
 type(x5) :: x1, x2
 type(LIPS3) :: p1, p2
 type(LIPS3_m5i2a3) :: q, q1, q2
 k1 = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k2 = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 call tao_random_number (rng, x)

```

```

q = invariants_from_x (x, S_0, MA_0, MB_0, M1_0, M2_0, M3_0)
p1 = invariants_to_p (q)
q1 = invariants_from_p (p1, k1, k2)
p2 = phase_space (x, 1)
q2 = invariants_from_p (p2, k1, k2)
x1 = invariants_to_x (q1)
x2 = invariants_to_x (q2)
print *, p1%jacobian, p2%jacobian, x1%jacobian, x2%jacobian
call print_lips3_m5i2a3 (q)
call print_lips3_m5i2a3 (q1)
call print_lips3_m5i2a3 (q2)
end subroutine check_kinematics

```

230a *<Implementation of cross\_section procedures 220a>+≡*

```

subroutine print_LIPS3_m5i2a3 (p)
 type(LIPS3_m5i2a3), intent(in) :: p
 print "(1x,5('m',a1,'=',e9.2,' '))", &
 'a', p%ma, 'b', p%mb, '1', p%m1, '2', p%m2, '3', p%m3
 print "(1x,'s=',e9.2,' s2=',e9.2,' t1=',e9.2)", &
 p%s, p%s2, p%t1
 print "(1x,'phi=',e9.2,' cos(th3)=',e9.2,' phi2=',e9.2)", &
 p%phi, p%cos_theta3, p%phi3
 print "(1x,'j=',e9.2)", &
 p%jacobian
end subroutine print_LIPS3_m5i2a3

```

230b *<Declaration of cross\_section procedures 219d>+≡*

```

public :: phi12, phi21, phi1, phi2
public :: g12, g21, g1, g2

```

230c *<Implementation of cross\_section procedures 220a>+≡*

```

pure function phi12 (x1, dummy) result (x2)
 real(kind=default), dimension(:), intent(in) :: x1
 integer, intent(in) :: dummy
 real(kind=default), dimension(size(x1)) :: x2
 type(LIPS3) :: p1, p2
 type(LIPS3_m5i2a3) :: q1, q2
 type(x5) :: x52
 real(kind=default), dimension(0:3) :: k1, k2
 k1 = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k2 = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 q1 = invariants_from_x (x1, S_0, MA_0, MB_0, M1_0, M2_0, M3_0)
 p1 = invariants_to_p (q1)
 p2%p(1,:) = p1%p(2,:)
 p2%p(2,:) = p1%p(1,:)

```

```

p2%p(3,:) = p1%p(3,:)
if (dummy < 0) then
 q2 = invariants_from_p (p2, k2, k1)
else
 q2 = invariants_from_p (p2, k1, k2)
end if
x52 = invariants_to_x (q2)
x2 = x52%x
end function phi12

```

231a *⟨Implementation of cross\_section procedures 220a⟩+≡*

```

pure function phi21 (x2, dummy) result (x1)
 real(kind=default), dimension(:), intent(in) :: x2
 integer, intent(in) :: dummy
 real(kind=default), dimension(size(x2)) :: x1
 type(LIPS3) :: p1, p2
 type(LIPS3_m5i2a3) :: q1, q2
 type(x5) :: x51
 real(kind=default), dimension(0:3) :: k1, k2
 k1 = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k2 = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 q2 = invariants_from_x (x2, S_0, MA_0, MB_0, M2_0, M1_0, M3_0)
 p2 = invariants_to_p (q2)
 p1%p(1,:) = p2%p(2,:)
 p1%p(2,:) = p2%p(1,:)
 p1%p(3,:) = p2%p(3,:)
 if (dummy < 0) then
 q1 = invariants_from_p (p1, k2, k1)
 else
 q1 = invariants_from_p (p1, k1, k2)
 end if
 x51 = invariants_to_x (q1)
 x1 = x51%x
end function phi21

```

231b *⟨Implementation of cross\_section procedures 220a⟩+≡*

```

pure function phi1 (x1) result (p1)
 real(kind=default), dimension(:), intent(in) :: x1
 type(LIPS3) :: p1
 type(LIPS3_m5i2a3) :: q1
 q1 = invariants_from_x (x1, S_0, MA_0, MB_0, M1_0, M2_0, M3_0)
 p1 = invariants_to_p (q1)
end function phi1

```

231c *⟨Implementation of cross\_section procedures 220a⟩+≡*



```

pure function phi2 (x2) result (p2)
 real(kind=default), dimension(:), intent(in) :: x2
 type(LIPS3) :: p2
 type(LIPS3_m5i2a3) :: q2
 q2 = invariants_from_x (x2, S_0, MA_0, MB_0, M2_0, M1_0, M3_0)
 p2 = invariants_to_p (q2)
end function phi2

```

232a *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

pure function g12 (x1) result (g)
 real(kind=default), dimension(:), intent(in) :: x1
 real(kind=default) :: g
 type(LIPS3) :: p1, p2
 type(LIPS3_m5i2a3) :: q1, q2
 type(x5) :: x52
 real(kind=default), dimension(0:3) :: k1, k2
 k1 = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k2 = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 q1 = invariants_from_x (x1, S_0, MA_0, MB_0, M1_0, M2_0, M3_0)
 p1 = invariants_to_p (q1)
 p2%p(1,:) = p1%p(2,:)
 p2%p(2,:) = p1%p(1,:)
 p2%p(3,:) = p1%p(3,:)
 q2 = invariants_from_p (p2, k2, k1)
 x52 = invariants_to_x (q2)
 g = x52%jacobian / p1%jacobian
end function g12

```

232b *⟨Implementation of cross\_section procedures 220a⟩*+≡

```

pure function g21 (x2) result (g)
 real(kind=default), dimension(:), intent(in) :: x2
 real(kind=default) :: g
 type(LIPS3) :: p1, p2
 type(LIPS3_m5i2a3) :: q1, q2
 type(x5) :: x51
 real(kind=default), dimension(0:3) :: k1, k2
 k1 = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k2 = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 q2 = invariants_from_x (x2, S_0, MA_0, MB_0, M2_0, M1_0, M3_0)
 p2 = invariants_to_p (q2)
 p1%p(1,:) = p2%p(2,:)
 p1%p(2,:) = p2%p(1,:)
 p1%p(3,:) = p2%p(3,:)
 q1 = invariants_from_p (p1, k2, k1)
 x51 = invariants_to_x (q1)

```

```

 g = x51%jacobian / p2%jacobian
end function g21

```

233a *<Implementation of cross\_section procedures 220a>+≡*

```

pure function g1 (x1) result (g)
 real(kind=default), dimension(:), intent(in) :: x1
 real(kind=default) :: g
 type(LIPS3) :: p1
 type(LIPS3_m5i2a3) :: q1
 q1 = invariants_from_x (x1, S_0, MA_0, MB_0, M1_0, M2_0, M3_0)
 p1 = invariants_to_p (q1)
 g = 1 / p1%jacobian
end function g1

```

233b *<Implementation of cross\_section procedures 220a>+≡*

```

pure function g2 (x2) result (g)
 real(kind=default), dimension(:), intent(in) :: x2
 real(kind=default) :: g
 type(LIPS3) :: p2
 type(LIPS3_m5i2a3) :: q2
 q2 = invariants_from_x (x2, S_0, MA_0, MB_0, M2_0, M1_0, M3_0)
 p2 = invariants_to_p (q2)
 g = 1 / p2%jacobian
end function g2

```

233c *<Declaration of cross\_section procedures 219d>+≡*

```

public :: wx

```

233d *<Implementation of cross\_section procedures 220a>+≡*

```

function wx (x, weights, channel, grids) result (w_x)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), dimension(:), intent(in) :: weights
 integer, intent(in) :: channel
 type(vamp_grid), dimension(:), intent(in) :: grids
 real(kind=default) :: w_x
 real(kind=default), dimension(size(weights)) :: g_x, p_q
 real(kind=default), dimension(size(x)) :: x1, x2
 real(kind=default), dimension(2,0:3) :: k
 type(LIPS3) :: q
 k(1,:) = (/ 100.0_double, 0.0_double, 0.0_double, 100.0_double /)
 k(2,:) = (/ 100.0_double, 0.0_double, 0.0_double, -100.0_double /)
 select case (abs (channel))
 case (1)
 x1 = x
 x2 = phi12 (x, 0)
 q = phi1 (x1)

```

```

case (2)
 x1 = phi21 (x, 0)
 x2 = x
 q = phi2 (x2)
end select
p_q(1) = vamp_probability (grids(1), x1)
p_q(2) = vamp_probability (grids(2), x2)
g_x(1) = p_q(1) * g1 (x1)
g_x(2) = p_q(2) * g2 (x2)
g_x = g_x / p_q(abs(channel))
if (channel > 0) then
 w_x = sigma_raw (k(1,:), k(2,:), q%p(1,:), q%p(2,:), q%p(3,:)) &
 / dot_product (weights, g_x)
else if (channel < 0) then
 w_x = vamp_probability (grids(-channel), x) / dot_product (weights, g_x)
else
 w_x = 0
end if
end function wx

```

```

234 <application.f90 218a>+≡
program application
 use kinds
 use utils
 use vampi
 use mpi90
 use linalg
 use exceptions
 use kinematics, only: phase_space_volume
 use cross_section !NODEP!
 use tao_random_numbers
 implicit none
 type(vamp_grid) :: gr
 type(vamp_grids) :: grs
 real(kind=default), dimension(:,:), allocatable :: region
 real(kind=default) :: integral, standard_dev, chi_squared
 real(kind=default) :: &
 single_integral, single_standard_dev, &
 rambo_integral, rambo_standard_dev
 real(kind=default), dimension(2) :: weight_vector
 integer, dimension(2) :: calls, iterations
 type(vamp_history), dimension(100) :: history
 type(vamp_history), dimension(100,size(weight_vector)) :: histories
 type(exception) :: exc

```

```

type(tao_random_state) :: rng
real(kind=default), dimension(5) :: x
real(kind=default) :: jac
integer :: i
integer :: num_proc, proc_id, ticks, ticks0, ticks_per_second, command
character(len=72) :: command_line
integer, parameter :: &
 CMD_SINGLE = 1, &
 CMD_MULTI = 2, &
 CMD_ROTATING = 3, &
 CMD_RAMBO = 4, &
 CMD_COMPARE = 5, &
 CMD_MASSLESS = 6, &
 CMD_ERROR = 0
call mpi90_init ()
call mpi90_size (num_proc)
call mpi90_rank (proc_id)
call system_clock (ticks0)
call tao_random_create (rng, 0)
call tao_random_seed (rng, ticks0 + proc_id)
!!! call tao_random_seed (rng, proc_id)
call vamp_create_history (history, verbose = .true.)
call vamp_create_history (histories, verbose = .true.)
iterations = (/ 3, 4 /)
calls = (/ 10000, 100000 /)
if (proc_id == 0) then
 read *, command_line
 if (command_line == "single") then
 command = CMD_SINGLE
 else if (command_line == "multi") then
 command = CMD_MULTI
 else if (command_line == "rotating") then
 command = CMD_ROTATING
 else if (command_line == "rambo") then
 command = CMD_RAMBO
 else if (command_line == "compare") then
 command = CMD_COMPARE
 else if (command_line == "massless") then
 command = CMD_MASSLESS
 else
 command = CMD_ERROR
 end if
end if

```

```

call mpi90_broadcast (command, 0)
call system_clock (ticks0)
select case (command)
case (CMD_SINGLE)
 ⟨Application in single channel mode 237⟩
case (CMD_MASSLESS)
 ⟨Application in massless single channel mode 238a⟩
case (CMD_MULTI)
 ⟨Application in multi channel mode 238b⟩
case (CMD_ROTATING)
 allocate (region(2,5))
 region(1,:) = 0.0
 region(2,:) = 1.0
 if (proc_id == 0) then
 print *, "rotating N/A yet ..."
 end if
case (CMD_RAMBO)
 ⟨Application in Rambo mode 239⟩
case (CMD_COMPARE)
 ⟨Application in single channel mode 237⟩
 single_integral = integral
 single_standard_dev = standard_dev
 ⟨Application in Rambo mode 239⟩
 if (proc_id == 0) then
 rambo_integral = integral
 rambo_standard_dev = standard_dev
 integral = &
 (single_integral / single_standard_dev**2 &
 + rambo_integral / rambo_standard_dev**2) &
 / (1.0_double / single_standard_dev**2 &
 + 1.0_double / rambo_standard_dev**2)
 standard_dev = 1.0_double &
 / sqrt (1.0_double / single_standard_dev**2 &
 + 1.0_double / rambo_standard_dev**2)
 chi_squared = &
 ((single_integral - integral)**2 / single_standard_dev**2) &
 + ((rambo_integral - integral)**2 / rambo_standard_dev**2)
 print *, "S&R: ", integral, standard_dev, chi_squared
 end if
case default
 if (proc_id == 0) then
 print *, "???: ", command
 !!! TO BE REMOVED !!!
 end if

```

```

call check_kinematics (rng)
allocate (region(2,5))
region(1,:) = 0
region(2,:) = 1
do i = 1, 10
 call tao_random_number (rng, x)
 call vamp_jacobian (phi12, 0, x, region, jac)
 print *, "12: ", jac, 1 / g12 (x), jac * g12 (x) - 1
 call vamp_jacobian (phi21, 0, x, region, jac)
 print *, "21: ", jac, 1 / g21 (x), jac * g21 (x) - 1
 print *, "1: ", real(x)
 print *, "2: ", real(phi12(phi21(x,0),0))
 print *, "2': ", real(phi12(phi21(x,-1),-1))
 print *, "3: ", real(phi21(phi12(x,0),0))
 print *, "3': ", real(phi21(phi12(x,-1),-1))
 print *, "2-1: ", real(phi12(phi21(x,0),0) - x)
 print *, "3-1: ", real(phi21(phi12(x,0),0) - x)
 print *, "a: ", real(phi12(x,0))
 print *, "a': ", real(phi12(x,-1))
 print *, "b: ", real(phi21(x,0))
 print *, "b': ", real(phi21(x,-1))
end do
deallocate (region)
! do i = 2, 5
! print *, i, phase_space_volume (i, 200.0_double)
! end do
end if
end select
if (proc_id == 0) then
 call system_clock (ticks, ticks_per_second)
 print "(1X,A,F8.2,A)", &
 "time = ", real (ticks - ticks0) / ticks_per_second, " secs"
end if
call mpi90_finalize ()
end program application

```

**237** *⟨Application in single channel mode 237⟩*≡

```

allocate (region(2,5))
region(1,:) = 0.0
region(2,:) = 1.0
call vamp_create_grid (gr, region, calls(1))
call clear_exception (exc)
call vamp_sample_grid &
 (rng, gr, sigma, iterations(1), history = history, exc = exc)

```

```

call handle_exception (exc)
call vamp_discard_integral (gr, calls(2))
call vamp_sample_grid &
 (rng, gr, sigma, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(iterations(1)+1:), exc = exc)
call handle_exception (exc)
call vamp_print_history (history, "single")
if (proc_id == 0) then
 print *, "SINGLE: ", integral, standard_dev, chi_squared
end if
call vamp_write_grid (gr, "application.grid")
call vamp_delete_grid (gr)
deallocate (region)

```

238a *Application in massless single channel mode 238a*≡

```

allocate (region(2,5))
region(1,:) = 0.0
region(2,:) = 1.0
call vamp_create_grid (gr, region, calls(1))
call clear_exception (exc)
call vamp_sample_grid &
 (rng, gr, sigma_massless, iterations(1), history = history, exc = exc)
call handle_exception (exc)
call vamp_discard_integral (gr, calls(2))
call vamp_sample_grid &
 (rng, gr, sigma_massless, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(iterations(1)+1:), exc = exc)
call handle_exception (exc)
call vamp_print_history (history, "single")
if (proc_id == 0) then
 print *, "M=0: ", integral, standard_dev, chi_squared
end if
call vamp_write_grid (gr, "application.grid")
call vamp_delete_grid (gr)
deallocate (region)

```

238b *Application in multi channel mode 238b*≡

```

allocate (region(2,5))
region(1,:) = 0.0
region(2,:) = 1.0
weight_vector = 1.0
if (proc_id == 0) then
 read *, weight_vector

```

```

end if
call mpi90_broadcast (weight_vector, 0)
weight_vector = weight_vector / sum (weight_vector)
call vamp_create_grids (grs, region, calls(1), weight_vector)
do i = 1, 3
 call clear_exception (exc)
 call vamp_sample_grids &
 (rng, grs, wx, iterations(1), &
 history = history(1+(i-1)*iterations(1):), &
 histories = histories(1+(i-1)*iterations(1):,:), exc = exc)
 call handle_exception (exc)
 call vamp_refine_weights (grs)
end do
call vamp_discard_integrals (grs, calls(2))
call vamp_sample_grids &
 (rng, grs, wx, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(3*iterations(1)+1:), &
 histories = histories(3*iterations(1)+1:,:), exc = exc)
call handle_exception (exc)
call vamp_print_history (history, "multi")
call vamp_print_history (histories, "multi")
if (proc_id == 0) then
 print *, "MULTI: ", integral, standard_dev, chi_squared
end if
call vamp_write_grids (grs, "application.grids")
call vamp_delete_grids (grs)
deallocate (region)

```

239 *Application in Rambo mode 239* ≡

```

allocate (region(2,12))
region(1,:) = 0.0
region(2,:) = 1.0
call vamp_create_grid (gr, region, calls(1))
call clear_exception (exc)
call vamp_sample_grid &
 (rng, gr, sigma_rambo, iterations(1), history = history, exc = exc)
call handle_exception (exc)
call vamp_discard_integral (gr, calls(2))
call vamp_sample_grid &
 (rng, gr, sigma_rambo, iterations(2), &
 integral, standard_dev, chi_squared, &
 history = history(iterations(1)+1:), exc = exc)
call handle_exception (exc)

```



```
call vamp_print_history (history, "rambo")
if (proc_id == 0) then
 print *, "RAMBO: ", integral, standard_dev, chi_squared
end if
call vamp_delete_grid (gr)
deallocate (region)
```

# —A—

## CONSTANTS

### *A.1 Kinds*

This borders on overkill, but it is the most portable way to get double precision in standard Fortran without relying on `kind (1.0D0)`. Currently, it is possible to change `double` to any other supported real kind. The MPI interface is a potential trouble source for such things, however.

```
241a <vamp_kinds.f90 241a>≡
 ! vamp_kinds.f90 --
 <Copyleft notice 1>
 module kinds
 implicit none
 integer, parameter, private :: single = &
 & selected_real_kind (precision(1.0), range(1.0))
 integer, parameter, private :: double = &
 & selected_real_kind (precision(1.0_single) + 1, range(1.0_single) + 1)
 integer, parameter, private :: quadruple = &
 & selected_real_kind (precision (1.0_double) + 1, range (1.0_double))
 integer, parameter, public :: default = double
 character(len=*), public, parameter :: KINDS_RCS_ID = &
 "$Id: kinds.nw 314 2010-04-17 20:32:33Z ohl $"
 end module kinds
```

### *A.2 Mathematical and Physical Constants*

```
241b <constants.f90 241b>≡
 ! constants.f90 --
 <Copyleft notice 1>
```

```
module constants
 use kinds
 implicit none
 private
 real(kind=default), public, parameter :: &
 PI = 3.1415926535897932384626433832795028841972_default
 character(len=*), public, parameter :: CONSTANTS_RCS_ID = &
 "$Id: constants.nw 314 2010-04-17 20:32:33Z ohl $"
end module constants
```

## —B—

# ERRORS AND EXCEPTIONS

Fortran95 does not allow *any* I/O in pure and elemental procedures, not even output to the unit \*. A stop statement is verboten as well. Therefore we have to use condition codes

```
243a <exceptions.f90 243a>≡
 ! exceptions.f90 --
 <Copyleft notice 1>
 module exceptions
 use kinds
 implicit none
 private
 <Declaration of exceptions procedures 244b>
 <Interfaces of exceptions procedures 358d>
 <Variables in exceptions 243c>
 <Declaration of exceptions types 243b>
 character(len=*, public, parameter :: EXCEPTIONS_RCS_ID = &
 "$Id: exceptions.nw 314 2010-04-17 20:32:33Z ohl $"
 contains
 <Implementation of exceptions procedures 244c>
 end module exceptions

243b <Declaration of exceptions types 243b>≡
 type, public :: exception
 integer :: level = EXC_NONE
 character(len=NAME_LENGTH) :: message = ""
 character(len=NAME_LENGTH) :: origin = ""
 end type exception

243c <Variables in exceptions 243c>≡
 integer, public, parameter :: &
 EXC_NONE = 0, &
 EXC_INFO = 1, &
 EXC_WARN = 2, &
```

```

 EXC_ERROR = 3, &
 EXC_FATAL = 4

```

244a *<Variables in exceptions 243c>+≡*

```

 integer, private, parameter :: EXC_DEFAULT = EXC_ERROR
 integer, private, parameter :: NAME_LENGTH = 64

```

244b *<Declaration of exceptions procedures 244b>≡*

```

 public :: handle_exception

```

244c *<Implementation of exceptions procedures 244c>≡*

```

 subroutine handle_exception (exc)
 type(exception), intent(inout) :: exc
 character(len=10) :: name
 if (exc%level > 0) then
 select case (exc%level)
 case (EXC_NONE)
 name = "(none)"
 case (EXC_INFO)
 name = "info"
 case (EXC_WARN)
 name = "warning"
 case (EXC_ERROR)
 name = "error"
 case (EXC_FATAL)
 name = "fatal"
 case default
 name = "invalid"
 end select
 print *, trim (exc%origin), ": ", trim(name), ": ", trim (exc%message)
 if (exc%level >= EXC_FATAL) then
 print *, "terminated."
 stop
 end if
 end if
 end subroutine handle_exception

```

244d *<Declaration of exceptions procedures 244b>+≡*

```

 public :: raise_exception, clear_exception, gather_exceptions

```

Raise an exception, but don't overwrite the messages in `exc` if it holds a more severe exception. This way we can accumulate error codes across procedure calls. We have `exc` optional to simplify life for the calling procedures, which might have it optional themselves.

244e *<Implementation of exceptions procedures 244c>+≡*

```

 elemental subroutine raise_exception (exc, level, origin, message)

```

```

type(exception), intent(inout), optional :: exc
integer, intent(in), optional :: level
character(len=*), intent(in), optional :: origin, message
integer :: local_level
if (present (exc)) then
 if (present (level)) then
 local_level = level
 else
 local_level = EXC_DEFAULT
 end if
 if (exc%level < local_level) then
 exc%level = local_level
 if (present (origin)) then
 exc%origin = origin
 else
 exc%origin = "[vamp]"
 end if
 if (present (message)) then
 exc%message = message
 else
 exc%message = "[vamp]"
 end if
 end if
end if
end subroutine raise_exception

```

245a *⟨Implementation of exceptions procedures 244c⟩*+≡

```

elemental subroutine clear_exception (exc)
 type(exception), intent(inout) :: exc
 exc%level = 0
 exc%message = ""
 exc%origin = ""
end subroutine clear_exception

```

245b *⟨Implementation of exceptions procedures 244c⟩*+≡

```

pure subroutine gather_exceptions (exc, excs)
 type(exception), intent(inout) :: exc
 type(exception), dimension(:), intent(in) :: excs
 integer :: i
 i = sum (maxloc (excs%level))
 if (exc%level < excs(i)%level) then
 call raise_exception (exc, excs(i)%level, excs(i)%origin, &
 excs(i)%message)
 end if
end subroutine gather_exceptions

```

Here's how to use `gather_exceptions`. `elemental_procedure`

246 *<Idioms 98b>*+≡

```
call clear_exception (excs)
call elemental_procedure_1 (y, x, excs)
call elemental_procedure_2 (b, a, excs)
if (any (excs%level > 0)) then
 call gather_exceptions (exc, excs)
 return
end if
```

# —C—

## THE ART OF RANDOM NUMBERS

Volume two of Donald E. Knuth' *The Art of Computer Programming* [15] has always been celebrated as the prime reference for random number generation. Recently, the third edition has been published and it contains a gem of a *portable* random number generator. It generates 30-bit integers with the following desirable properties

- they pass all the tests from George Marsaglia's "diehard" suite of tests for random number generators [23] (but see [15] for a caveat regarding the "birthday-spacing" test)
- they can be generated with portable signed 32-bit arithmetic (Fortran can't do unsigned arithmetic)
- it is faster than other lagged Fibonacci generators
- it can create at least  $2^{30} - 2$  independent sequences

### *C.1 Application Program Interface*

A function returning single reals and integers. Note that the static version without the `tao_random_state` argument does not require initialization. It will behave as if `call tao_random_seed(0)` had been executed. On the other hand, the parallelizable version with the explicit `tao_random_state` will fail if none of the `tao_random_create` have been called for the state. (This is a deficiency of Fortran90 that can be fixed in Fortran95).

247 *<API documentation 247>*≡  
    `call tao_random_number (r)`  
    `call tao_random_number (s, r)`



The state of the random number generator comes in two varieties: buffered and raw. The former is much more efficient, but it can be beneficial to flush the buffers and to pass only the raw state in order to save of interprocess communication (IPC) costs.

248a  $\langle$ API documentation 247 $\rangle + \equiv$   
`type(tao_random_state) :: s`  
`type(tao_random_raw_state) :: rs`

Subroutines filling arrays of reals and integers:

248b  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_number (a, num = n)`  
`call tao_random_number (s, a, num = n)`

Subroutine for changing the seed:

248c  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_seed (seed = seed)`  
`call tao_random_seed (s, seed = seed)`

Subroutine for changing the luxury. Per default, use all random numbers:

248d  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_luxury ()`  
`call tao_random_luxury (s)`

With an integer argument, use the first n of each fill of the buffer:

248e  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_luxury (n)`  
`call tao_random_luxury (s, n)`

With a floating point argument, use that fraction of each fill of the buffer:

248f  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_luxury (x)`  
`call tao_random_luxury (s, x)`

Create a `tao_random_state`

248g  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_create (s, seed, buffer_size = buffer_size)`  
`call tao_random_create (s, raw_state, buffer_size = buffer_size)`  
`call tao_random_create (s, state)`

Create a `tao_random_raw_state`

248h  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_create (rs, seed)`  
`call tao_random_create (rs, raw_state)`  
`call tao_random_create (rs, state)`

Destroy a `tao_random_state` or `tao_random_raw_state`

248i  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_destroy (s)`

Copy `tao_random_state` and `tao_random_raw_state` in all four combinations

249a  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_copy (lhs, rhs)`  
`lhs = rhs`

249b  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_flush (s)`

249c  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_read (s, unit)`  
`call tao_random_write (s, unit)`

249d  $\langle$ API documentation 247 $\rangle + \equiv$   
`call tao_random_test (name = name)`

Here is a sample application of random number states:

249e  $\langle$ API documentation 247 $\rangle + \equiv$   
`subroutine threads (args, y, state)`  
`real, dimension(:), intent(in) :: args`  
`real, dimension(:), intent(out) :: y`  
`type(tao_random_state) :: state`  
`integer :: seed`  
`type(tao_random_raw_state), dimension(size(y)) :: states`  
`integer :: s`  
`call tao_random_number (state, seed)`  
`call tao_random_create (states, (/ (s, s=seed,size(y)-1) /))`  
`y = thread (args, states)`  
`end function thread`

In this example, we could equivalently pass an integer seed, instead of `raw_state`. But in more complicated cases it can be beneficial to have the option of reusing `raw_state` in the calling routine.

249f  $\langle$ API documentation 247 $\rangle + \equiv$   
`elemental function thread (arg, raw_state) result (y)`  
`real, dimension, intent(in) :: arg`  
`type(tao_random_raw_state) :: raw_state`  
`real :: y`  
`type(tao_random_state) :: state`  
`real :: r`  
`call tao_random_create (state, raw_state)`  
`do`  
`...`  
`call tao_random_number (state, r)`  
`...`  
`end do`

```
end function thread
```

## C.2 Low Level Routines

Here the low level routines are *much* more interesting than the high level routines. The latter contain a lot of duplication (made necessary by Fortran’s lack of parametric polymorphism) and consist mostly of bookkeeping. We wil therefore start with the former.

### C.2.1 Generation of 30-bit Random Numbers

The generator is a subtractive lagged Fibonacci

$$X_j = (X_{j-K} - X_{j-L}) \mod 2^{30} \quad (\text{C.1})$$

with lags  $K = 100$  and  $L = 37$ .

250a `<Parameters in tao_random_numbers 250a>≡`  
`integer, parameter, private :: K = 100, L = 37`

Other good choices for  $K$  and  $L$  are (cf. [15], table 1 in section 3.2.2, p. 29)

250b `<Parameters in tao_random_numbers (alternatives) 250b>≡`  
`integer, parameter, private :: K = 55, L = 24`  
`integer, parameter, private :: K = 89, L = 38`  
`integer, parameter, private :: K = 100, L = 37`  
`integer, parameter, private :: K = 127, L = 30`  
`integer, parameter, private :: K = 258, L = 83`  
`integer, parameter, private :: K = 378, L = 107`  
`integer, parameter, private :: K = 607, L = 273`

A modulus of  $2^{30}$  is the largest we can handle in *portable* (i.e. *signed*) 32-bit arithmetic

250c `<Variables in 30-bit tao_random_numbers 250c>≡`  
`integer(kind=tao_i32), parameter, private :: M = 2**30`

`generate` fills the array  $a_1, \dots, a_n$  with random integers  $0 \leq a_i < 2^{30}$ . We *must* have at least  $n \geq K$ . Higher values don’t change the results, but make `generate` more efficient (about a factor of two, asymptotically). For  $K = 100$ , DEK recommends  $n \geq 1000$ . Best results are obtained using the first 100 random numbers out of 1009. Let’s therefore use 1009 as a default buffer size. The user can call `tao_random_luxury (100)` him/herself:

250d `<Parameters in tao_random_numbers 250a>+≡`  
`integer, parameter, private :: DEFAULT_BUFFER_SIZE = 1009`

Since users are not expected to call `generate` directly, we do *not* check for  $n \geq K$  and assume that the caller knows what (s)he's doing ...

**251a** *⟨Implementation of 30-bit tao\_random\_numbers 251a⟩*≡  

```

pure subroutine generate (a, state)
 integer(kind=tao_i32), dimension(:), intent(inout) :: a, state
 integer :: j, n
 n = size (a)
 ⟨Load a and refresh state 251c⟩
end subroutine generate

```

**251b** *⟨Declaration of tao\_random\_numbers 251b⟩*≡  

```

private :: generate
state(1:K) is already set up properly:

```

**251c** *⟨Load a and refresh state 251c⟩*≡  

```

a(1:K) = state(1:K)

```

The remaining  $n - K$  random numbers can be gotten directly from the recursion (C.1). Note that Fortran90's new `modulo` intrinsic does the right thing, since it guarantees (unlike Fortran77's `mod`) that  $0 \leq \text{modulo}(a, m) < a$  if  $m > 0$ :

**251d** *⟨Load a and refresh state 251c⟩*+≡  

```

do j = K+1, n
 a(j) = modulo (a(j-K) - a(j-L), M)
end do

```

Do the recursion (C.1)  $K$  more times to prepare `state(1:K)` for the next invocation of `generate`.

**251e** *⟨Load a and refresh state 251c⟩*+≡  

```

state(1:L) = modulo (a(n+1-K:n+L-K) - a(n+1-L:n), M)
do j = L+1, K
 state(j) = modulo (a(n+j-K) - state(j-L), M)
end do

```

### C.2.2 Initialization of 30-bit Random Numbers

The non-trivial and most beautiful part is the algorithm to initialize the random number generator state `state` with the first  $K$  numbers. I haven't studied algebra over finite fields in sufficient depth to consider the mathematics behind it straightforward. The commentary below is rather verbose and reflects my understanding of DEK's rather terse remarks (solution to exercise 3.6-9 [15]).

**251f** *⟨Implementation of tao\_random\_numbers 251f⟩*≡  

```

subroutine seed_static (seed)

```

```

integer, optional, intent(in) :: seed
call seed_stateless (s_state, seed)
s_virginal = .false.
s_last = size (s_buffer)
end subroutine seed_static

```

The static version of tao\_random\_raw\_state:

**252a** *⟨Variables in 30-bit tao\_random\_numbers 250c⟩*+≡  
integer(kind=tao\_i32), dimension(K), save, private :: s\_state  
logical, save, private :: s\_virginal = .true.

**252b** *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
elemental subroutine seed\_raw\_state (s, seed)  
type(tao\_random\_raw\_state), intent(inout) :: s  
integer, optional, intent(in) :: seed  
call seed\_stateless (s%x, seed)  
end subroutine seed\_raw\_state

**252c** *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
elemental subroutine seed\_state (s, seed)  
type(tao\_random\_state), intent(inout) :: s  
integer, optional, intent(in) :: seed  
call seed\_raw\_state (s%state, seed)  
s%last = size (s%buffer)  
end subroutine seed\_state

This incarnation of the procedure is pure.

**252d** *⟨Implementation of 30-bit tao\_random\_numbers 251a⟩*+≡  
pure subroutine seed\_stateless (state, seed)  
integer(kind=tao\_i32), dimension(:), intent(out) :: state  
integer, optional, intent(in) :: seed  
*⟨Parameters local to tao\_random\_seed 253a⟩*  
integer :: seed\_value, j, s, t  
integer(kind=tao\_i32), dimension(2\*K-1) :: x  
*⟨Set up seed\_value from seed or DEFAULT\_SEED 253c⟩*  
*⟨Bootstrap the x buffer 253d⟩*  
*⟨Set up s and t 253f⟩*  
do  
*⟨ $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) 254a⟩*  
*⟨ $p(z) \rightarrow zp(z)$  (modulo 2 and  $z^K + z^L + 1$ ) 255a⟩*  
*⟨Shift s or t and exit if  $t \leq 0$  255b⟩*  
end do  
*⟨Fill state from x 255c⟩*  
end subroutine seed\_stateless

Any default will do

253a  $\langle$ Parameters local to tao\_random\_seed 253a $\rangle \equiv$   
integer, parameter :: DEFAULT\_SEED = 0

These must not be changed:

253b  $\langle$ Parameters local to tao\_random\_seed 253a $\rangle + \equiv$   
integer, parameter :: MAX\_SEED = 2\*\*30 - 3  
integer, parameter :: TT = 70

253c  $\langle$ Set up seed\_value from seed or DEFAULT\_SEED 253c $\rangle \equiv$   
if (present (seed)) then  
seed\_value = seed  
else  
seed\_value = DEFAULT\_SEED  
end if  
if (seed\_value < 0 .or. seed\_value > MAX\_SEED) then  
!!! print \*, "tao\_random\_seed: seed (", seed\_value, &  
!!! " ) not in [ 0,", MAX\_SEED, "]"!  
seed\_value = modulo (abs (seed\_value), MAX\_SEED + 1)  
!!! print \*, "tao\_random\_seed: seed set to ", seed\_value, "!"  
end if

Fill the array  $x_1, \dots, x_K$  with even integers, shifted cyclically by 29 bits.

253d  $\langle$ Bootstrap the x buffer 253d $\rangle \equiv$   
s = seed\_value - modulo (seed\_value, 2) + 2  
do j = 1, K  
x(j) = s  
s = 2\*s  
if (s >= M) then  
s = s - M + 2  
end if  
end do  
x(K+1:2\*K-1) = 0

Make  $x_2$  (and only  $x_2$ ) odd:

253e  $\langle$ Bootstrap the x buffer 253d $\rangle + \equiv$   
x(2) = x(2) + 1

253f  $\langle$ Set up s and t 253f $\rangle \equiv$   
s = seed\_value  
t = TT - 1

Consider the polynomial

$$p(z) = \sum_{n=1}^K x_n z^{n-1} = x_K z^{K-1} + \dots + x_2 z + x_1 \quad (\text{C.2})$$

We have  $p(z)^2 = p(z^2) \pmod 2$  because cross terms have an even coefficient and  $x_n^2 = x_n \pmod 2$ . Therefore we can square the polynomial by shifting the coefficients. The coefficients for  $n > K$  will be reduced  $\pmod 2$  below.

$$\text{254a} \quad \langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ 254a)} \equiv \\ \mathbf{x}(3:2*K-1:2) = \mathbf{x}(2:K)$$

The coefficients of the odd powers (those with the even indices) have not been changed yet. Set them to a flipped version of the other coefficients with the least significant bit set to 0.

$$\begin{aligned} x_2 &\leftarrow \text{even } x_{2K-1} \\ x_4 &\leftarrow \text{even } x_{2K-3} \\ &\dots \\ x_{K+L-1} &\leftarrow \text{even } x_{K-L+2} \end{aligned} \tag{C.3}$$

Note that the array notation is unambiguous because  $2K - 1$  is odd and source and destination are therefore interleaved:

$$\text{254b} \quad \langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ 254a)} + \equiv \\ \mathbf{x}(2:K+L-1:2) = \mathbf{x}(2*K-1:K-L+2:-2) - \text{modulo } (\mathbf{x}(2*K-1:K-L+2:-2), 2)$$

The Fortran program in [15] reads

$$\text{254c} \quad \langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ (DEK, Fortran) 254c)} \equiv \\ \text{do } j = 2*K-1, K-L+1, -2 \\ \quad \mathbf{x}(2*K+1-j) = \mathbf{x}(j) - \text{mod } (\mathbf{x}(j), 2) \\ \text{end do}$$

i.e.

$$\text{254d} \quad \langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ (alternative) 254d)} \equiv \\ \mathbf{x}(2:K+L:2) = \mathbf{x}(2*K-1:K-L+1:-2) - \text{modulo } (\mathbf{x}(2*K-1:K-L+1:-2), 2)$$

which is equivalent, as long as  $K + L$  is odd. This is the case most of the time. The version used above is the direct translation of the C version

$$\text{254e} \quad \langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ (DEK, C) 254e)} \equiv \\ \text{for } (j = 2*K-2; j > K-L; j -= 2) \\ \quad \mathbf{x}[2*K-1-j] = \text{evenize } (\mathbf{x}[j]);$$



I should decide from theoretical considerations whether the difference matters and if so, which is the correct version. At least I should inform DEK of the inconsistency.

Let's return to the coefficients for  $n > K$  generated by the shifting above. Subtract  $z^n(z^K + z^L + 1) = z^n z^K (1 + z^{-(K-L)} + z^{-K})$  iff the coefficient of  $z^n z^K$  doesn't vanish  $\pmod 2$  after squaring. The coefficient of  $z^n z^K$  is left alone, because it doesn't belong to  $p(z)$  anyway.

```

254f $\langle p(z) \rightarrow p(z)^2 \text{ (modulo 2 and } z^K + z^L + 1) \text{ 254a} \rangle + \equiv$
 do j= 2*K-1, K+1, -1
 if (modulo (x(j), 2) == 1) then
 x(j-(K-L)) = modulo (x(j-(K-L)) - x(j), M)
 x(j-K) = modulo (x(j-K) - x(j), M)
 end if
 end do

255a $\langle p(z) \rightarrow zp(z) \text{ (modulo 2 and } z^K + z^L + 1) \text{ 255a} \rangle \equiv$
 if (modulo (s, 2) == 1) then
 x(2:K+1) = x(1:K)
 x(1) = x(K+1)
 if (modulo (x(K+1), 2) == 1) then
 x(L+1) = modulo (x(L+1) - x(K+1), M)
 end if
 end if

255b $\langle \text{Shift s or t and exit if } t \leq 0 \text{ 255b} \rangle \equiv$
 if (s /= 0) then
 s = s / 2
 else
 t = t - 1
 end if
 if (t <= 0) then
 exit
 end if

255c $\langle \text{Fill state from x 255c} \rangle \equiv$
 state(K-L+1:K) = x(1:L)
 state(1:K-L) = x(L+1:K)

255d $\langle \text{Interfaces of tao_random_numbers 255d} \rangle \equiv$
 interface tao_random_seed
 module procedure $\langle \text{Specific procedures for tao_random_seed 255f} \rangle$
 end interface

255e $\langle \text{Declaration of tao_random_numbers 251b} \rangle + \equiv$
 private :: $\langle \text{Specific procedures for tao_random_seed 255f} \rangle$

255f $\langle \text{Specific procedures for tao_random_seed 255f} \rangle \equiv$
 seed_static, seed_state, seed_raw_state

```

### C.2.3 Generation of 52-bit Random Numbers

$$X_j = (X_{j-K} + X_{j-L}) \mod 1 \quad (\text{C.4})$$

```

255g $\langle \text{Variables in 52-bit tao_random_numbers 255g} \rangle \equiv$
 real(kind=tao_r64), parameter, private :: M = 1.0_tao_r64

```



The state of the internal routines

```

256a <Variables in 52-bit tao_random_numbers 255g>+≡
 real(kind=tao_r64), dimension(K), save, private :: s_state
 logical, save, private :: s_virginal = .true.

256b <Implementation of 52-bit tao_random_numbers 256b>≡
 pure subroutine generate (a, state)
 real(kind=tao_r64), dimension(:), intent(inout) :: a
 real(kind=tao_r64), dimension(:), intent(inout) :: state
 integer :: j, n
 n = size (a)
 <Load 52-bit a and refresh state 256c>
 end subroutine generate

```

That's almost identical to the 30-bit version, except that the relative sign is flipped:

```

256c <Load 52-bit a and refresh state 256c>≡
 a(1:K) = state(1:K)
 do j = K+1, n
 a(j) = modulo (a(j-K) + a(j-L), M)
 end do
 state(1:L) = modulo (a(n+1-K:n+L-K) + a(n+1-L:n), M)
 do j = L+1, K
 state(j) = modulo (a(n+j-K) + state(j-L), M)
 end do

```

### C.2.4 Initialization of 52-bit Random Numbers

This incarnation of the procedure is pure.

```

256d <Implementation of 52-bit tao_random_numbers 256b>+≡
 pure subroutine seed_stateless (state, seed)
 real(kind=tao_r64), dimension(:), intent(out) :: state
 integer, optional, intent(in) :: seed
 <Parameters local to tao_random_seed 253a>
 <Variables local to 52-bit tao_random_seed 257b>
 <Set up seed_value from seed or DEFAULT_SEED 253c>
 <Bootstrap the x and x1 buffers 257d>
 <Set up s and t 253f>
 do
 <52-bit $p(z) \rightarrow p(z)^2$ (modulo 2 and $z^K + z^L + 1$) 257g>
 <52-bit $p(z) \rightarrow zp(z)$ (modulo 2 and $z^K + z^L + 1$) 258a>
 <Shift s or t and exit if t ≤ 0 255b>
 end do
 <Fill state from x 255c>

```

```

 end subroutine seed_stateless

257a \langle Declaration of tao_random_numbers 251b $\rangle + \equiv$
 private :: seed_stateless

257b \langle Variables local to 52-bit tao_random_seed 257b $\rangle \equiv$
 real(kind=tao_r64), parameter :: ULP = 2.0_tao_r64**(-52)

257c \langle Variables local to 52-bit tao_random_seed 257b $\rangle + \equiv$
 real(kind=tao_r64), dimension(2*K-1) :: x, x1
 real(kind=tao_r64) :: ss
 integer :: seed_value, t, s, j

257d \langle Bootstrap the x and x1 buffers 257d $\rangle \equiv$
 x1 = 0
 x1(2) = ULP

257e \langle Bootstrap the x and x1 buffers 257d $\rangle + \equiv$
 ss = 2*ULP * (seed_value + 2)
 do j = 1, K
 x(j) = ss
 ss = 2*ss
 if (ss >= 1) then
 ss = ss - 1 + 2*ULP
 end if
 end do
 x(K+1:2*K-1) = 0.0

257f \langle Bootstrap the x and x1 buffers 257d $\rangle + \equiv$
 x(2) = x(2) + ULP

257g \langle 52-bit $p(z) \rightarrow p(z)^2$ (modulo 2 and $z^K + z^L + 1$) 257g $\rangle \equiv$
 x1(3:2*K-1:2) = x1(2:K)
 x(3:2*K-1:2) = x(2:K)

```

This works because  $2*K-1$  is odd (the same problem as on page 254 arises here as well)

```

257h \langle 52-bit $p(z) \rightarrow p(z)^2$ (modulo 2 and $z^K + z^L + 1$) 257g $\rangle + \equiv$
 x1(2:K+L-1:2) = 0.0
 x(2:K+L-1:2) = x(2*K-1:K-L+2:-2) - x1(2*K-1:K-L+2:-2)
 do j = 2*K-1, K+1, -1
 if (x1(j) /= 0) then
 x1(j-(K-L)) = ULP - x1(j-(K-L))
 x(j-(K-L)) = modulo (x(j-(K-L)) + x(j), M)
 x1(j-K) = ULP - x1(j-K)
 x(j-K) = modulo (x(j-K) + x(j), M)
 end if
 end do

```

258a  $\langle 52\text{-bit } p(z) \rightarrow zp(z) \text{ (modulo 2 and } z^K + z^L + 1) \text{ 258a} \rangle \equiv$

```

 if (modulo (s, 2) == 1) then
 x1(2:K+1) = x1(1:K)
 x1(1) = x1(K+1)
 x(2:K+1) = x(1:K)
 x(1) = x(K+1)
 if (x1(K+1) /= 0) then
 x1(L+1) = ULP - x1(L+1)
 x(L+1) = modulo (x(L+1) + x(K+1), M)
 end if
 end if
end if

```

### C.3 The State

258b  $\langle \text{Declaration of 30-bit tao\_random\_numbers types 258b} \rangle \equiv$

```

type, public :: tao_random_raw_state
private
 integer(kind=tao_i32), dimension(K) :: x
end type tao_random_raw_state

```

258c  $\langle \text{Declaration of 30-bit tao\_random\_numbers types 258b} \rangle + \equiv$

```

type, public :: tao_random_state
private
 type(tao_random_raw_state) :: state
 integer(kind=tao_i32), dimension(:), pointer :: buffer => null ()
 integer :: buffer_end, last
end type tao_random_state

```

258d  $\langle \text{Declaration of 52-bit tao\_random\_numbers types 258d} \rangle \equiv$

```

type, public :: tao_random_raw_state
private
 real(kind=tao_r64), dimension(K) :: x
end type tao_random_raw_state

```

258e  $\langle \text{Declaration of 52-bit tao\_random\_numbers types 258d} \rangle + \equiv$

```

type, public :: tao_random_state
private
 type(tao_random_raw_state) :: state
 real(kind=tao_r64), dimension(:), pointer :: buffer => null ()
 integer :: buffer_end, last
end type tao_random_state

```

### C.3.1 Creation

259a *<Interfaces of tao\_random\_numbers 255d>+≡*  

```

interface tao_random_create
 module procedure <Specific procedures for tao_random_create 259c>
end interface

```

259b *<Declaration of tao\_random\_numbers 251b>+≡*  

```

private :: <Specific procedures for tao_random_create 259c>

```

259c *<Specific procedures for tao\_random\_create 259c>≡*  

```

create_state_from_seed, create_raw_state_from_seed, &
create_state_from_state, create_raw_state_from_state, &
create_state_from_raw_state, create_raw_state_from_raw_st

```

There are no procedures for copying the state of the static generator to or from an explicit `tao_random_state`. Users needing this functionality can be expected to handle explicit states anyway. Since the direction of the copying can not be obvious from the type of the argument, such functions would spoil the simplicity of the generic procedure interface.

259d *<Implementation of tao\_random\_numbers 251f>+≡*  

```

elemental subroutine create_state_from_seed (s, seed, buffer_size)
 type(tao_random_state), intent(out) :: s
 integer, intent(in) :: seed
 integer, intent(in), optional :: buffer_size
 call create_raw_state_from_seed (s%state, seed)
 if (present (buffer_size)) then
 s%buffer_end = max (buffer_size, K)
 else
 s%buffer_end = DEFAULT_BUFFER_SIZE
 end if
 allocate (s%buffer(s%buffer_end))
 call tao_random_flush (s)
end subroutine create_state_from_seed

```

259e *<Implementation of tao\_random\_numbers 251f>+≡*  

```

elemental subroutine create_state_from_state (s, state)
 type(tao_random_state), intent(out) :: s
 type(tao_random_state), intent(in) :: state
 call create_raw_state_from_raw_st (s%state, state%state)
 allocate (s%buffer(size(state%buffer)))
 call tao_random_copy (s, state)
end subroutine create_state_from_state

```

259f *<Implementation of tao\_random\_numbers 251f>+≡*  

```

elemental subroutine create_state_from_raw_state &

```

```

 (s, raw_state, buffer_size)
 type(tao_random_state), intent(out) :: s
 type(tao_random_raw_state), intent(in) :: raw_state
 integer, intent(in), optional :: buffer_size
 call create_raw_state_from_raw_st (s%state, raw_state)
 if (present (buffer_size)) then
 s%buffer_end = max (buffer_size, K)
 else
 s%buffer_end = DEFAULT_BUFFER_SIZE
 end if
 allocate (s%buffer(s%buffer_end))
 call tao_random_flush (s)
end subroutine create_state_from_raw_state

```

260a *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine create\_raw\_state\_from\_seed (s, seed)  
   type(tao\_random\_raw\_state), intent(out) :: s  
   integer, intent(in) :: seed  
   call seed\_raw\_state (s, seed)  
end subroutine create\_raw\_state\_from\_seed

260b *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine create\_raw\_state\_from\_state (s, state)  
   type(tao\_random\_raw\_state), intent(out) :: s  
   type(tao\_random\_state), intent(in) :: state  
   call copy\_state\_to\_raw\_state (s, state)  
end subroutine create\_raw\_state\_from\_state

260c *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine create\_raw\_state\_from\_raw\_st (s, raw\_state)  
   type(tao\_random\_raw\_state), intent(out) :: s  
   type(tao\_random\_raw\_state), intent(in) :: raw\_state  
   call copy\_raw\_state (s, raw\_state)  
end subroutine create\_raw\_state\_from\_raw\_st

### C.3.2 Destruction

260d *⟨Interfaces of tao\_random\_numbers 255d⟩*+≡  
 interface tao\_random\_destroy  
   module procedure destroy\_state, destroy\_raw\_state  
end interface

260e *⟨Declaration of tao\_random\_numbers 251b⟩*+≡  
 private :: destroy\_state, destroy\_raw\_state

261a *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine destroy\_state (s)  
   type(tao\_random\_state), intent(inout) :: s  
   deallocate (s%buffer)  
 end subroutine destroy\_state

Currently, this is a no-op, but we might need a non-trivial destruction method in the future

261b *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine destroy\_raw\_state (s)  
   type(tao\_random\_raw\_state), intent(inout) :: s  
 end subroutine destroy\_raw\_state

### C.3.3 Copying

261c *⟨Interfaces of tao\_random\_numbers 255d⟩*+≡  
 interface tao\_random\_copy  
   module procedure *⟨Specific procedures for tao\_random\_copy 261f⟩*  
 end interface

261d *⟨Interfaces of tao\_random\_numbers 255d⟩*+≡  
 interface assignment(=)  
   module procedure *⟨Specific procedures for tao\_random\_copy 261f⟩*  
 end interface

261e *⟨Declaration of tao\_random\_numbers 251b⟩*+≡  
 public :: assignment(=)  
 private :: *⟨Specific procedures for tao\_random\_copy 261f⟩*

261f *⟨Specific procedures for tao\_random\_copy 261f⟩*≡  
 copy\_state, copy\_raw\_state, &  
 copy\_raw\_state\_to\_state, copy\_state\_to\_raw\_state

261g *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine copy\_state (lhs, rhs)  
   type(tao\_random\_state), intent(inout) :: lhs  
   type(tao\_random\_state), intent(in) :: rhs  
   call copy\_raw\_state (lhs%state, rhs%state)  
   if (size (lhs%buffer) /= size (rhs%buffer)) then  
     deallocate (lhs%buffer)  
     allocate (lhs%buffer(size(rhs%buffer)))  
   end if  
   lhs%buffer = rhs%buffer  
   lhs%buffer\_end = rhs%buffer\_end  
   lhs%last = rhs%last  
 end subroutine copy\_state

262a *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine copy\_raw\_state (lhs, rhs)  
   type(tao\_random\_raw\_state), intent(out) :: lhs  
   type(tao\_random\_raw\_state), intent(in) :: rhs  
   lhs%x = rhs%x  
end subroutine copy\_raw\_state

262b *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine copy\_raw\_state\_to\_state (lhs, rhs)  
   type(tao\_random\_state), intent(inout) :: lhs  
   type(tao\_random\_raw\_state), intent(in) :: rhs  
   call copy\_raw\_state (lhs%state, rhs)  
   call tao\_random\_flush (lhs)  
end subroutine copy\_raw\_state\_to\_state

262c *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine copy\_state\_to\_raw\_state (lhs, rhs)  
   type(tao\_random\_raw\_state), intent(out) :: lhs  
   type(tao\_random\_state), intent(in) :: rhs  
   call copy\_raw\_state (lhs, rhs%state)  
end subroutine copy\_state\_to\_raw\_state

### C.3.4 Flushing

262d *⟨Implementation of tao\_random\_numbers 251f⟩*+≡  
 elemental subroutine tao\_random\_flush (s)  
   type(tao\_random\_state), intent(inout) :: s  
   s%last = size (s%buffer)  
end subroutine tao\_random\_flush

### C.3.5 Input and Output

262e *⟨Interfaces of tao\_random\_numbers 255d⟩*+≡  
 interface tao\_random\_write  
   module procedure &  
     write\_state\_unit, write\_state\_name, &  
     write\_raw\_state\_unit, write\_raw\_state\_name  
 end interface

262f *⟨Declaration of tao\_random\_numbers 251b⟩*+≡  
 private :: write\_state\_unit, write\_state\_name  
 private :: write\_raw\_state\_unit, write\_raw\_state\_name

```

263a <Interfaces of tao_random_numbers 255d>+≡
 interface tao_random_read
 module procedure &
 read_state_unit, read_state_name, &
 read_raw_state_unit, read_raw_state_name
 end interface

263b <Declaration of tao_random_numbers 251b>+≡
 private :: read_state_unit, read_state_name
 private :: read_raw_state_unit, read_raw_state_name

263c <Implementation of tao_random_numbers 251f>+≡
 subroutine write_state_unit (s, unit)
 type(tao_random_state), intent(in) :: s
 integer, intent(in) :: unit
 write (unit = unit, fmt = *) "BEGIN TAO_RANDOM_STATE"
 call write_raw_state_unit (s%state, unit)
 write (unit = unit, fmt = "(2(1x,a16,1x,i10/),1x,a16,1x,i10)") &
 "BUFFER_SIZE", size (s%buffer), &
 "BUFFER_END", s%buffer_end, &
 "LAST", s%last
 write (unit = unit, fmt = *) "BEGIN BUFFER"
 call write_state_array (s%buffer, unit)
 write (unit = unit, fmt = *) "END BUFFER"
 write (unit = unit, fmt = *) "END TAO_RANDOM_STATE"
 end subroutine write_state_unit

263d <Implementation of tao_random_numbers 251f>+≡
 subroutine read_state_unit (s, unit)
 type(tao_random_state), intent(inout) :: s
 integer, intent(in) :: unit
 integer :: buffer_size
 read (unit = unit, fmt = *)
 call read_raw_state_unit (s%state, unit)
 read (unit = unit, fmt = "(2(1x,16x,1x,i10/),1x,16x,1x,i10)") &
 buffer_size, s%buffer_end, s%last
 read (unit = unit, fmt = *)
 if (buffer_size /= size (s%buffer)) then
 deallocate (s%buffer)
 allocate (s%buffer(buffer_size))
 end if
 call read_state_array (s%buffer, unit)
 read (unit = unit, fmt = *)
 read (unit = unit, fmt = *)
 end subroutine read_state_unit

```



264a *<Implementation of tao\_random\_numbers 251f>+≡*  
 subroutine write\_raw\_state\_unit (s, unit)  
   type(tao\_random\_raw\_state), intent(in) :: s  
   integer, intent(in) :: unit  
   write (unit = unit, fmt = \*) "BEGIN TAO\_RANDOM\_RAW\_STATE"  
   call write\_state\_array (s%x, unit)  
   write (unit = unit, fmt = \*) "END TAO\_RANDOM\_RAW\_STATE"  
end subroutine write\_raw\_state\_unit

264b *<Implementation of tao\_random\_numbers 251f>+≡*  
 subroutine read\_raw\_state\_unit (s, unit)  
   type(tao\_random\_raw\_state), intent(inout) :: s  
   integer, intent(in) :: unit  
   read (unit = unit, fmt = \*)  
   call read\_state\_array (s%x, unit)  
   read (unit = unit, fmt = \*)  
end subroutine read\_raw\_state\_unit

264c *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*  
 subroutine write\_state\_array (a, unit)  
   integer(kind=tao\_i32), dimension(:), intent(in) :: a  
   integer, intent(in) :: unit  
   integer :: i  
   do i = 1, size (a)  
   write (unit = unit, fmt = "(1x,i10,1x,i10)") i, a(i)  
   end do  
end subroutine write\_state\_array

264d *<Declaration of 30-bit tao\_random\_numbers 264d>≡*  
 private :: write\_state\_array

264e *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*  
 subroutine read\_state\_array (a, unit)  
   integer(kind=tao\_i32), dimension(:), intent(inout) :: a  
   integer, intent(in) :: unit  
   integer :: i, idum  
   do i = 1, size (a)  
   read (unit = unit, fmt = \*) idum, a(i)  
   end do  
end subroutine read\_state\_array

264f *<Declaration of 30-bit tao\_random\_numbers 264d>+≡*  
 private :: read\_state\_array

Reading and writing 52-bit floating point numbers accurately is beyond most Fortran runtime libraries. Their job is simplified considerably if we rescale

by  $2^{52}$  before writing. Then the temptation to truncate will not be as overwhelming as before ...

```

265a <Implementation of 52-bit tao_random_numbers 256b>+≡
 subroutine write_state_array (a, unit)
 real(kind=tao_r64), dimension(:), intent(in) :: a
 integer, intent(in) :: unit
 integer :: i
 do i = 1, size (a)
 write (unit = unit, fmt = "(1x,i10,1x,f30.0)") i, 2.0_tao_r64**52 * a(i)
 end do
 end subroutine write_state_array

265b <Declaration of 52-bit tao_random_numbers 265b>≡
 private :: write_state_array

265c <Implementation of 52-bit tao_random_numbers 256b>+≡
 subroutine read_state_array (a, unit)
 real(kind=tao_r64), dimension(:), intent(inout) :: a
 integer, intent(in) :: unit
 real(kind=tao_r64) :: x
 integer :: i, idum
 do i = 1, size (a)
 read (unit = unit, fmt = *) idum, x
 a(i) = 2.0_tao_r64**(-52) * x
 end do
 end subroutine read_state_array

265d <Declaration of 52-bit tao_random_numbers 265b>+≡
 private :: read_state_array

265e <Implementation of tao_random_numbers 251f>+≡
 subroutine find_free_unit (u, iostat)
 integer, intent(out) :: u
 integer, intent(out), optional :: iostat
 logical :: exists, is_open
 integer :: i, status
 do i = MIN_UNIT, MAX_UNIT
 inquire (unit = i, exist = exists, opened = is_open, &
 iostat = iostat)
 if (status == 0) then
 if (exists .and. .not. is_open) then
 u = i
 if (present (iostat)) then
 iostat = 0
 end if
 end if
 end if
 end do
 end subroutine find_free_unit

```

```

 return
 end if
end if
end do
if (present (iostat)) then
 iostat = -1
end if
u = -1
end subroutine find_free_unit

```

266a *<Variables in tao\_random\_numbers 266a>*≡  
integer, parameter, private :: MIN\_UNIT = 11, MAX\_UNIT = 99

266b *<Declaration of tao\_random\_numbers 251b>*+≡  
private :: find\_free\_unit

266c *<Implementation of tao\_random\_numbers 251f>*+≡  
subroutine write\_state\_name (s, name)  
type(tao\_random\_state), intent(in) :: s  
character(len=\*), intent(in) :: name  
integer :: unit  
call find\_free\_unit (unit)  
open (unit = unit, action = "write", status = "replace", file = name)  
call write\_state\_unit (s, unit)  
close (unit = unit)  
end subroutine write\_state\_name

266d *<Implementation of tao\_random\_numbers 251f>*+≡  
subroutine write\_raw\_state\_name (s, name)  
type(tao\_random\_raw\_state), intent(in) :: s  
character(len=\*), intent(in) :: name  
integer :: unit  
call find\_free\_unit (unit)  
open (unit = unit, action = "write", status = "replace", file = name)  
call write\_raw\_state\_unit (s, unit)  
close (unit = unit)  
end subroutine write\_raw\_state\_name

266e *<Implementation of tao\_random\_numbers 251f>*+≡  
subroutine read\_state\_name (s, name)  
type(tao\_random\_state), intent(inout) :: s  
character(len=\*), intent(in) :: name  
integer :: unit  
call find\_free\_unit (unit)  
open (unit = unit, action = "read", status = "old", file = name)

```

 call read_state_unit (s, unit)
 close (unit = unit)
end subroutine read_state_name

```

267a *<Implementation of tao\_random\_numbers 251f>+≡*

```

subroutine read_raw_state_name (s, name)
 type(tao_random_raw_state), intent(inout) :: s
 character(len=*), intent(in) :: name
 integer :: unit
 call find_free_unit (unit)
 open (unit = unit, action = "read", status = "old", file = name)
 call read_raw_state_unit (s, unit)
 close (unit = unit)
end subroutine read_raw_state_name

```

### C.3.6 Marshaling and Unmarshaling

Note that we can not use the `transfer` intrinsic function for marshalling types that contain pointers that substitute for allocatable array components. `transfer` will copy the pointers in this case and not where they point to!

267b *<Interfaces of tao\_random\_numbers 255d>+≡*

```

interface tao_random_marshall_size
 module procedure marshal_state_size, marshal_raw_state_size
end interface
interface tao_random_marshall
 module procedure marshal_state, marshal_raw_state
end interface
interface tao_random_unmarshal
 module procedure unmarshal_state, unmarshal_raw_state
end interface

```

267c *<Declaration of tao\_random\_numbers 251b>+≡*

```

public :: tao_random_marshall
private :: marshal_state, marshal_raw_state
public :: tao_random_marshall_size
private :: marshal_state_size, marshal_raw_state_size
public :: tao_random_unmarshal
private :: unmarshal_state, unmarshal_raw_state

```

267d *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*

```

pure subroutine marshal_state (s, ibuf, dbuf)
 type(tao_random_state), intent(in) :: s
 integer, dimension(:), intent(inout) :: ibuf
 real(kind=tao_r64), dimension(:), intent(inout) :: dbuf

```

```

integer :: buf_size
buf_size = size (s%buffer)
ibuf(1) = s%buffer_end
ibuf(2) = s%last
ibuf(3) = buf_size
ibuf(4:3+buf_size) = s%buffer
call marshal_raw_state (s%state, ibuf(4+buf_size:), dbuf)
end subroutine marshal_state

268a <Implementation of 30-bit tao_random_numbers 251a>+≡
pure subroutine marshal_state_size (s, iwords, dwords)
type(tao_random_state), intent(in) :: s
integer, intent(out) :: iwords, dwords
call marshal_raw_state_size (s%state, iwords, dwords)
iwords = iwords + 3 + size (s%buffer)
end subroutine marshal_state_size

268b <Implementation of 30-bit tao_random_numbers 251a>+≡
pure subroutine unmarshal_state (s, ibuf, dbuf)
type(tao_random_state), intent(inout) :: s
integer, dimension(:), intent(in) :: ibuf
real(kind=tao_r64), dimension(:), intent(in) :: dbuf
integer :: buf_size
s%buffer_end = ibuf(1)
s%last = ibuf(2)
buf_size = ibuf(3)
s%buffer = ibuf(4:3+buf_size)
call unmarshal_raw_state (s%state, ibuf(4+buf_size:), dbuf)
end subroutine unmarshal_state

268c <Implementation of 30-bit tao_random_numbers 251a>+≡
pure subroutine marshal_raw_state (s, ibuf, dbuf)
type(tao_random_raw_state), intent(in) :: s
integer, dimension(:), intent(inout) :: ibuf
real(kind=tao_r64), dimension(:), intent(inout) :: dbuf
ibuf(1) = size (s%x)
ibuf(2:1+size(s%x)) = s%x
end subroutine marshal_raw_state

268d <Implementation of 30-bit tao_random_numbers 251a>+≡
pure subroutine marshal_raw_state_size (s, iwords, dwords)
type(tao_random_raw_state), intent(in) :: s
integer, intent(out) :: iwords, dwords
iwords = 1 + size (s%x)
dwords = 0
end subroutine marshal_raw_state_size

```

```

269a ⟨Implementation of 30-bit tao_random_numbers 251a⟩+≡
 pure subroutine unmarshal_raw_state (s, ibuf, dbuf)
 type(tao_random_raw_state), intent(inout) :: s
 integer, dimension(:), intent(in) :: ibuf
 real(kind=tao_r64), dimension(:), intent(in) :: dbuf
 integer :: buf_size
 buf_size = ibuf(1)
 s%x = ibuf(2:1+buf_size)
 end subroutine unmarshal_raw_state

269b ⟨Implementation of 52-bit tao_random_numbers 256b⟩+≡
 pure subroutine marshal_state (s, ibuf, dbuf)
 type(tao_random_state), intent(in) :: s
 integer, dimension(:), intent(inout) :: ibuf
 real(kind=tao_r64), dimension(:), intent(inout) :: dbuf
 integer :: buf_size
 buf_size = size (s%buffer)
 ibuf(1) = s%buffer_end
 ibuf(2) = s%last
 ibuf(3) = buf_size
 dbuf(1:buf_size) = s%buffer
 call marshal_raw_state (s%state, ibuf(4:), dbuf(buf_size+1:))
 end subroutine marshal_state

269c ⟨Implementation of 52-bit tao_random_numbers 256b⟩+≡
 pure subroutine marshal_state_size (s, iwords, dwords)
 type(tao_random_state), intent(in) :: s
 integer, intent(out) :: iwords, dwords
 call marshal_raw_state_size (s%state, iwords, dwords)
 iwords = iwords + 3
 dwords = dwords + size(s%buffer)
 end subroutine marshal_state_size

269d ⟨Implementation of 52-bit tao_random_numbers 256b⟩+≡
 pure subroutine unmarshal_state (s, ibuf, dbuf)
 type(tao_random_state), intent(inout) :: s
 integer, dimension(:), intent(in) :: ibuf
 real(kind=tao_r64), dimension(:), intent(in) :: dbuf
 integer :: buf_size
 s%buffer_end = ibuf(1)
 s%last = ibuf(2)
 buf_size = ibuf(3)
 s%buffer = dbuf(1:buf_size)
 call unmarshal_raw_state (s%state, ibuf(4:), dbuf(buf_size+1:))
 end subroutine unmarshal_state

```

270a *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*  

```

pure subroutine marshal_raw_state (s, ibuf, dbuf)
 type(tao_random_raw_state), intent(in) :: s
 integer, dimension(:), intent(inout) :: ibuf
 real(kind=tao_r64), dimension(:), intent(inout) :: dbuf
 ibuf(1) = size (s%x)
 dbuf(1:size(s%x)) = s%x
end subroutine marshal_raw_state

```

270b *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*  

```

pure subroutine marshal_raw_state_size (s, iwords, dwords)
 type(tao_random_raw_state), intent(in) :: s
 integer, intent(out) :: iwords, dwords
 iwords = 1
 dwords = size (s%x)
end subroutine marshal_raw_state_size

```

270c *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*  

```

pure subroutine unmarshal_raw_state (s, ibuf, dbuf)
 type(tao_random_raw_state), intent(inout) :: s
 integer, dimension(:), intent(in) :: ibuf
 real(kind=tao_r64), dimension(:), intent(in) :: dbuf
 integer :: buf_size
 buf_size = ibuf(1)
 s%x = dbuf(1:buf_size)
end subroutine unmarshal_raw_state

```

## C.4 High Level Routines

270d *<tao\_random\_numbers.f90 270d>≡*  

```

! tao_random_numbers.f90 --
<Copyleft notice 1>
module tao_random_numbers
 use kinds
 implicit none
 integer, parameter, private :: tao_i32 = selected_int_kind (9)
 integer, parameter, private :: tao_r64 = selected_real_kind (15)
 <Declaration of tao_random_numbers 251b>
 <Declaration of 30-bit tao_random_numbers 264d>
 <Interfaces of tao_random_numbers 255d>
 <Interfaces of 30-bit tao_random_numbers 277f>
 <Parameters in tao_random_numbers 250a>
 <Variables in tao_random_numbers 266a>

```

```

 <Variables in 30-bit tao_random_numbers 250c>
 <Declaration of 30-bit tao_random_numbers types 258b>
 character(len=*), public, parameter :: TAO_RANDOM_NUMBERS_RCS_ID = &
 "$Id: tao_random_numbers.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of tao_random_numbers 251f>
 <Implementation of 30-bit tao_random_numbers 251a>
end module tao_random_numbers

271a <tao52_random_numbers.f90 271a>≡
 ! tao52_random_numbers.f90 --
 <Copyleft notice 1>
module tao52_random_numbers
 use kinds
 implicit none
 integer, parameter, private :: tao_i32 = selected_int_kind (9)
 integer, parameter, private :: tao_r64 = selected_real_kind (15)
 <Declaration of tao_random_numbers 251b>
 <Declaration of 52-bit tao_random_numbers 265b>
 <Interfaces of tao_random_numbers 255d>
 <Interfaces of 52-bit tao_random_numbers 278d>
 <Parameters in tao_random_numbers 250a>
 <Variables in tao_random_numbers 266a>
 <Variables in 52-bit tao_random_numbers 255g>
 <Declaration of 52-bit tao_random_numbers types 258d>
 character(len=*), public, parameter :: TAO52_RANDOM_NUMBERS_RCS_ID = &
 "$Id: tao_random_numbers.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of tao_random_numbers 251f>
 <Implementation of 52-bit tao_random_numbers 256b>
end module tao52_random_numbers

Ten functions are exported

271b <Declaration of tao_random_numbers 251b>+≡
 public :: tao_random_number
 public :: tao_random_seed
 public :: tao_random_create
 public :: tao_random_destroy
 public :: tao_random_copy
 public :: tao_random_read
 public :: tao_random_write
 public :: tao_random_flush
 ! public :: tao_random_luxury
 public :: tao_random_test

```



### C.4.1 Single Random Numbers

A random integer  $r$  with  $0 \leq r < 2^{30} = 1073741824$ :

272a *Implementation of 30-bit tao\_random\_numbers 251a*+≡  
 pure subroutine integer\_stateless &  
     (state, buffer, buffer\_end, last, r)  
     integer(kind=tao\_i32), dimension(:), intent(inout) :: state, buffer  
     integer, intent(in) :: buffer\_end  
     integer, intent(inout) :: last  
     integer, intent(out) :: r  
     integer, parameter :: NORM = 1  
     *Body of tao\_random\_\* 272b*  
 end subroutine integer\_stateless

272b *Body of tao\_random\_\* 272b*≡  
*Step last and reload buffer iff necessary 272d*  
 r = NORM \* buffer(last)

The low level routine `generate` will fill an array  $a_1, \dots, a_n$ , which will be consumed and refilled like an input buffer. We need at least  $n \geq K$  for the call to `generate`.

272c *Variables in 30-bit tao\_random\_numbers 250c*+≡  
 integer(kind=tao\_i32), dimension(DEFAULT\_BUFFER\_SIZE), save, private :: s\_buffer  
 integer, save, private :: s\_buffer\_end = size(s\_buffer)  
 integer, save, private :: s\_last = size(s\_buffer)

Increment the index `last` and reload the array `buffer`, iff this buffer is exhausted. Throughout these routines, `last` will point to random number that has just been consumed. For the array filling routines below, this is simpler than pointing to the next waiting number.

272d *Step last and reload buffer iff necessary 272d*≡  
 last = last + 1  
 if (last > buffer\_end) then  
     call generate (buffer, state)  
     last = 1  
 end if

A random real  $r \in [0, 1)$ . This is almost identical to `tao_random_integer`, but we duplicate the code to avoid the function call overhead for speed.

272e *Implementation of 30-bit tao\_random\_numbers 251a*+≡  
 pure subroutine real\_stateless (state, buffer, buffer\_end, last, r)  
     integer(kind=tao\_i32), dimension(:), intent(inout) :: state, buffer  
     integer, intent(in) :: buffer\_end  
     integer, intent(inout) :: last  
     real(kind=default), intent(out) :: r

```

 real(kind=default), parameter :: NORM = 1.0_default / M
 <Body of tao_random_* 272b>
end subroutine real_stateless

```

A random real  $r \in [0, 1)$ .

273a <Implementation of 52-bit tao\_random\_numbers 256b>+≡

```

 pure subroutine real_stateless (state, buffer, buffer_end, last, r)
 real(kind=tao_r64), dimension(:), intent(inout) :: state, buffer
 integer, intent(in) :: buffer_end
 integer, intent(inout) :: last
 real(kind=default), intent(out) :: r
 integer, parameter :: NORM = 1
 <Body of tao_random_* 272b>
end subroutine real_stateless

```

The low level routine `generate` will fill an array  $a_1, \dots, a_N$ , which will be consumed and refilled like an input buffer.

273b <Variables in 52-bit tao\_random\_numbers 255g>+≡

```

 real(kind=tao_r64), dimension(DEFAULT_BUFFER_SIZE), save, private :: s_buffer
 integer, save, private :: s_buffer_end = size (s_buffer)
 integer, save, private :: s_last = size (s_buffer)

```

### C.4.2 Arrays of Random Numbers

Fill the array  $j_1, \dots, j_\nu$  with random integers  $0 \leq j_i < 2^{30} = 1073741824$ . This has to be done such that the underlying array length in `generate` is transparent to the user. At the same time we want to avoid the overhead of calling `tao_random_real`  $\nu$  times.

273c <Implementation of 30-bit tao\_random\_numbers 251a>+≡

```

 pure subroutine integer_array_stateless &
 (state, buffer, buffer_end, last, v, num)
 integer(kind=tao_i32), dimension(:), intent(inout) :: state, buffer
 integer, intent(in) :: buffer_end
 integer, intent(inout) :: last
 integer, dimension(:), intent(out) :: v
 integer, optional, intent(in) :: num
 integer, parameter :: NORM = 1
 <Body of tao_random*_array 273d>
end subroutine integer_array_stateless

```

273d <Body of tao\_random\*\_array 273d>≡

```

 integer :: nu, done, todo, chunk
 <Set nu to num or size(v) 274a>
 <Prepare array buffer and done, todo, chunk 274b>

```

```

v(1:chunk) = NORM * buffer(last+1:last+chunk)
do
 <Update last, done and todo and set new chunk 274c>
 <Reload buffer or exit 274d>
 v(done+1:done+chunk) = NORM * buffer(1:chunk)
end do

```

274a *<Set nu to num or size(v) 274a>*≡

```

if (present (num)) then
 nu = num
else
 nu = size (v)
end if

```

last is used as an offset into the buffer `buffer`, as usual. `done` is an offset into the target. We still have to process all `nu` numbers. The first chunk can only use what's left in the buffer.

274b *<Prepare array buffer and done, todo, chunk 274b>*≡

```

if (last >= buffer_end) then
 call generate (buffer, state)
 last = 0
end if
done = 0
todo = nu
chunk = min (todo, buffer_end - last)

```

This logic is a bit weird, but after the first chunk, `todo` will either vanish (in which case we're done) or we have consumed all of the buffer and must reload. In any case we can pretend that the next chunk can use the whole buffer.

274c *<Update last, done and todo and set new chunk 274c>*≡

```

last = last + chunk
done = done + chunk
todo = todo - chunk
chunk = min (todo, buffer_end)

```

274d *<Reload buffer or exit 274d>*≡

```

if (chunk <= 0) then
 exit
end if
call generate (buffer, state)
last = 0

```

274e *<Implementation of 30-bit tao\_random\_numbers 251a>*+≡

```

pure subroutine real_array_stateless &
 (state, buffer, buffer_end, last, v, num)

```

```

integer(kind=tao_i32), dimension(:), intent(inout) :: state, buffer
integer, intent(in) :: buffer_end
integer, intent(inout) :: last
real(kind=default), dimension(:), intent(out) :: v
integer, optional, intent(in) :: num
real(kind=default), parameter :: NORM = 1.0_default / M
 <Body of tao_random*_array 273d>
end subroutine real_array_stateless

```

Fill the array  $v_1, \dots, v_\nu$  with uniform deviates  $v_i \in [0, 1)$ .

275a <Implementation of 52-bit tao\_random\_numbers 256b>+≡

```

pure subroutine real_array_stateless &
 (state, buffer, buffer_end, last, v, num)
 real(kind=tao_r64), dimension(:), intent(inout) :: state, buffer
 integer, intent(in) :: buffer_end
 integer, intent(inout) :: last
 real(kind=default), dimension(:), intent(out) :: v
 integer, optional, intent(in) :: num
 integer, parameter :: NORM = 1
 <Body of tao_random*_array 273d>
end subroutine real_array_stateless

```

### C.4.3 Procedures With Explicit *tao\_random\_state*

Unfortunately, this is very boring, but Fortran's lack of parametric polymorphism forces this duplication on us:

275b <Implementation of 30-bit tao\_random\_numbers 251a>+≡

```

elemental subroutine integer_state (s, r)
 type(tao_random_state), intent(inout) :: s
 integer, intent(out) :: r
 call integer_stateless (s%state%x, s%buffer, s%buffer_end, s%last, r)
end subroutine integer_state

```

275c <Implementation of 30-bit tao\_random\_numbers 251a>+≡

```

elemental subroutine real_state (s, r)
 type(tao_random_state), intent(inout) :: s
 real(kind=default), intent(out) :: r
 call real_stateless (s%state%x, s%buffer, s%buffer_end, s%last, r)
end subroutine real_state

```

275d <Implementation of 52-bit tao\_random\_numbers 256b>+≡

```

elemental subroutine real_state (s, r)
 type(tao_random_state), intent(inout) :: s
 real(kind=default), intent(out) :: r

```

```

 call real_stateless (s%state%x, s%buffer, s%buffer_end, s%last, r)
 end subroutine real_state

276a <Implementation of 30-bit tao_random_numbers 251a>+≡
 pure subroutine integer_array_state (s, v, num)
 type(tao_random_state), intent(inout) :: s
 integer, dimension(:), intent(out) :: v
 integer, optional, intent(in) :: num
 call integer_array_stateless &
 (s%state%x, s%buffer, s%buffer_end, s%last, v, num)
 end subroutine integer_array_state

276b <Implementation of 30-bit tao_random_numbers 251a>+≡
 pure subroutine real_array_state (s, v, num)
 type(tao_random_state), intent(inout) :: s
 real(kind=default), dimension(:), intent(out) :: v
 integer, optional, intent(in) :: num
 call real_array_stateless &
 (s%state%x, s%buffer, s%buffer_end, s%last, v, num)
 end subroutine real_array_state

276c <Implementation of 52-bit tao_random_numbers 256b>+≡
 pure subroutine real_array_state (s, v, num)
 type(tao_random_state), intent(inout) :: s
 real(kind=default), dimension(:), intent(out) :: v
 integer, optional, intent(in) :: num
 call real_array_stateless &
 (s%state%x, s%buffer, s%buffer_end, s%last, v, num)
 end subroutine real_array_state

```

#### C.4.4 Static Procedures

First make sure that `tao_random_seed` has been called to initialize the generator state:

```

276d <Initialize a virginal random number generator 276d>≡
 if (s_virginal) then
 call tao_random_seed ()
 end if

276e <Implementation of 30-bit tao_random_numbers 251a>+≡
 subroutine integer_static (r)
 integer, intent(out) :: r
 <Initialize a virginal random number generator 276d>
 call integer_stateless (s_state, s_buffer, s_buffer_end, s_last, r)
 end subroutine integer_static

```

277a *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*  
 subroutine real\_static (r)  
   real(kind=default), intent(out) :: r  
   *<Initialize a virginal random number generator 276d>*  
   call real\_stateless (s\_state, s\_buffer, s\_buffer\_end, s\_last, r)  
 end subroutine real\_static

277b *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*  
 subroutine real\_static (r)  
   real(kind=default), intent(out) :: r  
   *<Initialize a virginal random number generator 276d>*  
   call real\_stateless (s\_state, s\_buffer, s\_buffer\_end, s\_last, r)  
 end subroutine real\_static

277c *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*  
 subroutine integer\_array\_static (v, num)  
   integer, dimension(:), intent(out) :: v  
   integer, optional, intent(in) :: num  
   *<Initialize a virginal random number generator 276d>*  
   call integer\_array\_stateless &  
     (s\_state, s\_buffer, s\_buffer\_end, s\_last, v, num)  
 end subroutine integer\_array\_static

277d *<Implementation of 30-bit tao\_random\_numbers 251a>+≡*  
 subroutine real\_array\_static (v, num)  
   real(kind=default), dimension(:), intent(out) :: v  
   integer, optional, intent(in) :: num  
   *<Initialize a virginal random number generator 276d>*  
   call real\_array\_stateless &  
     (s\_state, s\_buffer, s\_buffer\_end, s\_last, v, num)  
 end subroutine real\_array\_static

277e *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*  
 subroutine real\_array\_static (v, num)  
   real(kind=default), dimension(:), intent(out) :: v  
   integer, optional, intent(in) :: num  
   *<Initialize a virginal random number generator 276d>*  
   call real\_array\_stateless &  
     (s\_state, s\_buffer, s\_buffer\_end, s\_last, v, num)  
 end subroutine real\_array\_static

#### C.4.5 Generic Procedures

277f *<Interfaces of 30-bit tao\_random\_numbers 277f>≡*  
 interface tao\_random\_number

```

 module procedure <Specific procedures for 30-bit tao_random_number 278a>
 end interface

278a <Specific procedures for 30-bit tao_random_number 278a>≡
 integer_static, integer_state, &
 integer_array_static, integer_array_state, &
 real_static, real_state, real_array_static, real_array_state
 These are not exported

278b <Declaration of 30-bit tao_random_numbers 264d>+≡
 private :: &
 integer_stateless, integer_array_stateless, &
 real_stateless, real_array_stateless

278c <Declaration of 30-bit tao_random_numbers 264d>+≡
 private :: <Specific procedures for 30-bit tao_random_number 278a>

278d <Interfaces of 52-bit tao_random_numbers 278d>≡
 interface tao_random_number
 module procedure <Specific procedures for 52-bit tao_random_number 278e>
 end interface

278e <Specific procedures for 52-bit tao_random_number 278e>≡
 real_static, real_state, real_array_static, real_array_state
 Thes are not exported

278f <Declaration of 52-bit tao_random_numbers 265b>+≡
 private :: real_stateless, real_array_stateless

278g <Declaration of 52-bit tao_random_numbers 265b>+≡
 private :: <Specific procedures for 52-bit tao_random_number 278e>

```

### C.4.6 *Luxury*

```

278h <Implementation of tao_random_numbers 251f>+≡
 pure subroutine luxury_stateless &
 (buffer_size, buffer_end, last, consumption)
 integer, intent(in) :: buffer_size
 integer, intent(inout) :: buffer_end
 integer, intent(inout) :: last
 integer, intent(in) :: consumption
 if (consumption >= 1 .and. consumption <= buffer_size) then
 buffer_end = consumption
 last = min (last, buffer_end)
 else
 !!! print *, "tao_random_luxury: ", "invalid consumption ", &
 !!! consumption, ", not in [1,", buffer_size, "]"

```

```

 buffer_end = buffer_size
 end if
end subroutine luxury_stateless

279a <Implementation of tao_random_numbers 251f>+≡
 elemental subroutine luxury_state (s)
 type(tao_random_state), intent(inout) :: s
 call luxury_state_integer (s, size (s%buffer))
 end subroutine luxury_state

279b <Implementation of tao_random_numbers 251f>+≡
 elemental subroutine luxury_state_integer (s, consumption)
 type(tao_random_state), intent(inout) :: s
 integer, intent(in) :: consumption
 call luxury_stateless (size (s%buffer), s%buffer_end, s%last, consumption)
 end subroutine luxury_state_integer

279c <Implementation of tao_random_numbers 251f>+≡
 elemental subroutine luxury_state_real (s, consumption)
 type(tao_random_state), intent(inout) :: s
 real(kind=default), intent(in) :: consumption
 call luxury_state_integer (s, int (consumption * size (s%buffer)))
 end subroutine luxury_state_real

279d <Implementation of tao_random_numbers 251f>+≡
 subroutine luxury_static ()
 <Initialize a virginal random number generator 276d>
 call luxury_static_integer (size (s_buffer))
 end subroutine luxury_static

279e <Implementation of tao_random_numbers 251f>+≡
 subroutine luxury_static_integer (consumption)
 integer, intent(in) :: consumption
 <Initialize a virginal random number generator 276d>
 call luxury_stateless (size (s_buffer), s_buffer_end, s_last, consumption)
 end subroutine luxury_static_integer

279f <Implementation of tao_random_numbers 251f>+≡
 subroutine luxury_static_real (consumption)
 real(kind=default), intent(in) :: consumption
 <Initialize a virginal random number generator 276d>
 call luxury_static_integer (int (consumption * size (s_buffer)))
 end subroutine luxury_static_real

279g <Interfaces of tao_random_numbers (unused luxury) 279g>≡
 interface tao_random_luxury
 module procedure <Specific procedures for tao_random_luxury 280c>
 end interface

```



280a *<Declaration of tao\_random\_numbers (unused luxury) 280a>*≡  
 private :: luxury\_stateless

280b *<Declaration of tao\_random\_numbers (unused luxury) 280a>*+≡  
 private :: *<Specific procedures for tao\_random\_luxury 280c>*

280c *<Specific procedures for tao\_random\_luxury 280c>*≡  
 luxury\_static, luxury\_state, &  
 luxury\_static\_integer, luxury\_state\_integer, &  
 luxury\_static\_real, luxury\_state\_real

## C.5 Testing

### C.5.1 30-bit

280d *<Implementation of 30-bit tao\_random\_numbers 251a>*+≡  
 subroutine tao\_random\_test (name)  
   character(len=\*), optional, intent(in) :: name  
   character (len = \*), parameter :: &  
     OK = "(1x,i10,' is ok. ')", &  
     NOT\_OK = "(1x,i10,' is not ok, (expected ',i10,')!')"  
   *<Parameters in tao\_random\_test 280e>*  
   integer, parameter :: &  
     A\_2027082 = 461390032  
   integer, dimension(N) :: a  
   type( tao\_random\_state ) :: s, t  
   integer, dimension(:), allocatable :: ibuf  
   real(kind=tao\_r64), dimension(:), allocatable :: dbuf  
   integer :: i, ibuf\_size, dbuf\_size  
   print \*, TAO\_RANDOM\_NUMBERS\_RCS\_ID  
   print \*, "testing the 30-bit tao\_random\_numbers ..."  
   *<Perform simple tests of tao\_random\_numbers 281a>*  
   *<Perform more tests of tao\_random\_numbers 281d>*  
 end subroutine tao\_random\_test

280e *<Parameters in tao\_random\_test 280e>*≡  
 integer, parameter :: &  
   SEED = 310952, &  
   N = 2009, M = 1009, &  
   N\_SHORT = 1984

DEK's "official" test expects  $a_{1009 \cdot 2009 + 1} = a_{2027082} = 461390032$ :

```

281a <Perform simple tests of tao_random_numbers 281a>≡
 ! call tao_random_luxury ()
 call tao_random_seed (SEED)
 do i = 1, N+1
 call tao_random_number (a, M)
 end do
 <Test a(1) = A_2027082 281b>

281b <Test a(1) = A_2027082 281b>≡
 if (a(1) == A_2027082) then
 print OK, a(1)
 else
 print NOT_OK, a(1), A_2027082
 end if

```

Deja vu all over again, but 2027081 is factored the other way around this time

```

281c <Perform simple tests of tao_random_numbers 281a>+≡
 call tao_random_seed (SEED)
 do i = 1, M+1
 call tao_random_number (a)
 end do
 <Test a(1) = A_2027082 281b>

```

Now checkpoint the random number generator after  $N_{\text{short}} \cdot M$  numbers

```

281d <Perform more tests of tao_random_numbers 281d>≡
 print *, "testing the stateless stuff ..."
 call tao_random_create (s, SEED)
 do i = 1, N_SHORT
 call tao_random_number (s, a, M)
 end do
 call tao_random_create (t, s)
 do i = 1, N+1 - N_SHORT
 call tao_random_number (s, a, M)
 end do
 <Test a(1) = A_2027082 281b>

```

and restart the saved generator

```

281e <Perform more tests of tao_random_numbers 281d>+≡
 do i = 1, N+1 - N_SHORT
 call tao_random_number (t, a, M)
 end do
 <Test a(1) = A_2027082 281b>

```

The same story again, but this time saving the copy to a file

```
282a <Perform more tests of tao_random_numbers 281d>+≡
 if (present (name)) then
 print *, "testing I/O ..."
 call tao_random_seed (s, SEED)
 do i = 1, N_SHORT
 call tao_random_number (s, a, M)
 end do
 call tao_random_write (s, name)
 do i = 1, N+1 - N_SHORT
 call tao_random_number (s, a, M)
 end do
 <Test a(1) = A_2027082 281b>
 call tao_random_read (s, name)
 do i = 1, N+1 - N_SHORT
 call tao_random_number (s, a, M)
 end do
 <Test a(1) = A_2027082 281b>
 end if
```

And finally using marshaling/unmarshaling:

```
282b <Perform more tests of tao_random_numbers 281d>+≡
 print *, "testing marshaling/unmarshaling ..."
 call tao_random_seed (s, SEED)
 do i = 1, N_SHORT
 call tao_random_number (s, a, M)
 end do
 call tao_random_marshall_size (s, ibuf_size, dbuf_size)
 allocate (ibuf(ibuf_size), dbuf(dbuf_size))
 call tao_random_marshall (s, ibuf, dbuf)
 do i = 1, N+1 - N_SHORT
 call tao_random_number (s, a, M)
 end do
 <Test a(1) = A_2027082 281b>
 call tao_random_unmarshal (s, ibuf, dbuf)
 do i = 1, N+1 - N_SHORT
 call tao_random_number (s, a, M)
 end do
 <Test a(1) = A_2027082 281b>
```

### C.5.2 52-bit

DEK's "official" test expects  $x_{1009:2009+1} = x_{2027082} = 0.27452626307394156768$ :

282c *<Implementation of 52-bit tao\_random\_numbers 256b>+≡*

```

subroutine tao_random_test (name)
 character(len=*), optional, intent(in) :: name
 character(len=*), parameter :: &
 OK = "(1x,f22.20,' is ok. ')", &
 NOT_OK = "(1x,f22.20,' is not ok, (A_2027082 ',f22.20,')!')'"
 <Parameters in tao_random_test 280e>
 real(kind=default), parameter :: &
 A_2027082 = 0.27452626307394156768_default
 real(kind=default), dimension(N) :: a
 type(tao_random_state) :: s, t
 integer, dimension(:), allocatable :: ibuf
 real(kind=tao_r64), dimension(:), allocatable :: dbuf
 integer :: i, ibuf_size, dbuf_size
 print *, TAO52_RANDOM_NUMBERS_RCS_ID
 print *, "testing the 52-bit tao_random_numbers ..."
 <Perform simple tests of tao_random_numbers 281a>
 <Perform more tests of tao_random_numbers 281d>
end subroutine tao_random_test

```

### C.5.3 Test Program

283 *<tao\_test.f90 283>≡*

```

program tao_test
 use tao_random_numbers, only: test30 => tao_random_test
 use tao52_random_numbers, only: test52 => tao_random_test
 implicit none
 call test30 ("tmp.tao")
 call test52 ("tmp.tao")
end program tao_test

```

# —D—

## SPECIAL FUNCTIONS

```

284a <specfun.f90 284a>≡
 ! specfun.f90 --
 <Copyleft notice 1>
 module specfun
 use kinds
 ! use constants
 implicit none
 private
 <Declaration of specfun procedures 284b>
 character(len=*), public, parameter :: SPECFUN_RCS_ID = &
 "$Id: specfun.nw 314 2010-04-17 20:32:33Z ohl $"
 !WK:
 real(kind=default), public, parameter :: &
 PI = 3.1415926535897932384626433832795028841972_default
 contains
 <Implementation of specfun procedures 285c>
 end module specfun

```

The algorithm is stolen from the FORTRAN version in routine C303 of the CERN library [24]. It has an accuracy which is approximately one digit less than machine precision.

```

284b <Declaration of specfun procedures 284b>≡
 public :: gamma

```

The so-called reflection formula is used for negative arguments:

$$\Gamma(x)\Gamma(1-x) = \frac{\pi}{\sin \pi x} \quad (\text{D.1})$$

Here's the identity transformation that pulls the argument of  $\Gamma$  into  $[3, 4]$ :

$$\Gamma(u) = \begin{cases} (u-1)\Gamma(u-1) & \text{for } u > 4 \\ \frac{1}{u}\Gamma(u+1) & \text{for } u < 3 \end{cases} \quad (\text{D.2})$$

285a  $\langle$  Pull  $u$  into the intervall  $[3, 4]$  285a  $\rangle \equiv$

```

f = 1
if (u < 3) then
 do i = 1, int (4 - u)
 f = f / u
 u = u + 1
 end do
else
 do i = 1, int (u - 3)
 u = u - 1
 f = f * u
 end do
end if

```

A Chebyshev approximation for  $\Gamma(x)$  is used after mapping  $x \in [3, 4]$  linearly to  $h \in [-1, 1]$ . The series is evaluted by Clenshaw's recurrence formula:

$$\begin{aligned}
 d_m &= d_{m+1} = 0 \\
 d_j &= 2xd_{j+1} - d_{j+2} + c_j \text{ for } 0 < j < m - 1 \\
 f(x) &= d_0 = xd_1 - d_2 + \frac{1}{2}c_0
 \end{aligned} \tag{D.3}$$

285b  $\langle$  Clenshaw's recurrence formula 285b  $\rangle \equiv$

```

alpha = 2*g
b1 = 0
b2 = 0
do i = 15, 0, -1
 b0 = c(i) + alpha * b1 - b2
 b2 = b1
 b1 = b0
end do
g = f * (b0 - g * b2)

```

Note that we're assuming that  $c(0)$  is in fact  $c_0/2$ . This is for compatibility with the CERN library routines.

285c  $\langle$  Implementation of specfun procedures 285c  $\rangle \equiv$

```

pure function gamma (x) result (g)
 real(kind=default), intent(in) :: x
 real(kind=default) :: g
 integer :: i
 real(kind=default) :: u, f, alpha, b0, b1, b2
 real(kind=default), dimension(0:15), parameter :: &
 c = $\langle c_0/2, c_1, c_2, \dots, c_{15}$ for $\Gamma(x)$ 286a \rangle
 u = x
 if (u <= 0.0) then

```

```

 if (u == int (u)) then
 g = huge (g)
 return
 else
 u = 1 - u
 end if
 endif
 <Pull u into the intervall [3,4] 285a>
 g = 2*u - 7
 <Clenshaw's recurrence formula 285b>
 if (x < 0) then
 g = PI / (sin (PI * x) * g)
 end if
end function gamma

286a <c0/2, c1, c2, ..., c15 for $\Gamma(x)$ 286a>≡
 (/ 3.65738772508338244_default, &
 1.95754345666126827_default, &
 0.33829711382616039_default, &
 0.04208951276557549_default, &
 0.00428765048212909_default, &
 0.00036521216929462_default, &
 0.00002740064222642_default, &
 0.00000181240233365_default, &
 0.00000010965775866_default, &
 0.00000000598718405_default, &
 0.00000000030769081_default, &
 0.00000000001431793_default, &
 0.00000000000065109_default, &
 0.0000000000002596_default, &
 0.0000000000000111_default, &
 0.0000000000000004_default /)

```

## D.1 Test

```

286b <stest.f90 286b>≡
 ! stest.f90 --
 <Copyleft notice 1>
 module stest_functions
 use kinds
 use constants
 use specfun
 end module

```

```

private
 <Declaration of stest_functions procedures 287a>
contains
 <Implementation of stest_functions procedures 287b>
end module stest_functions

```

287a <Declaration of stest\_functions procedures 287a>≡  
 public :: gauss\_multiplication

Gauss' multiplication fomula can serve as a non-trivial test

$$\Gamma(nx) = (2\pi)^{(1-n)/2} n^{nx-1/2} \prod_{k=0}^{n-1} \Gamma(x + k/n) \quad (\text{D.4})$$

287b <Implementation of stest\_functions procedures 287b>≡  
 pure function gauss\_multiplication (x, n) result (delta)  
 real(kind=default), intent(in) :: x  
 integer, intent(in) :: n  
 real(kind=default) :: delta  
 real(kind=default) :: gx  
 integer :: k  
 gx = (2\*PI)\*\*(0.5\_double\*(1-n)) \* n\*\*(n\*x-0.5\_double)  
 do k = 0, n - 1  
 gx = gx \* gamma (x + real (k, kind=default) / n)  
 end do  
 delta = abs ((gamma (n\*x) - gx) / gamma (n\*x))  
end function gauss\_multiplication

287c <stest.f90 286b>+≡  
 program stest  
 use kinds  
 use specfun  
 use stest\_functions !NODEP!  
 implicit none  
 integer :: i, steps  
 real(kind=default) :: x, g, xmin, xmax  
 xmin = -4.5  
 xmax = 4.5  
 steps = 100 ! 9  
 do i = 0, steps  
 x = xmin + ((xmax - xmin) / real (steps)) \* i  
 print "(f6.3,4(1x,e9.2))", x, &  
 gauss\_multiplication (x, 2), &  
 gauss\_multiplication (x, 3), &  
 gauss\_multiplication (x, 4), &



```
 gauss_multiplication (x, 5)
 end do
end program stest
```

# —E—

## STATISTICS

289a  $\langle \text{vamp\_stat.f90 289a} \rangle \equiv$   
`! vamp_stat.f90 --`  
 *$\langle \text{Copyleft notice 1} \rangle$*   
`module vamp_stat`  
`use kinds`  
`implicit none`  
`private`  
 *$\langle \text{Declaration of vamp\_stat procedures 289b} \rangle$*   
`character(len=*), public, parameter :: VAMP_STAT_RCS_ID = &`  
`"$Id: vamp_stat.nw 314 2010-04-17 20:32:33Z ohl $"`  
`contains`  
 *$\langle \text{Implementation of vamp\_stat procedures 289c} \rangle$*   
`end module vamp_stat`

289b  $\langle \text{Declaration of vamp\_stat procedures 289b} \rangle \equiv$   
`public :: average, standard_deviation, value_spread`

$$\text{avg}(X) = \frac{1}{|X|} \sum_{x \in X} x \quad (\text{E.1})$$

289c  $\langle \text{Implementation of vamp\_stat procedures 289c} \rangle \equiv$   
`pure function average (x) result (a)`  
`real(kind=default), dimension(:), intent(in) :: x`  
`real(kind=default) :: a`  
`integer :: n`  
`n = size (x)`  
`if (n == 0) then`  
`a = 0.0`  
`else`  
`a = sum (x) / n`  
`end if`  
`end function average`

$$\text{stddev}(X) = \frac{1}{|X| - 1} \sum_{x \in X} (x - \text{avg}(X))^2 = \frac{1}{|X| - 1} \left( \frac{1}{|X|} \sum_{x \in X} x^2 - (\text{avg}(X))^2 \right) \quad (\text{E.2})$$

290a *⟨Implementation of vamp\_stat procedures 289c⟩*+≡  
 pure function standard\_deviation (x) result (s)  
   real(kind=default), dimension(:), intent(in) :: x  
   real(kind=default) :: s  
   integer :: n  
   n = size (x)  
   if (n < 2) then  
     s = huge (s)  
   else  
     s = sqrt (max ((sum (x\*\*2) / n - (average (x))\*\*2) / (n - 1), &  
                   0.0\_default))  
   end if  
end function standard\_deviation

$$\text{spread}(X) = \max_{x \in X}(x) - \min_{x \in X}(x) \quad (\text{E.3})$$

290b *⟨Implementation of vamp\_stat procedures 289c⟩*+≡  
 pure function value\_spread (x) result (s)  
   real(kind=default), dimension(:), intent(in) :: x  
   real(kind=default) :: s  
   s = maxval(x) - minval(x)  
end function value\_spread

290c *⟨Declaration of vamp\_stat procedures 289b⟩*+≡  
 public :: standard\_deviation\_percent, value\_spread\_percent

290d *⟨Implementation of vamp\_stat procedures 289c⟩*+≡  
 pure function standard\_deviation\_percent (x) result (s)  
   real(kind=default), dimension(:), intent(in) :: x  
   real(kind=default) :: s  
   real(kind=default) :: abs\_avg  
   abs\_avg = abs (average (x))  
   if (abs\_avg <= tiny (abs\_avg)) then  
     s = huge (s)  
   else  
     s = 100.0 \* standard\_deviation (x) / abs\_avg  
   end if  
end function standard\_deviation\_percent

290e *⟨Implementation of vamp\_stat procedures 289c⟩*+≡  
 pure function value\_spread\_percent (x) result (s)  
   real(kind=default), dimension(:), intent(in) :: x

```

real(kind=default) :: s
real(kind=default) :: abs_avg
abs_avg = abs (average (x))
if (abs_avg <= tiny (abs_avg)) then
 s = huge (s)
else
 s = 100.0 * value_spread (x) / abs_avg
end if
end function value_spread_percent

```

# —F—

## HISTOGRAMMING

⚡ Merged WK's improvements for WHIZARD. TODO *after* merging:

1. bins3 is a bad undescriptive name
2. bins3 should be added to histogram2
3. write\_histogram2\_unit for symmetry.

⚡ There's almost no sanity checking. If you call one of these functions on a histogram that has not been initialized, you loose. — *Big time.*

```
292a <histograms.f90 292a>≡
! histograms.f90 --
<Copyleft notice 1>
module histograms
 use kinds
 use utils, only: find_free_unit
 implicit none
 private
 <Declaration of histograms procedures 293b>
 <Interfaces of histograms procedures 293c>
 <Variables in histograms 293e>
 <Declaration of histograms types 292b>
 character(len=*), public, parameter :: HISTOGRAMS_RCS_ID = &
 "$Id: histograms.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of histograms procedures 293f>
end module histograms

292b <Declaration of histograms types 292b>≡
type, public :: histogram
 private
 integer :: n_bins
 real(kind=default) :: x_min, x_max
```

```

 real(kind=default), dimension(:), pointer :: bins => null ()
 real(kind=default), dimension(:), pointer :: bins2 => null ()
 real(kind=default), dimension(:), pointer :: bins3 => null ()
 end type histogram

```

293a  $\langle$ Declaration of histograms types 292b $\rangle + \equiv$

```

type, public :: histogram2
 private
 integer, dimension(2) :: n_bins
 real(kind=default), dimension(2) :: x_min, x_max
 real(kind=default), dimension(:,:), pointer :: bins => null ()
 real(kind=default), dimension(:,:), pointer :: bins2 => null ()
end type histogram2

```

293b  $\langle$ Declaration of histograms procedures 293b $\rangle \equiv$

```

public :: create_histogram
public :: fill_histogram
public :: delete_histogram
public :: write_histogram

```

293c  $\langle$ Interfaces of histograms procedures 293c $\rangle \equiv$

```

interface create_histogram
 module procedure create_histogram1, create_histogram2
end interface
interface fill_histogram
 module procedure fill_histogram1, fill_histogram2s, fill_histogram2v
end interface
interface delete_histogram
 module procedure delete_histogram1, delete_histogram2
end interface
interface write_histogram
 module procedure write_histogram1, write_histogram2
 module procedure write_histogram1_unit
end interface

```

293d  $\langle$ Declaration of histograms procedures 293b $\rangle + \equiv$

```

private :: create_histogram1, create_histogram2
private :: fill_histogram1, fill_histogram2s, fill_histogram2v
private :: delete_histogram1, delete_histogram2
private :: write_histogram1, write_histogram2

```

293e  $\langle$ Variables in histograms 293e $\rangle \equiv$

```

integer, parameter, private :: N_BINS_DEFAULT = 10

```

293f  $\langle$ Implementation of histograms procedures 293f $\rangle \equiv$

```

elemental subroutine create_histogram1 (h, x_min, x_max, nb)
 type(histogram), intent(out) :: h

```

```

real(kind=default), intent(in) :: x_min, x_max
integer, intent(in), optional :: nb
if (present (nb)) then
 h%n_bins = nb
else
 h%n_bins = N_BINS_DEFAULT
end if
h%x_min = x_min
h%x_max = x_max
allocate (h%bins(0:h%n_bins+1), h%bins2(0:h%n_bins+1))
h%bins = 0
h%bins2 = 0
allocate (h%bins3(0:h%n_bins+1))
h%bins3 = 0
end subroutine create_histogram1

```

294a *⟨Implementation of histograms procedures 293f⟩* +=

```

pure subroutine create_histogram2 (h, x_min, x_max, nb)
type(histogram2), intent(out) :: h
real(kind=default), dimension(:), intent(in) :: x_min, x_max
integer, intent(in), dimension(:), optional :: nb
if (present (nb)) then
 h%n_bins = nb
else
 h%n_bins = N_BINS_DEFAULT
end if
h%x_min = x_min
h%x_max = x_max
allocate (h%bins(0:h%n_bins(1)+1,0:h%n_bins(1)+1), &
 h%bins2(0:h%n_bins(2)+1,0:h%n_bins(2)+1))
h%bins = 0
h%bins2 = 0
end subroutine create_histogram2

```

294b *⟨Implementation of histograms procedures 293f⟩* +=

```

elemental subroutine fill_histogram1 (h, x, weight, excess)
type(histogram), intent(inout) :: h
real(kind=default), intent(in) :: x
real(kind=default), intent(in), optional :: weight
real(kind=default), intent(in), optional :: excess
integer :: i
if (x < h%x_min) then
 i = 0
else if (x > h%x_max) then
 i = h%n_bins + 1

```

```

else
 i = 1 + h%n_bins * (x - h%x_min) / (h%x_max - h%x_min)
!WK! i = min (max (i, 0), h%n_bins + 1)
end if
if (present (weight)) then
 h%bins(i) = h%bins(i) + weight
 h%bins2(i) = h%bins2(i) + weight*weight
else
 h%bins(i) = h%bins(i) + 1
 h%bins2(i) = h%bins2(i) + 1
end if
if (present (excess)) h%bins3(i) = h%bins3(i) + excess
end subroutine fill_histogram1

```

295a *⟨Implementation of histograms procedures 293f⟩*+≡  
 elemental subroutine fill\_histogram2s (h, x1, x2, weight)  
 type(histogram2), intent(inout) :: h  
 real(kind=default), intent(in) :: x1, x2  
 real(kind=default), intent(in), optional :: weight  
 call fill\_histogram2v (h, (/ x1, x2 /), weight)  
 end subroutine fill\_histogram2s

295b *⟨Implementation of histograms procedures 293f⟩*+≡  
 pure subroutine fill\_histogram2v (h, x, weight)  
 type(histogram2), intent(inout) :: h  
 real(kind=default), dimension(:), intent(in) :: x  
 real(kind=default), intent(in), optional :: weight  
 integer, dimension(2) :: i  
 i = 1 + h%n\_bins \* (x - h%x\_min) / (h%x\_max - h%x\_min)  
 i = min (max (i, 0), h%n\_bins + 1)  
 if (present (weight)) then  
 h%bins(i(1),i(2)) = h%bins(i(1),i(2)) + weight  
 h%bins2(i(1),i(2)) = h%bins2(i(1),i(2)) + weight\*weight  
 else  
 h%bins(i(1),i(2)) = h%bins(i(1),i(2)) + 1  
 h%bins2(i(1),i(2)) = h%bins2(i(1),i(2)) + 1  
 end if  
 end subroutine fill\_histogram2v

295c *⟨Implementation of histograms procedures 293f⟩*+≡  
 elemental subroutine delete\_histogram1 (h)  
 type(histogram), intent(inout) :: h  
 deallocate (h%bins, h%bins2)  
 deallocate (h%bins3)  
 end subroutine delete\_histogram1



```

296a <Implementation of histograms procedures 293f>+≡
 elemental subroutine delete_histogram2 (h)
 type(histogram2), intent(inout) :: h
 deallocate (h%bins, h%bins2)
 end subroutine delete_histogram2

296b <Implementation of histograms procedures 293f>+≡
 subroutine write_histogram1 (h, name, over)
 type(histogram), intent(in) :: h
 character(len=*), intent(in), optional :: name
 logical, intent(in), optional :: over
 integer :: i, iounit
 if (present (name)) then
 call find_free_unit (iounit)
 if (iounit > 0) then
 open (unit = iounit, action = "write", status = "replace", &
 file = name)
 if (present (over)) then
 if (over) then
 write (unit = iounit, fmt = *) &
 "underflow", h%bins(0), sqrt (h%bins2(0))
 end if
 end if
 do i = 1, h%n_bins
 write (unit = iounit, fmt = *) &
 midpoint (h, i), h%bins(i), sqrt (h%bins2(i))
 end do
 if (present (over)) then
 if (over) then
 write (unit = iounit, fmt = *) &
 "overflow", h%bins(h%n_bins+1), &
 sqrt (h%bins2(h%n_bins+1))
 end if
 end if
 close (unit = iounit)
 else
 print *, "write_histogram: Can't find a free unit!"
 end if
 else
 if (present (over)) then
 if (over) then
 print *, "underflow", h%bins(0), sqrt (h%bins2(0))
 end if
 end if
 end if
 end subroutine write_histogram1

```

```

do i = 1, h%n_bins
 print *, midpoint (h, i), h%bins(i), sqrt (h%bins2(i))
end do
if (present (over)) then
 if (over) then
 print *, "overflow", h%bins(h%n_bins+1), &
 sqrt (h%bins2(h%n_bins+1))
 end if
end if
end if
end subroutine write_histogram1

```

297a *<Declaration of histograms procedures 293b>+≡*  
 !WK! public :: write\_histogram1\_unit



I don't like the `format` statement with the line number. Use a character constant instead (after we have merged with WHIZARD's branch).

297b *<Implementation of histograms procedures 293f>+≡*

```

subroutine write_histogram1_unit (h, iounit, over, show_excess)
 type(histogram), intent(in) :: h
 integer, intent(in) :: iounit
 logical, intent(in), optional :: over, show_excess
 integer :: i
 logical :: show_exc
 show_exc = .false.; if (present(show_excess)) show_exc = show_excess
 if (present (over)) then
 if (over) then
 if (show_exc) then
 write (unit = iounit, fmt = 1) &
 "underflow", h%bins(0), sqrt (h%bins2(0)), h%bins3(0)
 else
 write (unit = iounit, fmt = 1) &
 "underflow", h%bins(0), sqrt (h%bins2(0))
 end if
 end if
 end if
 do i = 1, h%n_bins
 if (show_exc) then
 write (unit = iounit, fmt = 1) &
 midpoint (h, i), h%bins(i), sqrt (h%bins2(i)), h%bins3(i)
 else
 write (unit = iounit, fmt = 1) &
 midpoint (h, i), h%bins(i), sqrt (h%bins2(i))
 end if
 end do
end subroutine write_histogram1_unit

```

```

 end if
 end do
 if (present (over)) then
 if (over) then
 if (show_exc) then
 write (unit = iounit, fmt = 1) &
 "overflow", h%bins(h%n_bins+1), &
 sqrt (h%bins2(h%n_bins+1)), &
 h%bins3(h%n_bins+1)
 else
 write (unit = iounit, fmt = 1) &
 "overflow", h%bins(h%n_bins+1), &
 sqrt (h%bins2(h%n_bins+1))
 end if
 end if
 end if
 1 format (1x,4(G16.9,2x))
end subroutine write_histogram1_unit

```

298a *<Declaration of histograms procedures 293b>+≡*  
 private :: midpoint

298b *<Interfaces of histograms procedures 293c>+≡*  
 interface midpoint  
 module procedure midpoint1, midpoint2  
 end interface

298c *<Declaration of histograms procedures 293b>+≡*  
 private :: midpoint1, midpoint2

298d *<Implementation of histograms procedures 293f>+≡*  
 elemental function midpoint1 (h, bin) result (x)  
 type(histogram), intent(in) :: h  
 integer, intent(in) :: bin  
 real(kind=default) :: x  
 x = h%x\_min + (h%x\_max - h%x\_min) \* (bin - 0.5) / h%n\_bins  
 end function midpoint1

298e *<Implementation of histograms procedures 293f>+≡*  
 elemental function midpoint2 (h, bin, d) result (x)  
 type(histogram2), intent(in) :: h  
 integer, intent(in) :: bin, d  
 real(kind=default) :: x  
 x = h%x\_min(d) + (h%x\_max(d) - h%x\_min(d)) \* (bin - 0.5) / h%n\_bins(d)  
 end function midpoint2

299 *⟨Implementation of histograms procedures 293f⟩*+≡

```

subroutine write_histogram2 (h, name, over)
 type(histogram2), intent(in) :: h
 character(len=*), intent(in), optional :: name
 logical, intent(in), optional :: over
 integer :: i1, i2, iounit
 if (present (name)) then
 call find_free_unit (iounit)
 if (iounit > 0) then
 open (unit = iounit, action = "write", status = "replace", &
 file = name)
 if (present (over)) then
 if (over) then
 write (unit = iounit, fmt = *) &
 "double underflow", h%bins(0,0), sqrt (h%bins2(0,0))
 do i2 = 1, h%n_bins(2)
 write (unit = iounit, fmt = *) &
 "x1 underflow", midpoint (h, i2, 2), &
 h%bins(0,i2), sqrt (h%bins2(0,i2))
 end do
 do i1 = 1, h%n_bins(1)
 write (unit = iounit, fmt = *) &
 "x2 underflow", midpoint (h, i1, 1), &
 h%bins(i1,0), sqrt (h%bins2(i1,0))
 end do
 end if
 end if
 do i1 = 1, h%n_bins(1)
 do i2 = 1, h%n_bins(2)
 write (unit = iounit, fmt = *) &
 midpoint (h, i1, 1), midpoint (h, i2, 2), &
 h%bins(i1,i2), sqrt (h%bins2(i1,i2))
 end do
 end do
 if (present (over)) then
 if (over) then
 do i2 = 1, h%n_bins(2)
 write (unit = iounit, fmt = *) &
 "x1 overflow", midpoint (h, i2, 2), &
 h%bins(h%n_bins(1)+1,i2), &
 sqrt (h%bins2(h%n_bins(1)+1,i2))
 end do
 do i1 = 1, h%n_bins(1)

```

```

 write (unit = iounit, fmt = *) &
 "x2 overflow", midpoint (h, i1, 1), &
 h%bins(i1,h%n_bins(2)+1), &
 sqrt (h%bins2(i1,h%n_bins(2)+1))
 end do
 write (unit = iounit, fmt = *) "double overflow", &
 h%bins(h%n_bins(1)+1,h%n_bins(2)+1), &
 sqrt (h%bins2(h%n_bins(1)+1,h%n_bins(2)+1))
 end if
end if
close (unit = iounit)
else
 print *, "write_histogram: Can't find a free unit!"
end if
else
 if (present (over)) then
 if (over) then
 print *, "double underflow", h%bins(0,0), sqrt (h%bins2(0,0))
 do i2 = 1, h%n_bins(2)
 print *, "x1 underflow", midpoint (h, i2, 2), &
 h%bins(0,i2), sqrt (h%bins2(0,i2))
 end do
 do i1 = 1, h%n_bins(1)
 print *, "x2 underflow", midpoint (h, i1, 1), &
 h%bins(i1,0), sqrt (h%bins2(i1,0))
 end do
 end if
 end if
 do i1 = 1, h%n_bins(1)
 do i2 = 1, h%n_bins(2)
 print *, midpoint (h, i1, 1), midpoint (h, i2, 2), &
 h%bins(i1,i2), sqrt (h%bins2(i1,i2))
 end do
 end do
 if (present (over)) then
 if (over) then
 do i2 = 1, h%n_bins(2)
 print *, "x1 overflow", midpoint (h, i2, 2), &
 h%bins(h%n_bins(1)+1,i2), &
 sqrt (h%bins2(h%n_bins(1)+1,i2))
 end do
 do i1 = 1, h%n_bins(1)
 print *, "x2 overflow", midpoint (h, i1, 1), &

```

```

 h%bins(i1,h%n_bins(2)+1), &
 sqrt (h%bins2(i1,h%n_bins(2)+1))
 end do
 print *, "double overflow", &
 h%bins(h%n_bins(1)+1,h%n_bins(2)+1), &
 sqrt (h%bins2(h%n_bins(1)+1,h%n_bins(2)+1))
 end if
end if
end if
end subroutine write_histogram2

```

# —G—

## MISCELLANEOUS UTILITIES

```

302a <utils.f90 302a>≡
 ! utils.f90 --
 <Copyleft notice 1>
 module utils
 use kinds
 implicit none
 private
 <Declaration of utils procedures 302b>
 <Parameters in utils 309c>
 <Variables in utils 310b>
 <Interfaces of utils procedures 302c>
 character(len=*), public, parameter :: UTILS_RCS_ID = &
 "$Id: utils.nw 314 2010-04-17 20:32:33Z ohl $"
 contains
 <Implementation of utils procedures 303c>
 end module utils

```

### *G.1 Memory Management*

```

302b <Declaration of utils procedures 302b>≡
 public :: create_array_pointer
 private :: create_integer_array_pointer
 private :: create_real_array_pointer
 private :: create_integer_array2_pointer
 private :: create_real_array2_pointer

302c <Interfaces of utils procedures 302c>≡
 interface create_array_pointer
 module procedure &
 create_integer_array_pointer, &

```

```

 create_real_array_pointer, &
 create_integer_array2_pointer, &
 create_real_array2_pointer
 end interface
303a <Body of create_*_array_pointer 303a>≡
 if (associated (lhs)) then
 if (size (lhs) /= n) then
 deallocate (lhs)
 if (present (lb)) then
 allocate (lhs(lb:n+lb-1))
 else
 allocate (lhs(n))
 end if
 end if
 else
 if (present (lb)) then
 allocate (lhs(lb:n+lb-1))
 else
 allocate (lhs(n))
 end if
 end if
 lhs = 0
303b <Body of create_*_array2_pointer 303b>≡
 if (associated (lhs)) then
 if (any (ubound (lhs) /= n)) then
 deallocate (lhs)
 if (present (lb)) then
 allocate (lhs(lb(1):n(1)+lb(1)-1,lb(2):n(2)+lb(2)-1))
 else
 allocate (lhs(n(1),n(2)))
 end if
 end if
 else
 if (present (lb)) then
 allocate (lhs(lb(1):n(1)+lb(1)-1,lb(2):n(2)+lb(2)-1))
 else
 allocate (lhs(n(1),n(2)))
 end if
 end if
 lhs = 0
303c <Implementation of utils procedures 303c>≡
 pure subroutine create_integer_array_pointer (lhs, n, lb)

```



```

integer, dimension(:), pointer :: lhs
integer, intent(in) :: n
integer, intent(in), optional :: lb
<Body of create_*_array_pointer 303a>
end subroutine create_integer_array_pointer

```

304a <Implementation of utils procedures 303c>+≡

```

pure subroutine create_real_array_pointer (lhs, n, lb)
 real(kind=default), dimension(:), pointer :: lhs
 integer, intent(in) :: n
 integer, intent(in), optional :: lb
 <Body of create_*_array_pointer 303a>
end subroutine create_real_array_pointer

```

304b <Implementation of utils procedures 303c>+≡

```

pure subroutine create_integer_array2_pointer (lhs, n, lb)
 integer, dimension(:,:), pointer :: lhs
 integer, dimension(:), intent(in) :: n
 integer, dimension(:), intent(in), optional :: lb
 <Body of create_*_array2_pointer 303b>
end subroutine create_integer_array2_pointer

```

304c <Implementation of utils procedures 303c>+≡

```

pure subroutine create_real_array2_pointer (lhs, n, lb)
 real(kind=default), dimension(:,:), pointer :: lhs
 integer, dimension(:), intent(in) :: n
 integer, dimension(:), intent(in), optional :: lb
 <Body of create_*_array2_pointer 303b>
end subroutine create_real_array2_pointer

```

Copy an allocatable array component of a derived type, reshaping the target if necessary. The target can be disassociated, but its association *must not* be undefined.

304d <Declaration of utils procedures 302b>+≡

```

public :: copy_array_pointer
private :: copy_integer_array_pointer
private :: copy_real_array_pointer
private :: copy_integer_array2_pointer
private :: copy_real_array2_pointer

```

304e <Interfaces of utils procedures 302c>+≡

```

interface copy_array_pointer
 module procedure &
 copy_integer_array_pointer, &
 copy_real_array_pointer, &
 copy_integer_array2_pointer, &

```

```

 copy_real_array2_pointer
 end interface

305a <Implementation of utils procedures 303c>+≡
 pure subroutine copy_integer_array_pointer (lhs, rhs, lb)
 integer, dimension(:), pointer :: lhs
 integer, dimension(:), intent(in) :: rhs
 integer, intent(in), optional :: lb
 call create_integer_array_pointer (lhs, size (rhs), lb)
 lhs = rhs
 end subroutine copy_integer_array_pointer

305b <Implementation of utils procedures 303c>+≡
 pure subroutine copy_real_array_pointer (lhs, rhs, lb)
 real(kind=default), dimension(:), pointer :: lhs
 real(kind=default), dimension(:), intent(in) :: rhs
 integer, intent(in), optional :: lb
 call create_real_array_pointer (lhs, size (rhs), lb)
 lhs = rhs
 end subroutine copy_real_array_pointer

305c <Implementation of utils procedures 303c>+≡
 pure subroutine copy_integer_array2_pointer (lhs, rhs, lb)
 integer, dimension(:, :), pointer :: lhs
 integer, dimension(:, :), intent(in) :: rhs
 integer, dimension(:), intent(in), optional :: lb
 call create_integer_array2_pointer &
 (lhs, (/ size (rhs, dim=1), size (rhs, dim=2) /), lb)
 lhs = rhs
 end subroutine copy_integer_array2_pointer

305d <Implementation of utils procedures 303c>+≡
 pure subroutine copy_real_array2_pointer (lhs, rhs, lb)
 real(kind=default), dimension(:, :), pointer :: lhs
 real(kind=default), dimension(:, :), intent(in) :: rhs
 integer, dimension(:), intent(in), optional :: lb
 call create_real_array2_pointer &
 (lhs, (/ size (rhs, dim=1), size (rhs, dim=2) /), lb)
 lhs = rhs
 end subroutine copy_real_array2_pointer

```

## G.2 Sorting

```

305e <Declaration of utils procedures 302b>+≡
 public :: swap

```

```

 private :: swap_integer, swap_real
306a <Interfaces of utils procedures 302c>+≡
 interface swap
 module procedure swap_integer, swap_real
 end interface
306b <Implementation of utils procedures 303c>+≡
 elemental subroutine swap_integer (a, b)
 integer, intent(inout) :: a, b
 integer :: tmp
 tmp = a
 a = b
 b = tmp
 end subroutine swap_integer
306c <Implementation of utils procedures 303c>+≡
 elemental subroutine swap_real (a, b)
 real(kind=default), intent(inout) :: a, b
 real(kind=default) :: tmp
 tmp = a
 a = b
 b = tmp
 end subroutine swap_real
Straight insertion:
306d <Implementation of utils procedures 303c>+≡
 pure subroutine sort_real (key, reverse)
 real(kind=default), dimension(:), intent(inout) :: key
 logical, intent(in), optional :: reverse
 logical :: rev
 integer :: i, j
 <Set rev to reverse or .false. 306e>
 do i = 1, size (key) - 1
 <Set j to minloc(key) 307a>
 if (j /= i) then
 call swap (key(i), key(j))
 end if
 end do
 end subroutine sort_real
306e <Set rev to reverse or .false. 306e>≡
 if (present (reverse)) then
 rev = reverse
 else
 rev = .false.
 end if

```

```

307a <Set j to minloc(key) 307a>≡
 if (rev) then
 j = sum (maxloc (key(i:))) + i - 1
 else
 j = sum (minloc (key(i:))) + i - 1
 end if

307b <Implementation of utils procedures 303c>+≡
 pure subroutine sort_real_and_real_array (key, table, reverse)
 real(kind=default), dimension(:), intent(inout) :: key
 real(kind=default), dimension(:, :), intent(inout) :: table
 logical, intent(in), optional :: reverse
 logical :: rev
 integer :: i, j
 <Set rev to reverse or .false. 306e>
 do i = 1, size (key) - 1
 <Set j to minloc(key) 307a>
 if (j /= i) then
 call swap (key(i), key(j))
 call swap (table(:, i), table(:, j))
 end if
 end do
 end subroutine sort_real_and_real_array

307c <Implementation of utils procedures 303c>+≡
 pure subroutine sort_real_and_integer (key, table, reverse)
 real(kind=default), dimension(:), intent(inout) :: key
 integer, dimension(:), intent(inout) :: table
 logical, intent(in), optional :: reverse
 logical :: rev
 integer :: i, j
 <Set rev to reverse or .false. 306e>
 do i = 1, size (key) - 1
 <Set j to minloc(key) 307a>
 if (j /= i) then
 call swap (key(i), key(j))
 call swap (table(i), table(j))
 end if
 end do
 end subroutine sort_real_and_integer

307d <Declaration of utils procedures 302b>+≡
 public :: sort
 private :: sort_real, sort_real_and_real_array, sort_real_and_integer

```

308a *⟨Interfaces of utils procedures 302c⟩*+≡  

```

interface sort
 module procedure &
 sort_real, sort_real_and_real_array, &
 sort_real_and_integer
end interface

```

### G.3 Mathematics

308b *⟨Declaration of utils procedures 302b⟩*+≡  

```

public :: outer_product

```

Admittedly, one has to get used to this notation for the tensor product:

308c *⟨Implementation of utils procedures 303c⟩*+≡  

```

pure function outer_product (x, y) result (xy)
 real(kind=default), dimension(:), intent(in) :: x, y
 real(kind=default), dimension(size(x),size(y)) :: xy
 xy = spread (x, dim=2, ncopies=size(y)) &
 * spread (y, dim=1, ncopies=size(x))
end function outer_product

```

Greatest common divisor and least common multiple

308d *⟨Declaration of utils procedures 302b⟩*+≡  

```

public :: factorize, gcd, lcm
private :: gcd_internal

```

For our purposes, a straightforward implementation of Euclid's algorithm suffices:

308e *⟨Implementation of utils procedures 303c⟩*+≡  

```

pure recursive function gcd_internal (m, n) result (gcd_m_n)
 integer, intent(in) :: m, n
 integer :: gcd_m_n
 if (n <= 0) then
 gcd_m_n = m
 else
 gcd_m_n = gcd_internal (n, modulo (m, n))
 end if
end function gcd_internal

```

Wrap an elemental procedure around the recursive procedure:

308f *⟨Implementation of utils procedures 303c⟩*+≡  

```

elemental function gcd (m, n) result (gcd_m_n)
 integer, intent(in) :: m, n
 integer :: gcd_m_n

```

```

gcd_m_n = gcd_internal (m, n)
end function gcd

```

As long as  $m \cdot n$  does not overflow, we can use  $\text{gcd}(m, n) \text{lcm}(m, n) = mn$ :

309a *<Implementation of utils procedures 303c>+≡*

```

elemental function lcm (m, n) result (lcm_m_n)
 integer, intent(in) :: m, n
 integer :: lcm_m_n
 lcm_m_n = (m * n) / gcd (m, n)
end function lcm

```

A very simple minded factorization procedure, that is not fool proof at all. It maintains  $n == \text{product}(\text{factors}(1:i))$ , however, and will work in all cases of practical relevance.

309b *<Implementation of utils procedures 303c>+≡*

```

pure subroutine factorize (n, factors, i)
 integer, intent(in) :: n
 integer, dimension(:), intent(out) :: factors
 integer, intent(out) :: i
 integer :: nn, p
 nn = n
 i = 0
 do p = 1, size (PRIMES)
 try: do
 if (modulo (nn, PRIMES(p)) == 0) then
 i = i + 1
 factors(i) = PRIMES(p)
 nn = nn / PRIMES(p)
 if (i >= size (factors)) then
 factors(i) = nn
 return
 end if
 else
 exit try
 end if
 end do try
 if (nn == 1) then
 return
 end if
 end do
end subroutine factorize

```

309c *<Parameters in utils 309c>≡*

```

integer, dimension(13), parameter, private :: &
 PRIMES = (/ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 /)

```

## G.4 I/O

310a *<Declaration of utils procedures 302b>+≡*

```
public :: find_free_unit
```

310b *<Variables in utils 310b>≡*

```
integer, parameter, private :: MIN_UNIT = 11, MAX_UNIT = 99
```

310c *<Implementation of utils procedures 303c>+≡*

```
subroutine find_free_unit (u, iostat)
 integer, intent(out) :: u
 integer, intent(out), optional :: iostat
 logical :: exists, is_open
 integer :: i, status
 do i = MIN_UNIT, MAX_UNIT
 inquire (unit = i, exist = exists, opened = is_open, &
 iostat = status)
 if (status == 0) then
 if (exists .and. .not. is_open) then
 u = i
 if (present (iostat)) then
 iostat = 0
 end if
 return
 end if
 end if
 end do
 if (present (iostat)) then
 iostat = -1
 end if
 u = -1
end subroutine find_free_unit
```

# —H— LINEAR ALGEBRA

```

311a <linalg.f90 311a>≡
! linalg.f90 --
<Copyleft notice 1>
module linalg
 use kinds
 use utils
 implicit none
 private
 <Declaration of linalg procedures 311b>
 character(len=*), public, parameter :: LINALG_RCS_ID = &
 "$Id: linalg.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of linalg procedures 312>
end module linalg

```

## *H.1 LU Decomposition*

```

311b <Declaration of linalg procedures 311b>≡
 public :: lu_decompose

```

$$A = LU \tag{H.1a}$$

In more detail

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix} \tag{H.1b}$$



Rewriting (H.1) in block matrix notation

$$\begin{pmatrix} a_{11} & a_{1\cdot} \\ a_{\cdot 1} & A \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ l_{\cdot 1} & L \end{pmatrix} \begin{pmatrix} u_{11} & u_{1\cdot} \\ 0 & U \end{pmatrix} = \begin{pmatrix} u_{11} & u_{1\cdot} \\ l_{\cdot 1} u_{11} & l_{\cdot 1} \otimes u_{1\cdot} + LU \end{pmatrix} \quad (\text{H.2})$$

we can solve it easily

$$u_{11} = a_{11} \quad (\text{H.3a})$$

$$u_{1\cdot} = a_{1\cdot} \quad (\text{H.3b})$$

$$l_{\cdot 1} = \frac{a_{\cdot 1}}{a_{11}} \quad (\text{H.3c})$$

$$LU = A - \frac{a_{\cdot 1} \otimes a_{1\cdot}}{a_{11}} \quad (\text{H.3d})$$

and (H.3c) and (H.3d) define a simple iterative algorithm if we work from the outside in. It just remains to add pivoting.

312 *<Implementation of linalg procedures 312>*≡

```

pure subroutine lu_decompose (a, pivots, eps, l, u)
 real(kind=default), dimension(:, :), intent(inout) :: a
 integer, dimension(:), intent(out), optional :: pivots
 real(kind=default), intent(out), optional :: eps
 real(kind=default), dimension(:, :), intent(out), optional :: l, u
 real(kind=default), dimension(size(a,dim=1)) :: vv
 integer, dimension(size(a,dim=1)) :: p
 integer :: j, pivot
 <eps = 1 313a>
 vv = maxval (abs (a), dim=2)
 if (any (vv == 0.0)) then
 a = 0.0
 <pivots = 0 and eps = 0 313c>
 return
 end if
 vv = 1.0 / vv
 do j = 1, size (a, dim=1)
 pivot = j - 1 + sum (maxloc (vv(j:) * abs (a(j:,j))))
 if (j /= pivot) then
 call swap (a(pivot,:), a(j,:))
 <eps = - eps 313b>
 vv(pivot) = vv(j)
 end if
 p(j) = pivot
 if (a(j,j) == 0.0) then
 a(j,j) = tiny (a(j,j))

```

```

 end if
 a(j+1:,j) = a(j+1:,j) / a(j,j)
 a(j+1:,j+1:) &
 = a(j+1:,j+1:) - outer_product (a(j+1:,j), a(j,j+1:))
 end do
 <Return optional arguments in lu_decompose 313d>
end subroutine lu_decompose

313a <eps = 1 313a>≡
 if (present (eps)) then
 eps = 1.0
 end if

313b <eps = - eps 313b>≡
 if (present (eps)) then
 eps = - eps
 end if

313c <pivots = 0 and eps = 0 313c>≡
 if (present (pivots)) then
 pivots = 0
 end if
 if (present (eps)) then
 eps = 0
 end if

313d <Return optional arguments in lu_decompose 313d>≡
 if (present (pivots)) then
 pivots = p
 end if
 if (present (l)) then
 do j = 1, size (a, dim=1)
 l(1:j-1,j) = 0.0
 l(j,j) = 1.0
 l(j+1:,j) = a(j+1:,j)
 end do
 do j = size (a, dim=1), 1, -1
 call swap (l(j,:), l(p(j),:))
 end do
 end if
 if (present (u)) then
 do j = 1, size (a, dim=1)
 u(1:j,j) = a(1:j,j)
 u(j+1:,j) = 0.0
 end do
 end if

```

## H.2 Determinant

314a  $\langle$ Declaration of `linalg` procedures 311b $\rangle + \equiv$   
`public :: determinant`

This is a subroutine to comply with F's rules, otherwise, we would code it as a function.

314b  $\langle$ Implementation of `linalg` procedures 312 $\rangle + \equiv$   
`pure subroutine determinant (a, det)`  
`real(kind=default), dimension(:, :), intent(in) :: a`  
`real(kind=default), intent(out) :: det`  
`real(kind=default), dimension(size(a,dim=1),size(a,dim=2)) :: lu`  
`integer :: i`  
`lu = a`  
`call lu_decompose (lu, eps = det)`  
`do i = 1, size (a, dim = 1)`  
`det = det * lu(i,i)`  
`end do`  
`end subroutine determinant`

## H.3 Diagonalization

The code is an implementation of the algorithm presented in [16, 17], but independent from the code presented in [18] to avoid legal problems.

A Jacobi rotation around the angle  $\phi$  in row  $p$  and column  $q$

$$P(\phi; p, q) = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & \cos \phi & \cdots & \sin \phi \\ & & \vdots & 1 & \vdots \\ & & -\sin \phi & \cdots & \cos \phi \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix} \quad (\text{H.4})$$

results in

$$A' = P^T(\phi; p, q) \cdot A \cdot P(\phi; p, q) = \begin{pmatrix} & A'_{1p} & A'_{1q} & & \\ & \vdots & \vdots & & \\ A'_{p1} & \cdots & A'_{pq} & \cdots & A'_{pn} \\ & \vdots & \vdots & & \\ A'_{q1} & \cdots & A'_{qp} & \cdots & A'_{qn} \\ & \vdots & \vdots & & \\ & A'_{np} & A'_{nq} & & \end{pmatrix} \quad (\text{H.5})$$

**315a** *<Declaration of linalg procedures 311b>+≡*

`public :: diagonalize_real_symmetric`

**315b** *<Implementation of linalg procedures 312>+≡*

```
pure subroutine diagonalize_real_symmetric (a, eval, evec, num_rot)
 real(kind=default), dimension(:, :), intent(in) :: a
 real(kind=default), dimension(:), intent(out) :: eval
 real(kind=default), dimension(:, :), intent(out) :: evec
 integer, intent(out), optional :: num_rot
 real(kind=default), dimension(size(a,dim=1),size(a,dim=2)) :: aa
 real(kind=default) :: off_diagonal_norm, threshold, &
 c, g, h, s, t, tau, cot_2phi
 logical, dimension(size(eval),size(eval)) :: upper_triangle
 integer, dimension(size(eval)) :: one_to_ndim
 integer :: p, q, ndim, j, sweep
 integer, parameter :: MAX_SWEEPS = 50
 ndim = size (eval)
 one_to_ndim = (/ (j, j=1,ndim) /)
 upper_triangle = &
 spread (one_to_ndim, dim=1, ncopies=ndim) &
 > spread (one_to_ndim, dim=2, ncopies=ndim)
 aa = a
 call unit (evec)
<Initialize num_rot 318e>
 sweeps: do sweep = 1, MAX_SWEEPS
 off_diagonal_norm = sum (abs (aa), mask=upper_triangle)
 if (off_diagonal_norm == 0.0) then
 eval = diag (aa)
 return
 end if
 if (sweep < 4) then
 threshold = 0.2 * off_diagonal_norm / ndim**2
 else
```

```

 threshold = 0.0
 end if
 do p = 1, ndim - 1
 do q = p + 1, ndim
 ⟨Perform the Jacobi rotation resulting in $A'_{pq} = 0$ 316⟩
 end do
 end do
end do sweeps
if (present (num_rot)) then
 num_rot = -1
end if
!!! print *, "linalg::diagonalize_real_symmetric: exceeded sweep count"
end subroutine diagonalize_real_symmetric
316 ⟨Perform the Jacobi rotation resulting in $A'_{pq} = 0$ 316⟩≡
 g = 100 * abs (aa (p,q))
 if ((sweep > 4) &
 .and. (g <= min (spacing (aa(p,p)), spacing (aa(q,q)))) then
 aa(p,q) = 0.0
 else if (abs (aa(p,q)) > threshold) then
 ⟨Determine ϕ for the Jacobi rotation $P(\phi; p, q)$ with $A'_{pq} = 0$ 317a⟩
 ⟨ $A' = P^T(\phi; p, q) \cdot A \cdot P(\phi; p, q)$ 317c⟩
 ⟨ $V' = V \cdot P(\phi; p, q)$ 318d⟩
 ⟨Update num_rot 318f⟩
 end if

```

We want

$$A'_{pq} = (c^2 - s^2)A_{pq} + sc(A_{pp} - A_{qq}) = 0 \quad (\text{H.6})$$

and therefore

$$\cot 2\phi = \frac{1 - \tan^2 \phi}{2 \tan \phi} = \frac{\cos^2 \phi - \sin^2 \phi}{2 \sin \phi \cos \phi} = \frac{A_{pp} - A_{qq}}{2A_{pq}} \quad (\text{H.7})$$

i.e. with  $t = \tan \phi = s/c$

$$t^2 + 2t \cot 2\phi - 1 = 0 \quad (\text{H.8})$$

This quadratic equation has the roots

$$t = -\cot 2\phi \pm \sqrt{1 + \cot^2 2\phi} = \frac{\epsilon(\cot 2\phi)}{|\cot 2\phi| \pm \epsilon(\cot 2\phi)\sqrt{1 + \cot^2 2\phi}} \quad (\text{H.9})$$

and the smaller in magnitude of these is

$$t = \frac{\epsilon(\cot 2\phi)}{|\cot 2\phi| + \sqrt{1 + \cot^2 2\phi}} \quad (\text{H.10})$$

and since  $|t| \leq 1$ , it corresponds to  $|\phi| \leq \pi/4$ . For very large  $\cot 2\phi$  we will use

$$t = \frac{1}{2 \cot 2\phi} = \frac{A_{pq}}{A_{pp} - A_{qq}} \quad (\text{H.11})$$

$$h = A_{qq} - A_{pp} \quad (\text{H.12})$$

**317a**  $\langle \text{Determine } \phi \text{ for the Jacobi rotation } P(\phi; p, q) \text{ with } A'_{pq} = 0 \text{ } \mathbf{317a} \rangle \equiv$

```

h = aa(q,q) - aa(p,p)
if (g <= spacing (h)) then
 t = aa(p,q) / h
else
 cot_2phi = 0.5 * h / aa(p,q)
 t = sign (1.0_default, cot_2phi) &
 / (abs (cot_2phi) + sqrt (1.0 + cot_2phi**2))
end if

```

Trivia

$$\cos^2 \phi = \frac{\cos^2 \phi}{\cos^2 \phi + \sin^2 \phi} = \frac{1}{1 + \tan^2 \phi} \quad (\text{H.13a})$$

$$\sin \phi = \tan \phi \cos \phi \quad (\text{H.13b})$$

$$\tau \sin \phi = \frac{\sin^2}{1 + \cos \phi} = \frac{1 - \cos^2}{1 + \cos \phi} = 1 - \cos \phi \quad (\text{H.13c})$$

**317b**  $\langle \text{Determine } \phi \text{ for the Jacobi rotation } P(\phi; p, q) \text{ with } A'_{pq} = 0 \text{ } \mathbf{317a} \rangle + \equiv$

```

c = 1.0 / sqrt (1.0 + t**2)
s = t * c
tau = s / (1.0 + c)

```

$$\begin{aligned}
A'_{pp} &= c^2 A_{pp} + s^2 A_{qq} - 2sc A_{pq} = A_{pp} - t A_{pq} \\
A'_{qq} &= s^2 A_{pp} + c^2 A_{qq} + 2sc A_{pq} = A_{qq} + t A_{pq} \\
A'_{pq} &= (c^2 - s^2) A_{pq} + sc(A_{pp} - A_{qq})
\end{aligned} \quad (\text{H.14})$$

**317c**  $\langle A' = P^T(\phi; p, q) \cdot A \cdot P(\phi; p, q) \text{ } \mathbf{317c} \rangle \equiv$

```

aa(p,p) = aa(p,p) - t * aa(p,q)
aa(q,q) = aa(q,q) + t * aa(p,q)
aa(p,q) = 0.0

```

$$\begin{aligned}
r \neq p < q \neq r : A'_{rp} &= c A_{rp} - s A_{rq} \\
A'_{rq} &= s A_{rp} + c A_{rq}
\end{aligned} \quad (\text{H.15})$$

Here's how we cover the upper triangular region using array notation:

$$\begin{pmatrix} & a(1:p-1,p) & & a(1:p-1,q) & \\ \cdots & A_{pq} & a(p,p+1:q-1) & A_{pq} & a(p,q+1:ndim) \\ & \vdots & & a(p+1:q-1,q) & \\ \cdots & A_{qp} & \cdots & A_{qq} & a(q,q+1:ndim) \\ & \vdots & & \vdots & \end{pmatrix} \quad (\text{H.16})$$

**318a**  $\langle A' = P^T(\phi; p, q) \cdot A \cdot P(\phi; p, q)$  **317c**  $\rangle + \equiv$   
`call jacobi_rotation (s, tau, aa(1:p-1,p), aa(1:p-1,q))`  
`call jacobi_rotation (s, tau, aa(p,p+1:q-1), aa(p+1:q-1,q))`  
`call jacobi_rotation (s, tau, aa(p,q+1:ndim), aa(q,q+1:ndim))`

Using (H.13c), we can write the rotation as a perturbation:

$$\begin{aligned} V'_p &= cV_p - sV_q = V_p - s(V_q + \tau V_p) \\ V'_q &= sV_p + cV_q = V_q + s(V_p - \tau V_q) \end{aligned} \quad (\text{H.17})$$

**318b**  $\langle$ Implementation of linalg procedures **312** $\rangle + \equiv$   
`pure subroutine jacobi_rotation (s, tau, vp, vq)`  
`real(kind=default), intent(in) :: s, tau`  
`real(kind=default), dimension(:), intent(inout) :: vp, vq`  
`real(kind=default), dimension(size(vp)) :: vp_tmp`  
`vp_tmp = vp`  
`vp = vp - s * (vq + tau * vp)`  
`vq = vq + s * (vp_tmp - tau * vq)`  
`end subroutine jacobi_rotation`

**318c**  $\langle$ Declaration of linalg procedures **311b** $\rangle + \equiv$   
`private :: jacobi_rotation`

**318d**  $\langle V' = V \cdot P(\phi; p, q)$  **318d**  $\rangle \equiv$   
`call jacobi_rotation (s, tau, evec(:,p), evec(:,q))`

**318e**  $\langle$ Initialize num\_rot **318e** $\rangle \equiv$   
`if (present (num_rot)) then`  
`num_rot = 0`  
`end if`

**318f**  $\langle$ Update num\_rot **318f** $\rangle \equiv$   
`if (present (num_rot)) then`  
`num_rot = num_rot + 1`  
`end if`

319a *<Implementation of linalg procedures 312>+≡*

```
pure subroutine unit (u)
 real(kind=default), dimension(:,,:), intent(out) :: u
 integer :: i
 u = 0.0
 do i = 1, min (size (u, dim = 1), size (u, dim = 2))
 u(i,i) = 1.0
 end do
end subroutine unit
```

319b *<Implementation of linalg procedures 312>+≡*

```
pure function diag (a) result (d)
 real(kind=default), dimension(:,,:), intent(in) :: a
 real(kind=default), dimension(min(size(a,dim=1),size(a,dim=2))) :: d
 integer :: i
 do i = 1, min (size (a, dim = 1), size (a, dim = 2))
 d(i) = a(i,i)
 end do
end function diag
```

319c *<Declaration of linalg procedures 311b>+≡*

```
public :: unit, diag
```

## H.4 Test

319d *<la\_sample.f90 319d>≡*

```
! la_sample.f90 --
<Copyleft notice 1>
program la_sample
 use kinds
 use utils
 use tao_random_numbers
 use linalg
 implicit none
 integer, parameter :: N = 200
 real(kind=default), dimension(N,N) :: a, evec, a0, l, u, NAG_bug
 real(kind=default), dimension(N) :: b, eval
 real(kind=default) :: d
 integer :: i
 call system_clock (i)
 call tao_random_seed (i)
 print *, i
 do i = 1, N
```



```

 call tao_random_number (a(:,i))
 end do
 NAG_bug = (a + transpose (a)) / 2
 a = NAG_bug
 a0 = a
 call lu_decompose (a, l=l, u=u)
 a = matmul (l, u)
 print *, maxval (abs(a-a0))
 call determinant (a, d)
 print *, d
 call diagonalize_real_symmetric (a, eval, evec)
 print *, product (eval)
 stop
 call sort (eval, evec)
 do i = 1, N
 b = matmul (a, evec(:,i)) - eval(i) * evec(:,i)
 write (unit = *, fmt = "(A,I3, 2(A,E11.4))") &
 "eval #", i, " = ", eval(i), ", |(A-lambda)V|_infty = ", &
 maxval (abs(b)) / maxval (abs(evec(:,i)))
 end do
end program la_sample

```

# —I—

## PRODUCTS

```

321 <products.f90 321>≡
 ! products.f90 --
 <Copyleft notice 1>
 module products
 use kinds
 implicit none
 private
 public :: dot, sp, spc
 character(len=*), public, parameter :: PRODUCTS_RCS_ID = &
 "$Id: products.nw 314 2010-04-17 20:32:33Z ohl $"
 contains
 pure function dot (p, q) result (pq)
 real(kind=default), dimension(0:), intent(in) :: p, q
 real(kind=default) :: pq
 pq = p(0)*q(0) - dot_product (p(1:), q(1:))
 end function dot
 pure function sp (p, q) result (sppq)
 real(kind=default), dimension(0:), intent(in) :: p, q
 complex(kind=default) :: sppq
 sppq = cmplx (p(2), p(3)) * sqrt ((q(0)-q(1))/(p(0)-p(1))) &
 - cmplx (q(2), q(3)) * sqrt ((p(0)-p(1))/(q(0)-q(1)))
 end function sp
 pure function spc (p, q) result (spcpq)
 real(kind=default), dimension(0:), intent(in) :: p, q
 complex(kind=default) :: spcpq
 spcpq = conjg (sp (p, q))
 end function spc
 end module products

```

# —J—

## KINEMATICS

```

322a <kinematics.f90 322a>≡
! kinematics.f90 --
<Copyleft notice 1>
module kinematics
 use kinds
 use constants
 use products, only: dot
 use specfun, only: gamma
 implicit none
 private
 <Declaration of kinematics procedures 322b>
 <Interfaces of kinematics procedures 322c>
 <Declaration of kinematics types 324g>
 character(len=*), public, parameter :: KINEMATICS_RCS_ID = &
 "$Id: kinematics.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of kinematics procedures 323a>
end module kinematics

```

### *J.1 Lorentz Transformations*

```

322b <Declaration of kinematics procedures 322b>≡
 public :: boost_velocity
 private :: boost_one_velocity, boost_many_velocity
 public :: boost_momentum
 private :: boost_one_momentum, boost_many_momentum

322c <Interfaces of kinematics procedures 322c>≡
 interface boost_velocity
 module procedure boost_one_velocity, boost_many_velocity

```

```

end interface
interface boost_momentum
 module procedure boost_one_momentum, boost_many_momentum
end interface

```

Boost a four vector  $p$  to the inertial frame moving with the velocity  $\beta$ :

$$p'_0 = \gamma (p_0 - \vec{\beta} \vec{p}) \quad (\text{J.1a})$$

$$\vec{p}' = \gamma (\vec{p}_{\parallel} - \vec{\beta} p_0) + \vec{p}_{\perp} \quad (\text{J.1b})$$

with  $\gamma = 1/\sqrt{1 - \vec{\beta}^2}$ ,  $\vec{p}_{\parallel} = \vec{\beta}(\vec{\beta} \vec{p})/\vec{\beta}^2$  and  $\vec{p}_{\perp} = \vec{p} - \vec{p}_{\parallel}$ . Using  $1/\vec{\beta}^2 = \gamma^2/(\gamma + 1) \cdot 1/(\gamma - 1)$  and  $\vec{b} = \gamma \vec{\beta}$  this can be rewritten as

$$p'_0 = \gamma p_0 - \vec{b} \vec{p} \quad (\text{J.2a})$$

$$\vec{p}' = \vec{p} + \left( \frac{\vec{b} \vec{p}}{\gamma + 1} - p_0 \right) \vec{b} \quad (\text{J.2b})$$

**323a**  $\langle \text{Implementation of kinematics procedures } \text{323a} \rangle \equiv$

```

pure function boost_one_velocity (p, beta) result (p_prime)
 real(kind=default), dimension(0:), intent(in) :: p
 real(kind=default), dimension(1:), intent(in) :: beta
 real(kind=default), dimension(0:3) :: p_prime
 real(kind=default), dimension(1:3) :: b
 real(kind=default) :: gamma, b_dot_p
 gamma = 1.0 / sqrt (1.0 - dot_product (beta, beta))
 b = gamma * beta
 b_dot_p = dot_product (b, p(1:3))
 p_prime(0) = gamma * p(0) - b_dot_p
 p_prime(1:3) = p(1:3) + (b_dot_p / (1.0 + gamma) - p(0)) * b
end function boost_one_velocity

```

**323b**  $\langle \text{Implementation of kinematics procedures } \text{323a} \rangle + \equiv$

```

pure function boost_many_velocity (p, beta) result (p_prime)
 real(kind=default), dimension(:,0:), intent(in) :: p
 real(kind=default), dimension(1:), intent(in) :: beta
 real(kind=default), dimension(size(p,dim=1),0:3) :: p_prime
 integer :: i
 do i = 1, size (p, dim=1)
 p_prime(i,:) = boost_one_velocity (p(i,:), beta)
 end do
end function boost_many_velocity

```

Boost a four vector  $p$  to the rest frame of the four vector  $q$ . The velocity is  $\vec{\beta} = \vec{q}/|q_0|$ :

```

324a <Implementation of kinematics procedures 323a>+≡
 pure function boost_one_momentum (p, q) result (p_prime)
 real(kind=default), dimension(0:), intent(in) :: p, q
 real(kind=default), dimension(0:3) :: p_prime
 p_prime = boost_velocity (p, q(1:3) / abs (q(0)))
 end function boost_one_momentum

324b <Implementation of kinematics procedures 323a>+≡
 pure function boost_many_momentum (p, q) result (p_prime)
 real(kind=default), dimension(:,0:), intent(in) :: p
 real(kind=default), dimension(0:), intent(in) :: q
 real(kind=default), dimension(size(p,dim=1),0:3) :: p_prime
 p_prime = boost_many_velocity (p, q(1:3) / abs (q(0)))
 end function boost_many_momentum

```

## *J.2 Massive Phase Space*

$$\lambda(a, b, c) = a^2 + b^2 + c^2 - 2ab - 2bc - 2ca = (a - b - c)^2 - 4bc \quad (\text{J.3})$$

and permutations

```

324c <Implementation of kinematics procedures 323a>+≡
 pure function lambda (a, b, c) result (lam)
 real(kind=default), intent(in) :: a, b, c
 real(kind=default) :: lam
 lam = a**2 + b**2 + c**2 - 2*(a*b + b*c + c*a)
 end function lambda

324d <Declaration of kinematics procedures 322b>+≡
 public :: lambda

324e <Declaration of kinematics procedures 322b>+≡
 public :: two_to_three
 private :: two_to_three_massive, two_to_three_massless

324f <Interfaces of kinematics procedures 322c>+≡
 interface two_to_three
 module procedure two_to_three_massive, two_to_three_massless
 end interface

324g <Declaration of kinematics types 324g>≡
 type, public :: LIPS3
 real(kind=default), dimension(3,0:3) :: p
 real(kind=default) :: jacobian
 end type LIPS3

```

$$dLIPS_3 = \int \frac{d^3\vec{p}_1}{(2\pi)^3 2E_1} \frac{d^3\vec{p}_2}{(2\pi)^3 2E_2} \frac{d^3\vec{p}_3}{(2\pi)^3 2E_3} (2\pi)^4 \delta^4(p_1 + p_2 + p_3 - p_a - p_b) \quad (J.4)$$

The jacobian is given by

$$dLIPS_3 = \frac{1}{(2\pi)^5} \int d\phi dt_1 ds_2 d\Omega_3^{[23]} \frac{1}{32\sqrt{ss_2}} \frac{|p_3^{[23]}|}{|p_a^{[ab]}|} \quad (J.5)$$

where  $\vec{p}_i^{[jk]}$  denotes the momentum of particle  $i$  in the center of mass system of particles  $j$  and  $k$ .

**325a** *⟨Implementation of kinematics procedures 323a⟩* +≡  

```

pure function two_to_three_massive &
 (s, t1, s2, phi, cos_theta3, phi3, ma, mb, m1, m2, m3) result (p)
 real(kind=default), intent(in) :: &
 s, t1, s2, phi, cos_theta3, phi3, ma, mb, m1, m2, m3
 type(LIPS3) :: p
 real(kind=default), dimension(0:3) :: p23
 real(kind=default) :: Ea, pa_abs, E1, p1_abs, p3_abs, cos_theta
 pa_abs = sqrt (lambda (s, ma**2, mb**2) / (4 * s))
 Ea = sqrt (ma**2 + pa_abs**2)
 p1_abs = sqrt (lambda (s, m1**2, s2) / (4 * s))
 E1 = sqrt (m1**2 + p1_abs**2)
 p3_abs = sqrt (lambda (s2, m2**2, m3**2) / (4 * s2))
 p%jacobian = &
 1.0 / (2*PI)**5 * (p3_abs / pa_abs) / (32 * sqrt (s * s2))
 cos_theta = (t1 - ma**2 - m1**2 + 2*Ea*E1) / (2*pa_abs*p1_abs)
 p%p(1,1:3) = polar_to_cartesian (p1_abs, cos_theta, phi)
 p%p(1,0) = on_shell (p%p(1,:), m1)
 p23(1:3) = - p%p(1,1:3)
 p23(0) = on_shell (p23, sqrt (s2))
 p%p(3:2:-1,:) = one_to_two (p23, cos_theta3, phi3, m3, m2)
end function two_to_three_massive

```

A specialized version for massless particles can be faster, because the kinematics is simpler:

**325b** *⟨Implementation of kinematics procedures 323a⟩* +≡  

```

pure function two_to_three_massless (s, t1, s2, phi, cos_theta3, phi3) &
 result (p)
 real(kind=default), intent(in) :: s, t1, s2, phi, cos_theta3, phi3
 type(LIPS3) :: p
 real(kind=default), dimension(0:3) :: p23
 real(kind=default) :: pa_abs, p1_abs, p3_abs, cos_theta
 pa_abs = sqrt (s) / 2
 p1_abs = (s - s2) / (2 * sqrt (s))

```

```

 p3_abs = sqrt (s2) / 2
 p%jacobian = 1.0 / ((2*PI)**5 * 32 * s)
 cos_theta = 1 + t1 / (2*pa_abs*p1_abs)
 p%p(1,0) = p1_abs
 p%p(1,1:3) = polar_to_cartesian (p1_abs, cos_theta, phi)
 p23(1:3) = - p%p(1,1:3)
 p23(0) = on_shell (p23, sqrt (s2))
 p%p(3:2:-1,:) = one_to_two (p23, cos_theta3, phi3)
end function two_to_three_massless

326a <Declaration of kinematics procedures 322b>+≡
 public :: one_to_two
 private :: one_to_two_massive, one_to_two_massless

326b <Interfaces of kinematics procedures 322c>+≡
 interface one_to_two
 module procedure one_to_two_massive, one_to_two_massless
 end interface

326c <Implementation of kinematics procedures 323a>+≡
 pure function one_to_two_massive (p12, cos_theta, phi, m1, m2) result (p)
 real(kind=default), dimension(0:), intent(in) :: p12
 real(kind=default), intent(in) :: cos_theta, phi, m1, m2
 real(kind=default), dimension(2,0:3) :: p
 real(kind=default) :: s, p1_abs
 s = dot (p12, p12)
 p1_abs = sqrt (lambda (s, m1**2, m2**2) / (4 * s))
 p(1,1:3) = polar_to_cartesian (p1_abs, cos_theta, phi)
 p(2,1:3) = - p(1,1:3)
 p(1,0) = on_shell (p(1,:), m1)
 p(2,0) = on_shell (p(2,:), m2)
 p = boost_momentum (p, - p12)
 end function one_to_two_massive

326d <Implementation of kinematics procedures 323a>+≡
 pure function one_to_two_massless (p12, cos_theta, phi) result (p)
 real(kind=default), dimension(0:), intent(in) :: p12
 real(kind=default), intent(in) :: cos_theta, phi
 real(kind=default), dimension(2,0:3) :: p
 real(kind=default) :: p1_abs
 p1_abs = sqrt (dot (p12, p12)) / 2
 p(1,0) = p1_abs
 p(1,1:3) = polar_to_cartesian (p1_abs, cos_theta, phi)
 p(2,0) = p1_abs
 p(2,1:3) = - p(1,1:3)
 p = boost_momentum (p, - p12)

```

```

end function one_to_two_massless

327a <Declaration of kinematics procedures 322b>+≡
public :: polar_to_cartesian, on_shell

327b <Implementation of kinematics procedures 323a>+≡
pure function polar_to_cartesian (v_abs, cos_theta, phi) result (v)
 real(kind=default), intent(in) :: v_abs, cos_theta, phi
 real(kind=default), dimension(3) :: v
 real(kind=default) :: sin_phi, cos_phi, sin_theta
 sin_theta = sqrt (1.0 - cos_theta**2)
 cos_phi = cos (phi)
 sin_phi = sin (phi)
 v = (/ sin_theta * cos_phi, sin_theta * sin_phi, cos_theta /) * v_abs
end function polar_to_cartesian

327c <Implementation of kinematics procedures 323a>+≡
pure function on_shell (p, m) result (E)
 real(kind=default), dimension(0:), intent(in) :: p
 real(kind=default), intent(in) :: m
 real(kind=default) :: E
 E = sqrt (m**2 + dot_product (p(1:3), p(1:3)))
end function on_shell

```

### *J.3 Massive 3-Particle Phase Space Revisited*

$$\begin{array}{ccccc}
U_1 & \xrightarrow{\xi_1} & P_1 & \xrightarrow{\phi_1} & M \\
\pi_U \downarrow & & \downarrow \pi_P & & \parallel \\
U_2 & \xrightarrow{\xi_2} & P_2 & \xrightarrow{\phi_2} & M
\end{array} \tag{J.6}$$

$$\begin{array}{ccccc}
U_1 & \xrightarrow{\xi} & P_1 & \xrightarrow{\phi} & M \\
\pi_U \downarrow & & \downarrow \pi_P & & \downarrow \pi \\
U_2 & \xrightarrow{\xi} & P_2 & \xrightarrow{\phi} & M
\end{array} \tag{J.7}$$

```

327d <kinematics.f90 322a>+≡
module phase_space
 use kinds
 use constants
 use kinematics !NODEP!
 use tao_random_numbers
 implicit none
 private

```



```

 <Declaration of phase_space procedures 329b>
 <Interfaces of phase_space procedures 329c>
 <Declaration of phase_space types 328a>
 character(len=*), public, parameter :: PHASE_SPACE_RCS_ID = &
 "$Id: kinematics.nw 314 2010-04-17 20:32:33Z ohl $"
contains
 <Implementation of phase_space procedures 329d>
end module phase_space

LIPS3_unit : [0, 1]5 (J.8)

328a <Declaration of phase_space types 328a>≡
type, public :: LIPS3_unit
 real(kind=default), dimension(5) :: x
 real(kind=default) :: s
 real(kind=default), dimension(2) :: mass_in
 real(kind=default), dimension(3) :: mass_out
 real(kind=default) :: jacobian
end type LIPS3_unit

328b <Declaration of phase_space types 328a>+≡
type, public :: LIPS3_unit_massless
 real(kind=default), dimension(5) :: x
 real(kind=default) :: s
 real(kind=default) :: jacobian
end type LIPS3_unit_massless

LIPS3_s2_t1_angles : (s2, t1, φ, cos θ3, φ3) (J.9)

328c <Declaration of phase_space types 328a>+≡
type, public :: LIPS3_s2_t1_angles
 real(kind=default) :: s2, t1, phi, cos_theta3, phi3
 real(kind=default) :: s
 real(kind=default), dimension(2) :: mass_in
 real(kind=default), dimension(3) :: mass_out
 real(kind=default) :: jacobian
end type LIPS3_s2_t1_angles

328d <Declaration of phase_space types 328a>+≡
type, public :: LIPS3_s2_t1_angles_massless
 real(kind=default) :: s2, t1, phi, cos_theta3, phi3
 real(kind=default) :: s
 real(kind=default) :: jacobian
end type LIPS3_s2_t1_angles_massless

LIPS3_momenta : (p1, p2, p3) (J.10)

328e <Declaration of phase_space types 328a>+≡

```

```

type, public :: LIPS3_momenta
 real(kind=default), dimension(0:3,3) :: p
 real(kind=default) :: s
 real(kind=default), dimension(2) :: mass_in
 real(kind=default), dimension(3) :: mass_out
 real(kind=default) :: jacobian
end type LIPS3_momenta

329a <Declaration of phase_space types 328a>+≡
type, public :: LIPS3_momenta_massless
 real(kind=default), dimension(0:3,3) :: p
 real(kind=default) :: s
 real(kind=default) :: jacobian
end type LIPS3_momenta_massless

329b <Declaration of phase_space procedures 329b>≡
public :: random_LIPS3
private :: random_LIPS3_unit, random_LIPS3_unit_massless

329c <Interfaces of phase_space procedures 329c>≡
interface random_LIPS3
 module procedure random_LIPS3_unit, random_LIPS3_unit_massless
end interface

329d <Implementation of phase_space procedures 329d>≡
pure subroutine random_LIPS3_unit (rng, lips)
 type(tao_random_state), intent(inout) :: rng
 type(LIPS3_unit), intent(inout) :: lips
 call tao_random_number (rng, lips%x)
 lips%jacobian = 1
end subroutine random_LIPS3_unit

329e <Implementation of phase_space procedures 329d>+≡
pure subroutine random_LIPS3_unit_massless (rng, lips)
 type(tao_random_state), intent(inout) :: rng
 type(LIPS3_unit_massless), intent(inout) :: lips
 call tao_random_number (rng, lips%x)
 lips%jacobian = 1
end subroutine random_LIPS3_unit_massless

329f <Declaration of phase_space procedures 329b>+≡
private :: LIPS3_unit_to_s2_t1_angles, LIPS3_unit_to_s2_t1_angles_m0

329g <(Unused) Interfaces of phase_space procedures 329g>≡
interface assignment(=)
 module procedure &
 LIPS3_unit_to_s2_t1_angles, LIPS3_unit_to_s2_t1_angles_m0
end interface

```

330a *⟨Implementation of phase\_space procedures 329d⟩*+≡  
 pure subroutine LIPS3\_unit\_to\_s2\_t1\_angles (s2\_t1\_angles, unit)  
   type(LIPS3\_s2\_t1\_angles), intent(out) :: s2\_t1\_angles  
   type(LIPS3\_unit), intent(in) :: unit  
 end subroutine LIPS3\_unit\_to\_s2\_t1\_angles

330b *⟨Implementation of phase\_space procedures 329d⟩*+≡  
 pure subroutine LIPS3\_unit\_to\_s2\_t1\_angles\_m0 (s2\_t1\_angles, unit)  
   type(LIPS3\_s2\_t1\_angles\_massless), intent(out) :: s2\_t1\_angles  
   type(LIPS3\_unit\_massless), intent(in) :: unit  
 end subroutine LIPS3\_unit\_to\_s2\_t1\_angles\_m0

## *J.4 Massless n-Particle Phase Space: RAMBO*

330c *⟨Declaration of kinematics procedures 322b⟩*+≡  
 public :: massless\_isotropic\_decay

The massless RAMBO algorithm [25]:

330d *⟨Implementation of kinematics procedures 323a⟩*+≡  
 pure function massless\_isotropic\_decay (roots, ran) result (p)  
   real (kind=default), intent(in) :: roots  
   real (kind=default), dimension(:,,:), intent(in) :: ran  
   real (kind=default), dimension(size(ran,dim=1),0:3) :: p  
   real (kind=default), dimension(size(ran,dim=1),0:3) :: q  
   real (kind=default), dimension(0:3) :: qsum  
   real (kind=default) :: cos\_theta, sin\_theta, phi, qabs, x, r, z  
   integer :: k  
   *⟨Generate isotropic null vectors 330e⟩*  
   *⟨Boost and rescale the vectors 331a⟩*  
 end function massless\_isotropic\_decay

Generate a  $xe^{-x}$  distribution for  $q(k,0)$

330e *⟨Generate isotropic null vectors 330e⟩*≡  
 do k = 1, size (p, dim = 1)  
   q(k,0) = - log (ran(k,1) \* ran(k,2))  
   cos\_theta = 2 \* ran(k,3) - 1  
   sin\_theta = sqrt (1 - cos\_theta\*\*2)  
   phi = 2 \* PI \* ran(k,4)  
   q(k,1) = q(k,0) \* sin\_theta \* cos (phi)  
   q(k,2) = q(k,0) \* sin\_theta \* sin (phi)  
   q(k,3) = q(k,0) \* cos\_theta  
 enddo

The proof that the Jacobian of the transformation vanishes can be found in [25]. The transformation is really a Lorentz boost (as can be seen easily).

331a  $\langle$ Boost and rescale the vectors 331a $\rangle \equiv$

```
qsum = sum (q, dim = 1)
qabs = sqrt (dot (qsum, qsum))
x = roots / qabs
do k = 1, size (p, dim = 1)
 r = dot (q(k,:), qsum) / qabs
 z = (q(k,0) + r) / (qsum(0) + qabs)
 p(k,1:3) = x * (q(k,1:3) - qsum(1:3) * z)
 p(k,0) = x * r
enddo
```

331b  $\langle$ Declaration of kinematics procedures 322b $\rangle + \equiv$

```
public :: phase_space_volume
```

$$V_n(s) = \frac{1}{8\pi} \frac{n-1}{(\Gamma(n))^2} \left( \frac{s}{16\pi^2} \right)^{n-2} \quad (\text{J.11})$$

331c  $\langle$ Implementation of kinematics procedures 323a $\rangle + \equiv$

```
pure function phase_space_volume (n, roots) result (volume)
 integer, intent(in) :: n
 real (kind=default), intent(in) :: roots
 real (kind=default) :: volume
 real (kind=default) :: nd
 nd = n
 volume = (nd - 1) / (8*PI * (gamma (nd))**2) * (roots / (4*PI))**(2*n-4)
end function phase_space_volume
```

## J.5 Tests

331d  $\langle$ ktest.f90 331d $\rangle \equiv$

```
program ktest
 use kinds
 use constants
 use products
 use kinematics
 use tao_random_numbers
 implicit none
 real(kind=default) :: &
 ma, mb, m1, m2, m3, s, t1, s2, phi, cos_theta3, phi3
 real(kind=default) :: t1_min, t1_max
 real(kind=default), dimension(5) :: r
 type(LIPS3) :: p
```

```

integer :: i
character(len=*), parameter :: fmt = "(A,4(1X,E12.5))"
ma = 1.0
mb = 1.0
m1 = 10.0
m2 = 20.0
m3 = 30.0
s = 100.0 ** 2
do i = 1, 10
 call tao_random_number (r)
 s2 = (r(1) * (sqrt (s) - m1) + (1 - r(1)) * (m2 + m3)) ** 2
 t1_max = ma**2 + m1**2 - ((s + ma**2 - mb**2) * (s - s2 + m1**2) &
 + sqrt (lambda (s, ma**2, mb**2) * lambda (s, s2, m1**2))) / (2*s)
 t1_min = ma**2 + m1**2 - ((s + ma**2 - mb**2) * (s - s2 + m1**2) &
 - sqrt (lambda (s, ma**2, mb**2) * lambda (s, s2, m1**2))) / (2*s)
 t1 = r(2) * t1_max + (1 - r(2)) * t1_min
 phi = 2*PI * r(3)
 cos_theta3 = 2 * r(4) - 1
 phi3 = 2*PI * r(5)
 p = two_to_three (s, t1, s2, phi, cos_theta3, phi3, ma, mb, m1, m2, m3)
 print fmt, "p1 = ", p%p(1,:)
 print fmt, "p2 = ", p%p(2,:)
 print fmt, "p3 = ", p%p(3,:)
 print fmt, "p1,2,3^2 = ", dot (p%p(1,:), p%p(1,)), &
 dot (p%p(2,:), p%p(2,)), dot (p%p(3,:), p%p(3,))
 print fmt, "sum(p) = ", p%p(1,:) + p%p(2,:) + p%p(3,:)
 print fmt, "|J| = ", p%jacobian
end do
end program ktest

```



Trivial check for typos, should be removed from the finalized program!

332  $\langle$ Trivial ktest.f90 332 $\rangle \equiv$

```

program ktest
 use kinds
 use constants
 use products
 use kinematics
 use tao_random_numbers
 implicit none
 real(kind=default), dimension(0:3) :: p, q, p_prime, p0
 real(kind=default) :: m
 character(len=*), parameter :: fmt = "(A,4(1X,E12.5))"
 integer :: i

```

```

do i = 1, 5
 if (i == 1) then
 p = (/ 1.0_double, 0.0_double, 0.0_double, 0.0_double /)
 m = 1.0
 else
 call tao_random_number (p)
 m = sqrt (PI)
 end if
 call tao_random_number (q(1:3))
 q(0) = sqrt (m**2 + dot_product (q(1:3), q(1:3)))
 p_prime = boost_momentum (p, q)
 print fmt, "p = ", p
 print fmt, "q = ", q
 print fmt, "p' = ", p_prime
 print fmt, "p^2 = ", dot (p, p)
 print fmt, "p'^2 = ", dot (p_prime, p_prime)
 if (dot (p, p) > 0.0) then
 p0 = boost_momentum (p, p)
 print fmt, "p0 = ", p0
 print fmt, "p0^2 = ", dot (p0, p0)
 end if
end do
end program ktest

```

# —K—

## COORDINATES

```

334 <coordinates.f90 334>≡
 ! coordinates.f90 --
 <Copyleft notice 1>
 module coordinates
 use kinds
 use constants, only: PI
 use specfun, only: gamma
 implicit none
 private
 <Declaration of coordinates procedures 335a>
 contains
 <Implementation of coordinates procedures 335b>
 end module coordinates

```

### *K.1 Angular Spherical Coordinates*

$$\begin{aligned}
 x_n &= r \cos \theta_{n-2} \\
 x_{n-1} &= r \sin \theta_{n-2} \cos \theta_{n-3} \\
 &\dots \\
 x_3 &= r \sin \theta_{n-2} \sin \theta_{n-3} \cdots \cos \theta_1 \\
 x_2 &= r \sin \theta_{n-2} \sin \theta_{n-3} \cdots \sin \theta_1 \cos \phi \\
 x_1 &= r \sin \theta_{n-2} \sin \theta_{n-3} \cdots \sin \theta_1 \sin \phi
 \end{aligned} \tag{K.1}$$

and

$$J = r^{n-1} \prod_{i=1}^{n-2} (\sin \theta_i)^i \tag{K.2}$$

We can minimize the number of multiplications by computing the products

$$P_j = \prod_{i=j}^{n-2} \sin \theta_i \quad (\text{K.3})$$

Then

$$\begin{aligned} x_n &= r \cos \theta_{n-2} \\ x_{n-1} &= r P_{n-2} \cos \theta_{n-3} \\ &\dots \\ x_3 &= r P_2 \cos \theta_1 \\ x_2 &= r P_1 \cos \phi \\ x_1 &= r P_1 \sin \phi \end{aligned} \quad (\text{K.4})$$

and

$$J = r^{n-1} \prod_{i=1}^{n-2} P_i \quad (\text{K.5})$$

Note that  $\theta_i \in [0, \pi]$  and  $\phi \in [0, 2\pi]$  or  $\phi \in [-\pi, \pi]$ . Therefore  $\sin \theta_i \geq 0$  and

$$\sin \theta_i = \sqrt{1 - \cos^2 \theta_i} \quad (\text{K.6})$$

which is not true for  $\phi$ . Since `sqrt` is typically much faster than `sin` and `cos`, we use (K.6) where ever possible.

```

335a <Declaration of coordinates procedures 335a>≡
 public :: spherical_to_cartesian_2, &
 spherical_to_cartesian, spherical_to_cartesian_j
335b <Implementation of coordinates procedures 335b>≡
 pure subroutine spherical_to_cartesian_2 (r, phi, theta, x, jacobian)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: theta
 real(kind=default), dimension(:), intent(out), optional :: x
 real(kind=default), intent(out), optional :: jacobian
 real(kind=default), dimension(size(theta)) :: cos_theta
 real(kind=default), dimension(size(theta)+1) :: product_sin_theta
 integer :: n, i
 n = size (theta) + 2
 cos_theta = cos (theta)
 product_sin_theta(n-1) = 1.0_default
 do i = n - 2, 1, -1
 product_sin_theta(i) = &
 product_sin_theta(i+1) * sqrt (1 - cos_theta(i)**2)
 end do

```



```

if (present (x)) then
 x(1) = r * product_sin_theta(1) * sin (phi)
 x(2) = r * product_sin_theta(1) * cos (phi)
 x(3:) = r * product_sin_theta(2:n-1) * cos_theta
end if
if (present (jacobian)) then
 jacobian = r**(n-1) * product (product_sin_theta)
end if
end subroutine spherical_to_cartesian_2

```



Note that `call` inside of a function breaks F-compatibility. Here it would be easy to fix, but the inverse can not be coded as a function, unless a type for spherical coordinates is introduced, where `theta` could not be assumed shape ...

**336a** *⟨Implementation of coordinates procedures 335b⟩*+≡

```

pure function spherical_to_cartesian (r, phi, theta) result (x)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: theta
 real(kind=default), dimension(size(theta)+2) :: x
 call spherical_to_cartesian_2 (r, phi, theta, x = x)
end function spherical_to_cartesian

```

**336b** *⟨Implementation of coordinates procedures 335b⟩*+≡

```

pure function spherical_to_cartesian_j (r, phi, theta) &
 result (jacobian)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: theta
 real(kind=default) :: jacobian
 call spherical_to_cartesian_2 (r, phi, theta, jacobian = jacobian)
end function spherical_to_cartesian_j

```

**336c** *⟨Declaration of coordinates procedures 335a⟩*+≡

```

public :: cartesian_to_spherical_2, &
 cartesian_to_spherical, cartesian_to_spherical_j

```

**336d** *⟨Implementation of coordinates procedures 335b⟩*+≡

```

pure subroutine cartesian_to_spherical_2 (x, r, phi, theta, jacobian)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), intent(out), optional :: r, phi
 real(kind=default), dimension(:), intent(out), optional :: theta
 real(kind=default), intent(out), optional :: jacobian
 real(kind=default) :: local_r
 real(kind=default), dimension(size(x)-2) :: cos_theta
 real(kind=default), dimension(size(x)-1) :: product_sin_theta

```

```

integer :: n, i
n = size (x)
local_r = sqrt (dot_product (x, x))
product_sin_theta(n-1) = 1
do i = n, 3, -1
 cos_theta(i-2) = x(i) / product_sin_theta(i-1) / local_r
 product_sin_theta(i-2) = &
 product_sin_theta(i-1) * sqrt (1 - cos_theta(i-2)**2)
end do
if (present (r)) then
 r = local_r
end if
if (present (phi)) then
 phi = atan2 (x(1), x(2))
end if
if (present (theta)) then
 theta = acos (cos_theta)
end if
if (present (jacobian)) then
 jacobian = local_r**(1-n) / product (product_sin_theta)
end if
end subroutine cartesian_to_spherical_2

```

**337a** *⟨Implementation of coordinates procedures 335b⟩+≡*

```

pure subroutine cartesian_to_spherical (x, r, phi, theta)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), intent(out) :: r, phi
 real(kind=default), dimension(:), intent(out) :: theta
 call cartesian_to_spherical_2 (x, r, phi, theta)
end subroutine cartesian_to_spherical

```

**337b** *⟨Implementation of coordinates procedures 335b⟩+≡*

```

pure function cartesian_to_spherical_j (x) result (jacobian)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default) :: jacobian
 call cartesian_to_spherical_2 (x, jacobian = jacobian)
end function cartesian_to_spherical_j

```

## *K.2 Trigonometric Spherical Coordinates*

**337c** *⟨Declaration of coordinates procedures 335a⟩+≡*

```

public :: spherical_cos_to_cartesian_2, &
 spherical_cos_to_cartesian, spherical_cos_to_cartesian_j

```

Using the cosine, we have to drop  $P_1$  from the Jacobian

338a *<Implementation of coordinates procedures 335b>+≡*

```
pure subroutine spherical_cos_to_cartesian_2 (r, phi, cos_theta, x, jacobian)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: cos_theta
 real(kind=default), dimension(:), intent(out), optional :: x
 real(kind=default), intent(out), optional :: jacobian
 real(kind=default), dimension(size(cos_theta)+1) :: product_sin_theta
 integer :: n, i
 n = size (cos_theta) + 2
 product_sin_theta(n-1) = 1.0_default
 do i = n - 2, 1, -1
 product_sin_theta(i) = &
 product_sin_theta(i+1) * sqrt (1 - cos_theta(i)**2)
 end do
 if (present (x)) then
 x(1) = r * product_sin_theta(1) * sin (phi)
 x(2) = r * product_sin_theta(1) * cos (phi)
 x(3:) = r * product_sin_theta(2:n-1) * cos_theta
 end if
 if (present (jacobian)) then
 jacobian = r**(n-1) * product (product_sin_theta(2:))
 end if
end subroutine spherical_cos_to_cartesian_2
```

338b *<Implementation of coordinates procedures 335b>+≡*

```
pure function spherical_cos_to_cartesian (r, phi, theta) result (x)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: theta
 real(kind=default), dimension(size(theta)+2) :: x
 call spherical_cos_to_cartesian_2 (r, phi, theta, x = x)
end function spherical_cos_to_cartesian
```

338c *<Implementation of coordinates procedures 335b>+≡*

```
pure function spherical_cos_to_cartesian_j (r, phi, theta) &
 result (jacobian)
 real(kind=default), intent(in) :: r, phi
 real(kind=default), dimension(:), intent(in) :: theta
 real(kind=default) :: jacobian
 call spherical_cos_to_cartesian_2 (r, phi, theta, jacobian = jacobian)
end function spherical_cos_to_cartesian_j
```

338d *<Declaration of coordinates procedures 335a>+≡*

```
public :: cartesian_to_spherical_cos_2, &
 cartesian_to_spherical_cos, cartesian_to_spherical_cos_j
```

339a *<Implementation of coordinates procedures 335b>+≡*

```

pure subroutine cartesian_to_spherical_cos_2 (x, r, phi, cos_theta, jacobian)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), intent(out), optional :: r, phi
 real(kind=default), dimension(:), intent(out), optional :: cos_theta
 real(kind=default), intent(out), optional :: jacobian
 real(kind=default) :: local_r
 real(kind=default), dimension(size(x)-2) :: local_cos_theta
 real(kind=default), dimension(size(x)-1) :: product_sin_theta
 integer :: n, i
 n = size (x)
 local_r = sqrt (dot_product (x, x))
 product_sin_theta(n-1) = 1
 do i = n, 3, -1
 local_cos_theta(i-2) = x(i) / product_sin_theta(i-1) / local_r
 product_sin_theta(i-2) = &
 product_sin_theta(i-1) * sqrt (1 - local_cos_theta(i-2)**2)
 end do
 if (present (r)) then
 r = local_r
 end if
 if (present (phi)) then
 phi = atan2 (x(1), x(2))
 end if
 if (present (cos_theta)) then
 cos_theta = local_cos_theta
 end if
 if (present (jacobian)) then
 jacobian = local_r**(1-n) / product (product_sin_theta(2:))
 end if
end subroutine cartesian_to_spherical_cos_2

```

339b *<Implementation of coordinates procedures 335b>+≡*

```

pure subroutine cartesian_to_spherical_cos (x, r, phi, cos_theta)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default), intent(out) :: r, phi
 real(kind=default), dimension(:), intent(out), optional :: cos_theta
 call cartesian_to_spherical_cos_2 (x, r, phi, cos_theta)
end subroutine cartesian_to_spherical_cos

```

339c *<Implementation of coordinates procedures 335b>+≡*

```

pure function cartesian_to_spherical_cos_j (x) result (jacobian)
 real(kind=default), dimension(:), intent(in) :: x
 real(kind=default) :: jacobian
 call cartesian_to_spherical_cos_2 (x, jacobian = jacobian)

```

```
end function cartesian_to_spherical_cos_j
```

### *K.3 Surface of a Sphere*

340a *⟨Declaration of coordinates procedures 335a⟩+≡*  

```
public :: surface
```

$$\int d\Omega_n = \frac{2\pi^{n/2}}{\Gamma(n/2)} = S_n \quad (\text{K.7})$$

340b *⟨Implementation of coordinates procedures 335b⟩+≡*  

```
pure function surface (n) result (vol)
 integer, intent(in) :: n
 real(kind=default) :: vol
 real(kind=default) :: n_by_2
 n_by_2 = 0.5_default * n
 vol = 2 * PI**n_by_2 / gamma (n_by_2)
end function surface
```

# —L—

## IDIOMATIC FORTRAN90 INTERFACE FOR MPI

```
341a <mpi90.f90 341a>≡
 ! mpi90.f90 --
 <Copyleft notice 1>
 module mpi90
 use kinds
 use mpi
 implicit none
 private
 <Declaration of mpi90 procedures 341b>
 <Interfaces of mpi90 procedures 344c>
 <Parameters in mpi90 (never defined)>
 <Variables in mpi90 (never defined)>
 <Declaration of mpi90 types 346b>
 character(len=*), public, parameter :: MPI90_RCS_ID = &
 "$Id: mpi90.nw 314 2010-04-17 20:32:33Z ohl $"
 contains
 <Implementation of mpi90 procedures 342a>
 end module mpi90
```

### *L.1 Basics*

```
341b <Declaration of mpi90 procedures 341b>≡
 public :: mpi90_init
 public :: mpi90_finalize
 public :: mpi90_abort
 public :: mpi90_print_error
 public :: mpi90_size
 public :: mpi90_rank
```

```

342a <Implementation of mpi90 procedures 342a>≡
 subroutine mpi90_init (error)
 integer, intent(out), optional :: error
 integer :: local_error
 character(len=*), parameter :: FN = "mpi90_init"
 external mpi_init
 call mpi_init (local_error)
 <Handle local_error (no mpi90_abort) 342b>
 end subroutine mpi90_init

342b <Handle local_error (no mpi90_abort) 342b>≡
 if (present (error)) then
 error = local_error
 else
 if (local_error /= MPI_SUCCESS) then
 call mpi90_print_error (local_error, FN)
 stop
 end if
 end if

342c <Handle local_error 342c>≡
 if (present (error)) then
 error = local_error
 else
 if (local_error /= MPI_SUCCESS) then
 call mpi90_print_error (local_error, FN)
 call mpi90_abort (local_error)
 stop
 end if
 end if

342d <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_finalize (error)
 integer, intent(out), optional :: error
 integer :: local_error
 character(len=*), parameter :: FN = "mpi90_finalize"
 external mpi_finalize
 call mpi_finalize (local_error)
 <Handle local_error 342c>
 end subroutine mpi90_finalize

342e <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_abort (code, domain, error)
 integer, intent(in), optional :: code, domain
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_abort"

```

```

integer :: local_domain, local_code, local_error
external mpi_abort
if (present (code)) then
 local_code = code
else
 local_code = MPI_ERR_UNKNOWN
end if
<Set default for domain 343b>
call mpi_abort (local_domain, local_code, local_error)
<Handle local_error (no mpi90_abort) 342b>
end subroutine mpi90_abort

```

343a <Implementation of mpi90 procedures 342a>+≡

```

subroutine mpi90_print_error (error, msg)
integer, intent(in) :: error
character(len=*), optional :: msg
character(len=*), parameter :: FN = "mpi90_print_error"
integer :: msg_len, local_error
external mpi_error_string
call mpi_error_string (error, msg, msg_len, local_error)
if (local_error /= MPI_SUCCESS) then
 print *, "PANIC: even MPI_ERROR_STRING() failed!!!"
 call mpi90_abort (local_error)
else if (present (msg)) then
 print *, trim (msg), ": ", trim (msg(msg_len+1:))
else
 print *, "mpi90: ", trim (msg(msg_len+1:))
end if
end subroutine mpi90_print_error

```

343b <Set default for domain 343b>≡

```

if (present (domain)) then
 local_domain = domain
else
 local_domain = MPI_COMM_WORLD
end if

```

343c <Implementation of mpi90 procedures 342a>+≡

```

subroutine mpi90_size (sz, domain, error)
integer, intent(out) :: sz
integer, intent(in), optional :: domain
integer, intent(out), optional :: error
character(len=*), parameter :: FN = "mpi90_size"
integer :: local_domain, local_error
external mpi_comm_size

```



```

 <Set default for domain 343b>
 call mpi_comm_size (local_domain, sz, local_error)
 <Handle local_error 342c>
end subroutine mpi90_size

344a <Implementation of mpi90 procedures 342a>+≡
subroutine mpi90_rank (rank, domain, error)
 integer, intent(out) :: rank
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_rank"
 integer :: local_domain, local_error
 external mpi_comm_rank
 <Set default for domain 343b>
 call mpi_comm_rank (local_domain, rank, local_error)
 <Handle local_error 342c>
end subroutine mpi90_rank

```

## *L.2 Point to Point*

```

344b <Declaration of mpi90 procedures 341b>+≡
 public :: mpi90_send
 public :: mpi90_receive
 public :: mpi90_receive_pointer

344c <Interfaces of mpi90 procedures 344c>≡
 interface mpi90_send
 module procedure &
 mpi90_send_integer, mpi90_send_double, &
 mpi90_send_integer_array, mpi90_send_double_array, &
 mpi90_send_integer_array2, mpi90_send_double_array2
 end interface

344d <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_send_integer (value, target, tag, domain, error)
 integer, intent(in) :: value
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 call mpi90_send_integer_array ((/ value /), target, tag, domain, error)
 end subroutine mpi90_send_integer

344e <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_send_double (value, target, tag, domain, error)

```

```

 real(kind=default), intent(in) :: value
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 call mpi90_send_double_array (/ value /), target, tag, domain, error)
end subroutine mpi90_send_double

345a ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_send_integer_array (buffer, target, tag, domain, error)
 integer, dimension(:), intent(in) :: buffer
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_send_integer_array"
 integer, parameter :: datatype = MPI_INTEGER
 ⟨Body of mpi90_send_*_array 345b⟩
 end subroutine mpi90_send_integer_array

345b ⟨Body of mpi90_send_*_array 345b⟩≡
 integer :: local_domain, local_error
 external mpi_send
 ⟨Set default for domain 343b⟩
 call mpi_send (buffer, size (buffer), datatype, target, tag, &
 local_domain, local_error)
 ⟨Handle local_error 342c⟩

345c ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_send_double_array (buffer, target, tag, domain, error)
 real(kind=default), dimension(:), intent(in) :: buffer
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_send_double_array"
 integer, parameter :: datatype = MPI_DOUBLE_PRECISION
 ⟨Body of mpi90_send_*_array 345b⟩
 end subroutine mpi90_send_double_array

345d ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_send_integer_array2 (value, target, tag, domain, error)
 integer, dimension(:, :), intent(in) :: value
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_send_integer_array (buffer, target, tag, domain, error)

```

```

 end subroutine mpi90_send_integer_array2

346a <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_send_double_array2 (value, target, tag, domain, error)
 real(kind=default), dimension(:,:), intent(in) :: value
 integer, intent(in) :: target, tag
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 real(kind=default), dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_send_double_array (buffer, target, tag, domain, error)
 end subroutine mpi90_send_double_array2

346b <Declaration of mpi90 types 346b>≡
 type, public :: mpi90_status
 integer :: count, source, tag, error
 end type mpi90_status

346c <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_receive_integer (value, source, tag, domain, status, error)
 integer, intent(out) :: value
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 integer, dimension(1) :: buffer
 call mpi90_receive_integer_array (buffer, source, tag, domain, status, error)
 value = buffer(1)
 end subroutine mpi90_receive_integer

346d <Interfaces of mpi90 procedures 344c>+≡
 interface mpi90_receive
 module procedure &
 mpi90_receive_integer, mpi90_receive_double, &
 mpi90_receive_integer_array, mpi90_receive_double_array, &
 mpi90_receive_integer_array2, mpi90_receive_double_array2
 end interface

346e <Set defaults for source, tag and domain 346e>≡
 if (present (source)) then
 local_source = source
 else
 local_source = MPI_ANY_SOURCE
 end if
 if (present (tag)) then
 local_tag = tag
 else

```

```

 local_tag = MPI_ANY_TAG
 end if
 <Set default for domain 343b>
347a <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_receive_double (value, source, tag, domain, status, error)
 real(kind=default), intent(out) :: value
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 real(kind=default), dimension(1) :: buffer
 call mpi90_receive_double_array (buffer, source, tag, domain, status, error)
 value = buffer(1)
 end subroutine mpi90_receive_double
347b <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_receive_integer_array &
 (buffer, source, tag, domain, status, error)
 integer, dimension(:), intent(out) :: buffer
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_receive_integer_array"
 integer, parameter :: datatype = MPI_INTEGER
 <Body of mpi90_receive_*_array 347c>
 end subroutine mpi90_receive_integer_array
347c <Body of mpi90_receive_*_array 347c>≡
 integer :: local_source, local_tag, local_domain, local_error
 integer, dimension(MPI_STATUS_SIZE) :: local_status
 external mpi_receive, mpi_get_count
 <Set defaults for source, tag and domain 346e>
 call mpi_recv (buffer, size (buffer), datatype, local_source, local_tag, &
 local_domain, local_status, local_error)
 <Handle local_error 342c>
 if (present (status)) then
 call decode_status (status, local_status, datatype)
 end if
347d <Declaration of mpi90 procedures 341b>+≡
 private :: decode_status

```

 Can we ignore ierror???

```

347e <Implementation of mpi90 procedures 342a>+≡
 subroutine decode_status (status, mpi_status, datatype)

```

```

type(mpi90_status), intent(out) :: status
integer, dimension(:), intent(in) :: mpi_status
integer, intent(in), optional :: datatype
integer :: ierror
if (present (datatype)) then
 call mpi_get_count (mpi_status, datatype, status%count, ierror)
else
 status%count = 0
end if
status%source = mpi_status(MPI_SOURCE)
status%tag = mpi_status(MPI_TAG)
status%error = mpi_status(MPI_ERROR)
end subroutine decode_status

```

348a *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_receive_double_array &
 (buffer, source, tag, domain, status, error)
 real(kind=default), dimension(:), intent(out) :: buffer
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_receive_double_array"
 integer, parameter :: datatype = MPI_DOUBLE_PRECISION
 <Body of mpi90_receive_*_array 347c>
end subroutine mpi90_receive_double_array

```

348b *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_receive_integer_array2 &
 (value, source, tag, domain, status, error)
 integer, dimension(:,:), intent(out) :: value
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 integer, dimension(size(value)) :: buffer
 call mpi90_receive_integer_array &
 (buffer, source, tag, domain, status, error)
 value = reshape (buffer, shape(value))
end subroutine mpi90_receive_integer_array2

```

348c *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_receive_double_array2 &
 (value, source, tag, domain, status, error)
 real(kind=default), dimension(:,:), intent(out) :: value
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status

```

```

integer, intent(out), optional :: error
real(kind=default), dimension(size(value)) :: buffer
call mpi90_receive_double_array &
 (buffer, source, tag, domain, status, error)
value = reshape (buffer, shape(value))
end subroutine mpi90_receive_double_array2

```

349a *<Interfaces of mpi90 procedures 344c>+≡*

```

interface mpi90_receive_pointer
 module procedure &
 mpi90_receive_integer_pointer, mpi90_receive_double_pointer
end interface

```

349b *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_receive_integer_pointer &
 (buffer, source, tag, domain, status, error)
integer, dimension(:), pointer :: buffer
integer, intent(in), optional :: source, tag, domain
type(mpi90_status), intent(out), optional :: status
integer, intent(out), optional :: error
character(len=*), parameter :: FN = "mpi90_receive_integer_pointer"
integer, parameter :: datatype = MPI_INTEGER
<Body of mpi90_receive_*_pointer 349c>
end subroutine mpi90_receive_integer_pointer

```

349c *<Body of mpi90\_receive\_\*\_pointer 349c>≡*

```

integer :: local_source, local_tag, local_domain, local_error, buffer_size
integer, dimension(MPI_STATUS_SIZE) :: local_status
integer :: ierror
external mpi_receive, mpi_get_count
<Set defaults for source, tag and domain 346e>

```

349d *<Body of mpi90\_receive\_\*\_pointer 349c>+≡*

```

call mpi_probe (local_source, local_tag, local_domain, &
 local_status, local_error)
<Handle local_error 342c>

```

 Can we ignore ierror???

349e *<Body of mpi90\_receive\_\*\_pointer 349c>+≡*

```

call mpi_get_count (local_status, datatype, buffer_size, ierror)
if (associated (buffer)) then
 if (size (buffer) /= buffer_size) then
 deallocate (buffer)
 allocate (buffer(buffer_size))
 end if

```

```

else
 allocate (buffer(buffer_size))
end if

350a <Body of mpi90_receive_*_pointer 349c>+≡
 call mpi_recv (buffer, size (buffer), datatype, local_source, local_tag, &
 local_domain, local_status, local_error)

350b <Body of mpi90_receive_*_pointer 349c>+≡
 <Handle local_error 342c>
 if (present (status)) then
 call decode_status (status, local_status, datatype)
 end if

350c <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_receive_double_pointer &
 (buffer, source, tag, domain, status, error)
 real(kind=default), dimension(:), pointer :: buffer
 integer, intent(in), optional :: source, tag, domain
 type(mpi90_status), intent(out), optional :: status
 integer, intent(out), optional :: error
 character(len=*), parameter :: FN = "mpi90_receive_double_pointer"
 integer, parameter :: datatype = MPI_DOUBLE_PRECISION
 <Body of mpi90_receive_*_pointer 349c>
 end subroutine mpi90_receive_double_pointer

```

### *L.3 Collective Communication*

```

350d <Declaration of mpi90 procedures 341b>+≡
 public :: mpi90_broadcast

350e <Interfaces of mpi90 procedures 344c>+≡
 interface mpi90_broadcast
 module procedure &
 mpi90_broadcast_integer, mpi90_broadcast_integer_array, &
 mpi90_broadcast_integer_array2, mpi90_broadcast_integer_array3, &
 mpi90_broadcast_double, mpi90_broadcast_double_array, &
 mpi90_broadcast_double_array2, mpi90_broadcast_double_array3, &
 mpi90_broadcast_logical, mpi90_broadcast_logical_array, &
 mpi90_broadcast_logical_array2, mpi90_broadcast_logical_array3
 end interface

350f <Set default for domain 343b>+≡
 if (present (domain)) then
 local_domain = domain
 end if

```

```

else
 local_domain = MPI_COMM_WORLD
end if

351a ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_broadcast_integer (value, root, domain, error)
 integer, intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, dimension(1) :: buffer
 buffer(1) = value
 call mpi90_broadcast_integer_array (buffer, root, domain, error)
 value = buffer(1)
 end subroutine mpi90_broadcast_integer

351b ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_broadcast_double (value, root, domain, error)
 real(kind=default), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 real(kind=default), dimension(1) :: buffer
 buffer(1) = value
 call mpi90_broadcast_double_array (buffer, root, domain, error)
 value = buffer(1)
 end subroutine mpi90_broadcast_double

351c ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_broadcast_logical (value, root, domain, error)
 logical, intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 logical, dimension(1) :: buffer
 buffer(1) = value
 call mpi90_broadcast_logical_array (buffer, root, domain, error)
 value = buffer(1)
 end subroutine mpi90_broadcast_logical

351d ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_broadcast_integer_array (buffer, root, domain, error)
 integer, dimension(:), intent(inout) :: buffer
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error

```



```

 character(len=*), parameter :: FN = "mpi90_broadcast_integer_array"
 integer, parameter :: datatype = MPI_INTEGER
 <Body of mpi90_broadcast_*_array 352a>
 end subroutine mpi90_broadcast_integer_array

352a <Body of mpi90_broadcast_*_array 352a>≡
 integer :: local_domain, local_error
 external mpi_bcast
 <Set default for domain 343b>
 call mpi_bcast (buffer, size (buffer), datatype, root, &
 local_domain, local_error)
 <Handle local_error 342c>

352b <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_broadcast_double_array (buffer, root, domain, error)
 real(kind=default), dimension(:), intent(inout) :: buffer
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, parameter :: datatype = MPI_DOUBLE_PRECISION
 character(len=*), parameter :: FN = "mpi90_broadcast_double_array"
 <Body of mpi90_broadcast_*_array 352a>
 end subroutine mpi90_broadcast_double_array

352c <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_broadcast_logical_array (buffer, root, domain, error)
 logical, dimension(:), intent(inout) :: buffer
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, parameter :: datatype = MPI_LOGICAL
 character(len=*), parameter :: FN = "mpi90_broadcast_logical_array"
 <Body of mpi90_broadcast_*_array 352a>
 end subroutine mpi90_broadcast_logical_array

352d <Implementation of mpi90 procedures 342a>+≡
 subroutine mpi90_broadcast_integer_array2 (value, root, domain, error)
 integer, dimension(:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_integer_array (buffer, root, domain, error)
 value = reshape (buffer, shape(value))
 end subroutine mpi90_broadcast_integer_array2

```

353a *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_broadcast_double_array2 (value, root, domain, error)
 real(kind=default), dimension(:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 real(kind=default), dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_double_array (buffer, root, domain, error)
 value = reshape (buffer, shape(value))
end subroutine mpi90_broadcast_double_array2

```

353b *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_broadcast_logical_array2 (value, root, domain, error)
 logical, dimension(:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 logical, dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_logical_array (buffer, root, domain, error)
 value = reshape (buffer, shape(value))
end subroutine mpi90_broadcast_logical_array2

```

353c *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_broadcast_integer_array3 (value, root, domain, error)
 integer, dimension(:,:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 integer, dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_integer_array (buffer, root, domain, error)
 value = reshape (buffer, shape(value))
end subroutine mpi90_broadcast_integer_array3

```

353d *<Implementation of mpi90 procedures 342a>+≡*

```

subroutine mpi90_broadcast_double_array3 (value, root, domain, error)
 real(kind=default), dimension(:,:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 real(kind=default), dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_double_array (buffer, root, domain, error)

```

```

 value = reshape (buffer, shape(value))
 end subroutine mpi90_broadcast_double_array3
354 ⟨Implementation of mpi90 procedures 342a⟩+≡
 subroutine mpi90_broadcast_logical_array3 (value, root, domain, error)
 logical, dimension(:,:,:), intent(inout) :: value
 integer, intent(in) :: root
 integer, intent(in), optional :: domain
 integer, intent(out), optional :: error
 logical, dimension(size(value)) :: buffer
 buffer = reshape (value, shape(buffer))
 call mpi90_broadcast_logical_array (buffer, root, domain, error)
 value = reshape (buffer, shape(value))
 end subroutine mpi90_broadcast_logical_array3

```

# —M—

## POOR MAN’S ELEMENTAL PROCEDURES

On one hand, I want to take advantage of Fortran95’s improvements, but on the other hand, I want to continue to use F as a style guide and also allow people with Fortran90 compilers to use the library.

### *M.1 m4 Macros*

```
355a <f95.m4 355a>≡
 dnl f95.m4 --
 divert(-1)dnl
 <Kill m4(1) builtins 355b>
 define('_specific_sv','dnl')
 define('_interface_sv','dnl')
 define('_specific_sva','dnl')
 define('_interface_sva','dnl')
 define('_begin_f90','divert(-1)dnl')
 define('_end_f90','divert''dnl')
 divert''dnl

355b <Kill m4(1) builtins 355b>≡
 undefine('eval')

355c <f90.m4 355c>≡
 dnl f90.m4 --
 divert(-1)dnl
 <Kill m4(1) builtins 355b>
 define('pure','')
 define('elemental','')
 define('_specific_sv','private :: $1_s, $1_v')
 define('_interface_sv','interface $1
 module procedure $1_s, $1_v
 end interface''define($1,$1_s)')
 define('_specific_sva','private :: $1_s, $1_v, $1_a')
```

```

define('_interface_sva','interface $1
 module procedure $1_s, $1_v, $1_a
 end interface''define($1,$1_s)')
define('_begin_f90','dnl')
define('_end_f90','dnl')
divert''dnl

```

## *M.2 Miscellaneous Utilities*

**356a** *<Implementation of utils procedures 303c>+≡*  
       \_begin\_f90

**356b** *<Declaration of utils procedures 302b>+≡*  
       \_begin\_f90  
       private :: swap\_integer\_array, swap\_real\_array  
       \_end\_f90

**356c** *<Interfaces of utils procedures 302c>+≡*  
       \_begin\_f90  
       interface swap  
           module procedure swap\_integer\_array, swap\_real\_array  
       end interface  
       \_end\_f90

**356d** *<Implementation of utils procedures 303c>+≡*  
       pure subroutine swap\_integer\_array (a, b)  
           integer, dimension(:), intent(inout) :: a, b  
           integer, dimension(max(size(a),size(b))) :: tmp  
           tmp = a  
           a = b  
           b = tmp  
       end subroutine swap\_integer\_array

**356e** *<Implementation of utils procedures 303c>+≡*  
       pure subroutine swap\_real\_array (a, b)  
           real(kind=default), dimension(:), intent(inout) :: a, b  
           real(kind=default), dimension(max(size(a),size(b))) :: tmp  
           tmp = a  
           a = b  
           b = tmp  
       end subroutine swap\_real\_array

**356f** *<Declaration of utils procedures 302b>+≡*  
       \_specific\_sva(gcd)  
       \_specific\_sva(lcm)

```

357a <Interfaces of utils procedures 302c>+≡
 _interface_sva(gcd)
 _interface_sva(lcm)

357b <Implementation of utils procedures 303c>+≡
 pure function gcd_v (m, n) result (gcd_m_n)
 integer, dimension(:), intent(in) :: m, n
 integer, dimension(size(m)) :: gcd_m_n
 integer :: i
 do i = 1, size (m)
 gcd_m_n(i) = gcd (m(i), n(i))
 end do
 end function gcd_v

357c <Implementation of utils procedures 303c>+≡
 pure function lcm_v (m, n) result (lcm_m_n)
 integer, dimension(:), intent(in) :: m, n
 integer, dimension(size(m)) :: lcm_m_n
 integer :: i
 do i = 1, size (m)
 lcm_m_n(i) = lcm (m(i), n(i))
 end do
 end function lcm_v

```

This abuses the `_a` suffix, which is used elsewhere for two-dimensional arrays:

```

357d <Implementation of utils procedures 303c>+≡
 pure function gcd_a (m, n) result (gcd_m_n)
 integer, dimension(:), intent(in) :: m
 integer, intent(in) :: n
 integer, dimension(size(m)) :: gcd_m_n
 integer :: i
 do i = 1, size (m)
 gcd_m_n(i) = gcd (m(i), n)
 end do
 end function gcd_a

357e <Implementation of utils procedures 303c>+≡
 pure function lcm_a (m, n) result (lcm_m_n)
 integer, dimension(:), intent(in) :: m
 integer, intent(in) :: n
 integer, dimension(size(m)) :: lcm_m_n
 integer :: i
 do i = 1, size (m)
 lcm_m_n(i) = lcm (m(i), n)
 end do
 end function lcm_a

```

358a *<Implementation of utils procedures 303c>+≡*  
       \_end\_f90

### *M.3 Errors and Exceptions*

358b *<Implementation of exceptions procedures 244c>+≡*  
       \_begin\_f90

358c *<Declaration of exceptions procedures 244b>+≡*  
       \_specific\_sv(raise\_exception)  
       \_specific\_sv(clear\_exception)

358d *<Interfaces of exceptions procedures 358d>≡*  
       \_interface\_sv(raise\_exception)  
       \_interface\_sv(clear\_exception)

358e *<Implementation of exceptions procedures 244c>+≡*  
       pure subroutine raise\_exception\_v (exc, level, message, origin)  
           type(exception), dimension(:), intent(inout) :: exc  
           integer, dimension(:), intent(in) :: level  
           character(len=\*), dimension(:), intent(in), optional :: message, origin  
           integer :: i  
           do i = 1, size (exc)  
               call raise\_exception (exc(i), level(i), message(i), origin(i))  
           end do  
       end subroutine raise\_exception\_v

358f *<Implementation of exceptions procedures 244c>+≡*  
       pure subroutine clear\_exception\_v (exc)  
           type(exception), dimension(:), intent(inout) :: exc  
           integer :: i  
           do i = 1, size (exc)  
               call clear\_exception (exc(i))  
           end do  
       end subroutine clear\_exception\_v

358g *<Implementation of exceptions procedures 244c>+≡*  
       \_end\_f90

### *M.4 The Abstract Datatype division*

358h *<Implementation of divisions procedures 38b>+≡*  
       \_begin\_f90

```

359a <Declaration of divisions procedures 38a>+≡
 _specific_sv(create_division)
 _specific_sv(create_empty_division)
 _specific_sva(copy_division)
 _specific_sv(set_rigid_division)
 _specific_sv(reshape_division)
 _specific_sv(delete_division)

359b <Interfaces of divisions procedures 60e>+≡
 _interface_sv(create_division)
 _interface_sv(create_empty_division)
 _interface_sva(copy_division)
 _interface_sv(set_rigid_division)
 _interface_sv(reshape_division)
 _interface_sv(delete_division)

359c <Implementation of divisions procedures 38b>+≡
 pure subroutine create_division_v (d, x_min, x_max, x_min_true, x_max_true)
 type(division), dimension(:), intent(out) :: d
 real(kind=default), dimension(:), intent(in) :: x_min, x_max
 real(kind=default), dimension(:), intent(in), optional :: &
 x_min_true, x_max_true
 integer :: j
 do j = 1, size (d)
 if (present (x_min_true) .and. present (x_max_true)) then
 call create_division &
 (d(j), x_min(j), x_max(j), x_min_true(j), x_max_true(j))
 else
 call create_division (d(j), x_min(j), x_max(j))
 end if
 end do
 end subroutine create_division_v

359d <Implementation of divisions procedures 38b>+≡
 pure subroutine create_empty_division_v (d)
 type(division), dimension(:), intent(out) :: d
 integer :: j
 do j = 1, size (d)
 call create_empty_division (d(j))
 end do
 end subroutine create_empty_division_v

359e <Implementation of divisions procedures 38b>+≡
 pure subroutine copy_division_v (lhs, rhs)
 type(division), dimension(:), intent(inout) :: lhs
 type(division), dimension(:), intent(in) :: rhs

```



```

integer :: j
do j = 1, size(lhs)
 call copy_division (lhs(j), rhs(j))
end do
end subroutine copy_division_v

```

This abuses the `_a` suffix, which is used elsewhere for two-dimensional arrays:

**360a** *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine copy_division_a (lhs, rhs)
 type(division), dimension(:), intent(inout) :: lhs
 type(division), intent(in) :: rhs
 integer :: j
 do j = 1, size(lhs)
 call copy_division (lhs(j), rhs)
 end do
end subroutine copy_division_a

```

**360b** *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine set_rigid_division_v (d, ng)
 type(division), dimension(:), intent(inout) :: d
 integer, dimension(:), intent(in) :: ng
 integer :: j
 do j = 1, size(d)
 call set_rigid_division (d(j), ng(j))
 end do
end subroutine set_rigid_division_v

```

**360c** *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine reshape_division_v (d, max_num_div, ng, use_variance)
 type(division), dimension(:), intent(inout) :: d
 integer, dimension(:), intent(in) :: max_num_div
 integer, dimension(:), intent(in), optional :: ng
 logical, intent(in), optional :: use_variance
 integer :: j
 do j = 1, size(d)
 call reshape_division (d(j), max_num_div(j), ng(j), use_variance)
 end do
end subroutine reshape_division_v

```

**360d** *<Implementation of divisions procedures 38b>+≡*

```

pure subroutine delete_division_v (d)
 type(division), dimension(:), intent(inout) :: d
 integer :: j
 do j = 1, size(d)
 call delete_division (d(j))
 end do

```

```

 end subroutine delete_division_v

361a <Declaration of divisions procedures 38a>+≡
 _specific_sv(inject_division)
 _specific_sv(inject_division_short)

361b <Interfaces of divisions procedures 60e>+≡
 _interface_sv(inject_division)
 _interface_sv(inject_division_short)

361c <Implementation of divisions procedures 38b>+≡
 pure subroutine inject_division_v (d, r, cell, x, x_mid, idx, wgt)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(:), intent(in) :: r
 integer, dimension(:), intent(in) :: cell
 real(kind=default), dimension(:), intent(out) :: x, x_mid
 integer, dimension(:), intent(out) :: idx
 real(kind=default), dimension(:), intent(out) :: wgt
 integer :: j
 do j = 1, size (d)
 call inject_division (d(j), r(j), cell(j), x(j), &
 x_mid(j), idx(j), wgt(j))
 end do
 end subroutine inject_division_v

361d <Implementation of divisions procedures 38b>+≡
 pure subroutine inject_division_short_v (d, r, x, idx, wgt)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(:), intent(in) :: r
 real(kind=default), dimension(:), intent(out) :: x
 integer, dimension(:), intent(out) :: idx
 real(kind=default), dimension(:), intent(out) :: wgt
 integer :: j
 do j = 1, size (d)
 call inject_division_short (d(j), r(j), x(j), idx(j), wgt(j))
 end do
 end subroutine inject_division_short_v

361e <Declaration of divisions procedures 38a>+≡
 _specific_sv(record_integral)
 _specific_sv(record_variance)
 _specific_sv(clear_integral_and_variance)

361f <Declaration of divisions procedures (removed from WHIZARD) 59c>+≡
 _specific_sv(record_efficiency)

```

362a  $\langle$ Interfaces of divisions procedures 60e $\rangle + \equiv$   
`_interface_sv(record_integral)`  
`_interface_sv(record_variance)`  
`_interface_sv(clear_integral_and_variance)`

362b  $\langle$ Interfaces of divisions procedures (removed from WHIZARD) 362b $\rangle \equiv$   
`_interface_sv(record_efficiency)`

362c  $\langle$ Implementation of divisions procedures 38b $\rangle + \equiv$   

```

pure subroutine record_integral_v (d, i, f)
 type(division), dimension(:), intent(inout) :: d
 integer, dimension(:), intent(in) :: i
 real(kind=default), intent(in) :: f
 integer :: j
 do j = 1, size (d)
 call record_integral (d(j), i(j), f)
 end do
end subroutine record_integral_v

```

362d  $\langle$ Implementation of divisions procedures 38b $\rangle + \equiv$   

```

pure subroutine record_variance_v (d, i, var_f)
 type(division), dimension(:), intent(inout) :: d
 integer, dimension(:), intent(in) :: i
 real(kind=default), intent(in) :: var_f
 integer :: j
 do j = 1, size (d)
 call record_variance (d(j), i(j), var_f)
 end do
end subroutine record_variance_v

```

362e  $\langle$ Implementation of divisions procedures (removed from WHIZARD) 44e $\rangle + \equiv$   

```

pure subroutine record_efficiency_v (d, i, eff_f)
 type(division), dimension(:), intent(inout) :: d
 integer, dimension(:), intent(in) :: i
 real(kind=default), intent(in) :: eff_f
 integer :: j
 do j = 1, size (d)
 call record_efficiency (d(j), i(j), eff_f)
 end do
end subroutine record_efficiency_v

```

362f  $\langle$ Implementation of divisions procedures 38b $\rangle + \equiv$   

```

pure subroutine clear_integral_and_variance_v (d)
 type(division), dimension(:), intent(inout) :: d
 integer :: j
 do j = 1, size (d)

```

```

 call clear_integral_and_variance (d(j))
 end do
end subroutine clear_integral_and_variance_v

363a <Declaration of divisions procedures 38a>+≡
 _specific_sv(refine_division)

363b <Interfaces of divisions procedures 60e>+≡
 _interface_sv(refine_division)

363c <Implementation of divisions procedures 38b>+≡
 pure subroutine refine_division_v (d)
 type(division), dimension(:), intent(inout) :: d
 integer :: j
 do j = 1, size (d)
 call refine_division (d(j))
 end do
 end subroutine refine_division_v

363d <Declaration of divisions procedures 38a>+≡
 _specific_sv(probability)
 _specific_sv(inside_division)
 _specific_sv(stratified_division)
 _specific_sv(volume_division)
 _specific_sv(rigid_division)
 _specific_sv(adaptive_division)
 _specific_sv(quadrapole_division)

363e <Interfaces of divisions procedures 60e>+≡
 _interface_sv(probability)
 _interface_sv(inside_division)
 _interface_sv(stratified_division)
 _interface_sv(volume_division)
 _interface_sv(rigid_division)
 _interface_sv(adaptive_division)
 _interface_sv(quadrapole_division)

363f <Implementation of divisions procedures 38b>+≡
 pure function probability_v (d, xi) result (p)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(:), intent(in) :: xi
 real(kind=default), dimension(size(d)) :: p
 integer :: j
 do j = 1, size (d)
 p(j) = probability (d(j), xi(j))
 end do
 end function probability_v

```

```

364a <Implementation of divisions procedures 38b>+≡
 pure function inside_division_v (d, x) result (theta)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(:), intent(in) :: x
 logical, dimension(size(d)) :: theta
 integer :: j
 do j = 1, size (d)
 theta(j) = inside_division (d(j), x(j))
 end do
 end function inside_division_v

364b <Implementation of divisions procedures 38b>+≡
 pure function stratified_division_v (d) result (yorn)
 type(division), dimension(:), intent(in) :: d
 logical, dimension(size(d)) :: yorn
 integer :: j
 do j = 1, size (d)
 yorn(j) = stratified_division (d(j))
 end do
 end function stratified_division_v

364c <Implementation of divisions procedures 38b>+≡
 pure function volume_division_v (d) result (vol)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(size(d)) :: vol
 integer :: j
 do j = 1, size(d)
 vol(j) = volume_division (d(j))
 end do
 end function volume_division_v

364d <Implementation of divisions procedures 38b>+≡
 pure function rigid_division_v (d) result (n)
 type(division), dimension(:), intent(in) :: d
 integer, dimension(size(d)) :: n
 integer :: j
 do j = 1, size(d)
 n(j) = rigid_division (d(j))
 end do
 end function rigid_division_v

364e <Implementation of divisions procedures 38b>+≡
 pure function adaptive_division_v (d) result (n)
 type(division), dimension(:), intent(in) :: d
 integer, dimension(size(d)) :: n
 integer :: j

```

```

 do j = 1, size(d)
 n(j) = adaptive_division (d(j))
 end do
 end function adaptive_division_v

365a <Implementation of divisions procedures 38b>+≡
 pure function quadrupole_division_v (d) result (q)
 type(division), dimension(:), intent(in) :: d
 real(kind=default), dimension(size(d)) :: q
 integer :: j
 do j = 1, size (d)
 q(j) = quadrupole_division (d(j))
 end do
 end function quadrupole_division_v

365b <Declaration of divisions procedures 38a>+≡
 _specific_sv(copy_history)
 _specific_sv(summarize_division)

365c <Interfaces of divisions procedures 60e>+≡
 _interface_sv(copy_history)
 _interface_sv(summarize_division)

365d <Implementation of divisions procedures 38b>+≡
 pure subroutine copy_history_v (lhs, rhs)
 type(div_history), dimension(:), intent(inout) :: lhs
 type(div_history), dimension(:), intent(in) :: rhs
 integer :: j
 do j = 1, size (rhs)
 call copy_history (lhs(j), rhs(j))
 end do
 end subroutine copy_history_v

365e <Implementation of divisions procedures 38b>+≡
 pure function summarize_division_v (d) result (s)
 type(division), dimension(:), intent(in) :: d
 type(div_history), dimension(size(d)) :: s
 integer :: j
 do j = 1, size (d)
 s(j) = summarize_division (d(j))
 end do
 end function summarize_division_v

365f <Implementation of divisions procedures 38b>+≡
 _end_f90

```

## M.5 The Abstract Datatype *vamp\_grid*

- 366a *⟨Implementation of vamp procedures 76d⟩*+≡  
    \_begin\_f90
- 366b *⟨Declaration of vamp procedures 76b⟩*+≡  
    \_specific\_sv(vamp\_create\_empty\_grid)  
    \_specific\_sv(vamp\_nullify\_covariance)  
    \_specific\_sv(vamp\_get\_variance)  
    \_specific\_sv(vamp\_nullify\_variance)
- 366c *⟨Interfaces of vamp procedures 93b⟩*+≡  
    \_interface\_sv(vamp\_create\_empty\_grid)  
    \_interface\_sv(vamp\_nullify\_covariance)  
    \_interface\_sv(vamp\_get\_variance)  
    \_interface\_sv(vamp\_nullify\_variance)
- 366d *⟨Implementation of vamp procedures 76d⟩*+≡  
    pure subroutine vamp\_create\_empty\_grid\_v (g)  
        type(vamp\_grid), dimension(:), intent(inout) :: g  
        integer :: i  
        do i = 1, size (g)  
            call vamp\_create\_empty\_grid (g(i))  
        end do  
    end subroutine vamp\_create\_empty\_grid\_v
- 366e *⟨Implementation of vamp procedures 76d⟩*+≡  
    pure subroutine vamp\_nullify\_covariance\_v (g)  
        type(vamp\_grid), dimension(:), intent(inout) :: g  
        integer :: i  
        do i = 1, size (g)  
            call vamp\_nullify\_covariance (g(i))  
        end do  
    end subroutine vamp\_nullify\_covariance\_v
- 366f *⟨Implementation of vamp procedures 76d⟩*+≡  
    pure function vamp\_get\_variance\_v (g) result (v)  
        type(vamp\_grid), dimension(:), intent(in) :: g  
        real(kind=default), dimension(size(g)) :: v  
        integer :: i  
        do i = 1, size (g)  
            v(i) = vamp\_get\_variance (g(i))  
        end do  
    end function vamp\_get\_variance\_v

367a *<Implementation of vamp procedures 76d>+≡*  

```

pure subroutine vamp_nullify_variance_v (g)
 type(vamp_grid), dimension(:), intent(inout) :: g
 integer :: i
 do i = 1, size (g)
 call vamp_nullify_variance (g(i))
 end do
end subroutine vamp_nullify_variance_v

```

367b *<Declaration of vamp procedures 76b>+≡*  

```

_specific_sv(vamp_copy_grid)
_specific_sv(vamp_delete_grid)

```

367c *<Interfaces of vamp procedures 93b>+≡*  

```

_interface_sv(vamp_copy_grid)
_interface_sv(vamp_delete_grid)

```

367d *<Implementation of vamp procedures 76d>+≡*  

```

pure subroutine vamp_copy_grid_v (lhs, rhs)
 type(vamp_grid), dimension(:), intent(inout) :: lhs
 type(vamp_grid), dimension(:), intent(in) :: rhs
 integer :: i
 do i = 1, size (lhs)
 call vamp_copy_grid (lhs(i), rhs(i))
 end do
end subroutine vamp_copy_grid_v

```

367e *<Implementation of vamp procedures 76d>+≡*  

```

pure subroutine vamp_delete_grid_v (g)
 type(vamp_grid), dimension(:), intent(inout) :: g
 integer :: i
 do i = 1, size (g)
 call vamp_delete_grid (g(i))
 end do
end subroutine vamp_delete_grid_v

```

367f *<Declaration of vamp procedures 76b>+≡*  

```

_specific_sv(vamp_copy_grids)
_specific_sv(vamp_delete_grids)

```

367g *<Interfaces of vamp procedures 93b>+≡*  

```

_interface_sv(vamp_copy_grids)
_interface_sv(vamp_delete_grids)

```

367h *<Implementation of vamp procedures 76d>+≡*  

```

pure subroutine vamp_copy_grids_v (lhs, rhs)
 type(vamp_grids), dimension(:), intent(inout) :: lhs

```



```

 type(vamp_grids), dimension(:), intent(in) :: rhs
 integer :: i
 do i = 1, size (lhs)
 call vamp_copy_grids (lhs(i), rhs(i))
 end do
 end subroutine vamp_copy_grids_v
368a <Implementation of vamp procedures 76d>+≡
 pure subroutine vamp_delete_grids_v (g)
 type(vamp_grids), dimension(:), intent(inout) :: g
 integer :: i
 do i = 1, size (g)
 call vamp_delete_grids (g(i))
 end do
 end subroutine vamp_delete_grids_v
368b <Declaration of vamp procedures 76b>+≡
 _specific_sva(vamp_create_history)
 _specific_sv(vamp_terminate_history)
 _specific_sva(vamp_copy_history)
 _specific_sva(vamp_delete_history)
368c <Interfaces of vamp procedures 93b>+≡
 _interface_sva(vamp_create_history)
 _interface_sv(vamp_terminate_history)
 _interface_sva(vamp_copy_history)
 _interface_sva(vamp_delete_history)
368d <Implementation of vamp procedures 76d>+≡
 pure subroutine vamp_create_history_v (h, ndim, verbose)
 type(vamp_history), dimension(:), intent(inout) :: h
 integer, intent(in), optional :: ndim
 logical, intent(in), optional :: verbose
 integer :: i
 do i = 1, size (h)
 call vamp_create_history (h(i), ndim, verbose)
 end do
 end subroutine vamp_create_history_v
368e <Implementation of vamp procedures 76d>+≡
 pure subroutine vamp_create_history_a (h, ndim, verbose)
 type(vamp_history), dimension(:, :), intent(inout) :: h
 integer, intent(in), optional :: ndim
 logical, intent(in), optional :: verbose
 integer :: i
 do i = 1, size (h, dim=2)

```

```

 call vamp_create_history_v (h(:,i), ndim, verbose)
 end do
end subroutine vamp_create_history_a

369a <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_terminate_history_v (h)
 type(vamp_history), dimension(:), intent(inout) :: h
 integer :: i
 do i = 1, size (h)
 call vamp_terminate_history (h(i))
 end do
end subroutine vamp_terminate_history_v

369b <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_copy_history_v (lhs, rhs)
 type(vamp_history), dimension(:), intent(out) :: lhs
 type(vamp_history), dimension(:), intent(in) :: rhs
 integer :: i
 do i = 1, size (rhs)
 call vamp_copy_history (lhs(i), rhs(i))
 end do
end subroutine vamp_copy_history_v

369c <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_copy_history_a (lhs, rhs)
 type(vamp_history), dimension(:, :), intent(out) :: lhs
 type(vamp_history), dimension(:, :), intent(in) :: rhs
 integer :: i
 do i = 1, size (rhs, dim=2)
 call vamp_copy_history_v (lhs(:,i), rhs(:,i))
 end do
end subroutine vamp_copy_history_a

369d <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_delete_history_v (h)
 type(vamp_history), dimension(:), intent(inout) :: h
 integer :: i
 do i = 1, size (h)
 call vamp_delete_history (h(i))
 end do
end subroutine vamp_delete_history_v

369e <Implementation of vamp procedures 76d>+≡
pure subroutine vamp_delete_history_a (h)
 type(vamp_history), dimension(:, :), intent(inout) :: h
 integer :: i

```

```

do i = 1, size (h, dim=2)
 call vamp_delete_history_v (h(:,i))
end do
end subroutine vamp_delete_history_a
370 ⟨Implementation of vamp procedures 76d⟩+≡
 _end_f90

```

# —N— IDEAS

## *N.1 Toolbox for Interactive Optimization*

*Idea:* Provide a OpenGL interface to visualize the grid optimization.

*Motivation:* Would help multi channel developers.

*Implementation:* Coding is straightforward, but interface design is hard.

## *N.2 Partially Non-Factorized Importance Sampling*

*Idea:* Allow non-factorized grid optimization in two- or three-dimensional subspaces.

*Motivation:* Handle nastiest subspaces. Non-factorized approaches are impossible in higher than three dimensions (and probably only realistic in two dimensions), but there are cases that are best handled by including non-factorized optimization in two dimensions.

*Implementation:* The problem is that the present `vamp_sample_grid0` can't accomodate this, because other auxiliary information has to be collected, but a generalization is straightforward. Work has to start from an extended `divisions` module.

## *N.3 Correlated Importance Sampling (?)*

*Idea:* Is it possible to include *some* correlations in a mainly factorized context?

*Motivation:* Would be nice ...

*Implementation:* First, I have to think about the maths ...

## *N.4 Align Coordinate System (i.e. the grid) with Singularities (or the hot region)*

*Idea:* Solve **vegas**' nastiest problem by finding the direction(s) along which singularities are aligned.

*Motivation:* Automatically choose proper coordinate system in generator generators and separate wild and smooth directions.

*Implementation:* Diagonalize the covariance matrix  $\text{cov}(x_i x_j)$  to find better axes. Caveats:

- damp rotations (rotate only if eigenvalues are spread out sufficiently).
- be careful about blow up of the integration volume, which is  $V' = V d^{d/2}$  in the worst case for hypercubes and can be even worse for stretched cubes. (An adaptive grid can help, since we will have more smooth directions!)

*Maybe* try non-linear transformations as well.

## *N.5 Automagic Multi Channel*

*Idea:* Find and extract one singularity after the other.

*Motivation:* Obvious.

*Implementation:* Either use multiple of **vegas**'  $p(x)$  for importance sampling. Or find hot region(s) and split the integration region (à la signal/background).

# —O—

## CROSS REFERENCES

### *O.1 Identifiers*

BUFFER\_SIZE: [58d](#), [59a](#), [105d](#), [107a](#), [107b](#), [107c](#), [263c](#)  
DEFAULT\_BUFFER\_SIZE: [250d](#), [259d](#), [259f](#), [272c](#), [273b](#)  
DEFAULT\_SEED: [253a](#), [253c](#)  
EXC\_DEFAULT: [244a](#), [244e](#)  
EXC\_ERROR: [88c](#), [243c](#), [244a](#), [244c](#)  
EXC\_FATAL: [49d](#), [50](#), [51a](#), [89a](#), [95a](#), [243c](#), [244c](#)  
EXC\_INFO: [94a](#), [166b](#), [176a](#), [243c](#), [244c](#)  
EXC\_NONE: [243b](#), [243c](#), [244c](#)  
EXC\_WARN: [88c](#), [96d](#), [103d](#), [119a](#), [121d](#), [133c](#), [135b](#), [135c](#), [166b](#), [173d](#), [174a](#),  
[177b](#), [243c](#), [244c](#)  
K: [250a](#), [250b](#), [250a](#), [251c](#), [251d](#), [250a](#), [251e](#), [252a](#), [252d](#), [253d](#), [254a](#), [254b](#),  
[254c](#), [254d](#), [254e](#), [254f](#), [255a](#), [255c](#), [256a](#), [256c](#), [257c](#), [257e](#), [257g](#), [250a](#),  
[257h](#), [258a](#), [258b](#), [258d](#), [259d](#), [259f](#)  
L: [250a](#), [250b](#), [251d](#), [251e](#), [254b](#), [254c](#), [254d](#), [254e](#), [254f](#), [255a](#), [255c](#), [256c](#),  
[257h](#), [258a](#)  
LIPS3: [222b](#), [224a](#), [224e](#), [225a](#), [226c](#), [227b](#), [228b](#), [229d](#), [230c](#), [231a](#), [231b](#),  
[231c](#), [232a](#), [232b](#), [233a](#), [233b](#), [233d](#), [324g](#), [325a](#), [325b](#), [331d](#)  
LIPS3\_m5i2a3: [224b](#), [224e](#), [225a](#), [225b](#), [226a](#), [229d](#), [230a](#), [230c](#), [231a](#), [231b](#),  
[231c](#), [232a](#), [232b](#), [233a](#), [233b](#)  
LIPS3\_momenta: [328e](#)  
LIPS3\_momenta\_massless: [329a](#)  
LIPS3\_s2\_t1\_angles: [328c](#), [330a](#)  
LIPS3\_s2\_t1\_angles\_massless: [328d](#), [330b](#)  
LIPS3\_unit: [328a](#), [329d](#), [330a](#)  
LIPS3\_unit\_massless: [328b](#), [329e](#), [330b](#)  
LIPS3\_unit\_to\_s2\_t1\_angles: [329f](#), [329g](#), [330a](#)  
LIPS3\_unit\_to\_s2\_t1\_angles\_m0: [329f](#), [329g](#), [330b](#)

M: [238a](#), [250c](#), [251d](#), [251e](#), [253d](#), [254f](#), [255a](#), [255g](#), [256c](#), [257h](#), [258a](#), [272e](#),  
[274e](#), [280e](#), [281a](#), [281c](#), [281d](#), [281e](#), [282a](#), [282b](#)  
MAX\_SEED: [253b](#), [253c](#)  
MAX\_UNIT: [265e](#), [266a](#), [310b](#), [310c](#)  
MIN\_NUM\_DIV: [45d](#)  
MIN\_UNIT: [265e](#), [266a](#), [310b](#), [310c](#)  
NAME\_LENGTH: [243b](#), [244a](#)  
NUM\_DIV\_DEFAULT: [76d](#), [78a](#), [78a](#)  
PI: [188b](#), [190](#), [191a](#), [191b](#), [221a](#), [221b](#), [222a](#), [223c](#), [224e](#), [225b](#), [226a](#), [241b](#),  
[284a](#), [285c](#), [287b](#), [325a](#), [325b](#), [330e](#), [331c](#), [331d](#), [332](#), [334](#), [340b](#)  
PRIMES: [309b](#), [309c](#)  
QUAD\_POWER: [91a](#), [91c](#)  
TAG\_GRID: [174b](#), [175a](#), [175b](#), [175e](#)  
TAG\_HISTORY: [175a](#), [175b](#), [175e](#)  
TAG\_INTEGRAL: [175a](#), [175b](#), [175e](#)  
TAG\_NEXT\_FREE: [175e](#)  
TAG\_STD\_DEFS: [175e](#)  
TT: [253b](#), [253f](#)  
ULP: [257b](#), [257d](#), [257e](#), [257f](#), [257h](#), [258a](#)  
VAMP\_MAX\_WASTE: [172](#), [173a](#), [173a](#)  
VAMP\_ROOT: [165b](#), [165c](#), [165d](#), [166a](#), [166b](#), [168a](#), [169c](#), [169d](#), [170c](#), [170d](#),  
[171a](#), [171b](#), [172](#), [173c](#), [174a](#), [174b](#), [175a](#), [175f](#), [176a](#), [176c](#), [177b](#), [178d](#),  
[179a](#), [180d](#), [181a](#), [181b](#), [181c](#), [182a](#), [182b](#), [182c](#), [182d](#)  
abs\_evec: [125b](#), [125c](#), [125d](#), [126b](#), [127b](#), [127c](#)  
accuracy: [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [94a](#), [100c](#), [117b](#), [137c](#),  
[138](#), [139c](#), [166b](#), [172](#), [176a](#)  
adaptive\_division: [56b](#), [57c](#), [363d](#), [363e](#), [364e](#)  
average: [105d](#), [107c](#), [131](#), [289b](#), [289c](#), [290a](#), [290d](#), [290e](#)  
avg\_chi2: [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [92](#), [93c](#), [100c](#),  
[103c](#), [104e](#), [105d](#), [107c](#), [117b](#), [120c](#), [122b](#), [137c](#), [138](#), [139c](#), [158b](#), [159b](#),  
[162b](#), [166b](#), [172](#), [175f](#)  
\_begin\_f90: [355a](#), [355c](#), [356a](#), [356b](#), [356c](#), [358b](#), [358h](#), [366a](#)  
boost\_many\_momentum: [322b](#), [322c](#), [324b](#)  
boost\_many\_velocity: [322b](#), [322c](#), [323b](#), [324b](#)  
boost\_momentum: [224e](#), [322b](#), [322c](#), [326c](#), [326d](#), [332](#)  
boost\_one\_momentum: [322b](#), [322c](#), [324a](#)  
boost\_one\_velocity: [322b](#), [322c](#), [323a](#), [323b](#)  
boost\_velocity: [322b](#), [322c](#), [324a](#)  
buffer\_end: [258c](#), [258e](#), [259d](#), [259f](#), [261g](#), [263c](#), [263d](#), [267d](#), [268b](#), [269b](#),  
[269d](#), [272a](#), [272d](#), [272e](#), [273a](#), [273b](#), [273c](#), [274b](#), [274c](#), [274e](#), [275a](#), [275b](#),  
[275c](#), [275d](#), [276a](#), [276b](#), [276c](#), [278h](#), [279b](#)

calls: 82, 82, 82, 82, 82, 24d, 82, 82, 82, 82, 76a, [82](#), 83a, 87b, 96b, 98a, 103c, 104c, 104d, 104e, 105d, 107c, 122b, 133a, 137c, 138, 139c, 141c, 143b, 148b, 149b, 154c, 156b, 158b, 159b, 160, 162b, 234, 237, 238a, 238b, 239  
calls\_per\_cell: 76a, [82](#), 83a, 86c, 88a, 95a, 96b, 96c, 141c, 143b, 148b, 149b, 154c, 156b, 160  
cartesian\_to\_spherical: 336c, [337a](#)  
cartesian\_to\_spherical\_2: 336c, [336d](#), 337a, 337b  
cartesian\_to\_spherical\_cos: 191a, 338d, [339b](#)  
cartesian\_to\_spherical\_cos\_2: 193b, 338d, [339a](#), 339b, 339c  
cartesian\_to\_spherical\_cos\_j: 191b, 338d, [339c](#)  
cartesian\_to\_spherical\_j: 336c, [337b](#)  
cell: 85b, 43b, 85b, [85b](#), 85c, [86b](#), 86d, 361c  
check\_inverses: 192c, [193a](#), 196c  
check\_inverses3: 192c, [193b](#), 196c  
check\_jacobians: 192c, [192d](#), 196c, 207c, [207d](#), 214e  
check\_kinematics: 229c, [229d](#), 234  
cl: [129b](#), [129c](#), [130a](#), 130b  
clear\_exception: 24a, 24b, 24c, 24d, 100c, 117b, 210b, 211, 237, 238a, 238b, 239, 244d, [245a](#), 246, 358c, 358d, 358f  
clear\_integral\_and\_variance: 44b, [45a](#), 82, 85c, 361e, 362a, 362f  
cl\_num: [129b](#), [129c](#), [130a](#), 130b  
cl\_pos: [129b](#), [129c](#), [130a](#), 130b  
collect: 54a, 54b, [55a](#)  
constants: 188b, 218a, [241b](#), 284a, 286b, 322a, 327d, 331d, 332, 334  
coordinates: 188b, 192b, 196c, 198b, 198c, [334](#)  
copy\_array\_pointer: 51d, 52c, 59d, 60a, 60b, 60c, 68c, 95b, 156b, 160, 161b, 304d, [304e](#)  
copy\_division: 38a, 51a, 68c, 68c, 96e, 160, 359a, 359b, 359e, 360a  
copy\_history: 57e, [69b](#), 104e, 105d, 107c, 162b, 365b, 365c, 365d  
copy\_integer\_array2\_pointer: 304d, 304e, [305c](#)  
copy\_integer\_array\_pointer: 304d, 304e, [305a](#)  
copy\_raw\_state: 260c, 261f, 261g, [262a](#), 262b, 262c  
copy\_raw\_state\_to\_state: 261f, [262b](#)  
copy\_real\_array2\_pointer: 304d, 304e, [305d](#)  
copy\_real\_array\_pointer: 304d, 304e, [305b](#)  
copy\_state: 261f, [261g](#)  
copy\_state\_to\_raw\_state: 260b, 261f, [262c](#)  
cos\_theta: [125e](#), 126a, 126b, 193b, 325a, 325b, 326c, 326d, 327b, 330d, 330e, 335b, 336d, 338a, 339a, 339b  
create\_array\_pointer: 95b, 143b, 149b, 302b, [302c](#)



create\_division: 38a, [38b](#), 76d, 359a, 359b, 359c  
 create\_empty\_division: 38a, [39a](#), 97a, 144, 359a, 359b, 359d  
 create\_histogram: 210b, 211, [293b](#), [293c](#)  
 create\_histogram1: [293c](#), [293d](#), [293f](#)  
 create\_histogram2: [293c](#), [293d](#), [294a](#)  
 create\_integer\_array2\_pointer: 302b, 302c, [304b](#), 305c  
 create\_integer\_array\_pointer: 302b, 302c, [303c](#), 305a  
 create\_raw\_state\_from\_raw\_st: [259c](#), [259e](#), [259f](#), [260c](#)  
 create\_raw\_state\_from\_seed: [259c](#), [259d](#), [260a](#)  
 create\_raw\_state\_from\_state: [259c](#), [260b](#)  
 create\_real\_array2\_pointer: 302b, 302c, [304c](#), 305d  
 create\_real\_array\_pointer: 302b, 302c, [304a](#), 305b  
 create\_sample: 201, [203c](#), 215e  
 create\_state\_from\_raw\_state: [259c](#), [259f](#)  
 create\_state\_from\_seed: [259c](#), [259d](#)  
 create\_state\_from\_state: [259c](#), [259e](#)  
 ctest: [131](#)  
 debug\_division: [55c](#), [55d](#)  
 decode\_status: [347c](#), [347d](#), [347e](#), 350b  
 delete\_division: 38a, [69a](#), 144, 160, 161a, 359a, 359b, 360d  
 delete\_histogram: 210b, 211, [293b](#), [293c](#)  
 delete\_histogram1: [293c](#), [293d](#), [295c](#)  
 delete\_histogram2: [293c](#), [293d](#), [296a](#)  
 delete\_sample: 201, [203b](#), 215f  
 delta: [47a](#), [47a](#), [47a](#), [47c](#), 48a, 112, 192d, 287b  
 descr\_fmt: 61a, [61b](#), 62, 141c, [143a](#), 143b, 146a, 146b  
 destroy\_raw\_state: [260d](#), [260e](#), [261b](#)  
 destroy\_state: [260d](#), [260e](#), [261a](#)  
 determinant: 111c, 113b, 314a, [314b](#), 319d  
 diag: [315b](#), [319b](#), 319c  
 diagonalize\_real\_symmetric: 315b, 126b, 127a, 127b, 128c, 139c, 315a, [315b](#), [319d](#)  
 distribute: 15, 16, 52c, 54b, [54d](#), 100c  
 div\_history: [57d](#), 57f, 58c, 58d, 67c, 68a, 68b, 69b, 103c, 105d, 107c, [365d](#), [365e](#)  
 division: [37b](#), 38b, 39a, 39b, 39c, 43b, 44a, 44c, 44d, 44e, 45a, 45c, 48c, [49b](#), [49d](#), 50, 55b, 55d, 56a, 56c, 56d, 57a, 57b, 57c, 57f, 59d, 60a, 60b, 60c, 61a, 62, 63b, 63c, 63d, 64b, 65a, 65b, 66b, 66c, 67a, 68c, 69a, 76a, [37b](#), 94d, 97b, 359c, 359d, 359e, 360a, 360b, 360c, 360d, 361c, 361d, 362c, [362d](#), [362e](#), [362f](#), 363c, 363f, 364a, 364b, 364c, 364d, 364e, 365a, 365e  
 division\_efficiency: 59c, [60c](#)

[illegible]



handle\_exception: 24a, 24b, 24c, 24d, 119a, 210b, 211, 237, 238a, 238b, 239, 244b, 244c  
 histogram: 210b, 211, 292b, 293f, 294b, 295c, 296b, 297b, 298d  
 histogram2: 293a, 293a, 294a, 295a, 295b, 296a, 298e, 299  
 histograms: 192b, 198b, 207b, 292a  
 i: 15, 16, 17, 125b, 125b, 125b, 43b, 43c, 44a, 44c, 44d, 44e, 47a, 48a, 49d, 50, 51d, 52a, 52c, 54a, 54c, 54d, 55a, 55b, 55d, 56a, 58d, 61a, 62, 63d, 64b, 72a, 72d, 73c, 74a, 74c, 89d, 94d, 95b, 96b, 96c, 125b, 96e, 97a, 97b, 97c, 97d, 98a, 98c, 99b, 100a, 100c, 102b, 103b, 105d, 107b, 107c, 110c, 111a, 112, 125b, 125c, 125d, 125e, 126a, 126b, 127b, 127c, 128c, 129b, 130b, 130d, 131, 141c, 143b, 146a, 146b, 148b, 149b, 152a, 152c, 154c, 156a, 156b, 159a, 166b, 185c, 189c, 193a, 193b, 202a, 203a, 210b, 211, 215d, 215e, 234, 238b, 245b, 264c, 264e, 265a, 265c, 265e, 280d, 281a, 281c, 281d, 281e, 282a, 282b, 282c, 285a, 285b, 285c, 287c, 294b, 295b, 296b, 297b, 306d, 307a, 307b, 307c, 125b, 309b, 310c, 314b, 319a, 319b, 319d, 323b, 331d, 332, 335b, 336d, 338a, 339a, 357b, 357c, 357d, 357e, 358e, 358f, 362c, 362d, 362e, 366d, 366e, 366f, 367a, 367d, 367e, 367h, 368a, 368d, 368e, 369a, 369b, 369c, 369d, 369e  
 ia: 86d, 87c, 87d, 88a, 132d, 133a  
 ihp: 31b, 31b, 110c, 188b, 191a, 192a, 193a  
 inject\_division: 43a, 43b, 86d, 361a, 361b, 361c  
 inject\_division\_short: 43a, 44a, 133a, 361a, 361b, 361d  
 inside\_division: 56b, 56c, 87a, 363d, 363e, 364a  
 integer\_array\_fmt: 141c, 143a, 143b, 146a, 146b  
 integer\_array\_state: 276a, 278a  
 integer\_array\_stateless: 273c, 276a, 277c, 278b  
 integer\_array\_static: 277c, 278a  
 integer\_fmt: 61a, 61b, 62, 141c, 143a, 143b, 146a, 146b  
 integer\_state: 275b, 278a  
 integer\_stateless: 272a, 275b, 276e, 278b  
 integer\_static: 276e, 278a  
 integral: 91d, 91d, 91d, 91d, 23c, 24c, 24d, 91d, 91d, 91d, 91d, 91d, 91d, 37b, 38b, 39a, 40b, 44c, 45a, 51d, 91d, 52a, 52c, 54a, 55b, 55d, 56a, 57f, 60a, 91d, 61a, 62, 63a, 63d, 64b, 66b, 67a, 68c, 69a, 88c, 91d, 92, 93c, 100c, 103c, 104e, 105d, 107c, 117b, 120c, 122b, 137c, 138, 139c, 158b, 159b, 162b, 166b, 172, 175f, 194b, 195a, 195b, 196b, 208b, 209b, 210b, 211, 234, 237, 238a, 238b, 239  
 \_interface\_sv: 355a, 355c, 358d, 359b, 361b, 362a, 362b, 363b, 363e, 365c, 366c, 367c, 367g, 368c  
 \_interface\_sva: 355a, 355c, 357a, 359b, 368c  
 invariants\_from\_p: 224d, 224e, 229d, 230c, 231a, 232a, 232b

invariants\_from\_x: 224d, 225b, 229d, 230c, 231a, 231b, 231c, 232a, 232b, 233a, 233b  
 invariants\_to\_p: 224d, 225a, 229d, 230c, 231a, 231b, 231c, 232a, 232b, 233a, 233b  
 invariants\_to\_x: 224d, 226a, 229d, 230c, 231a, 232a, 232b  
 iteration: 14, 15, 17, 91d, 91d, 91d, 92, 100c, 103d, 117b, 119a, 120c, 121d, 166b, 172, 173d, 174a, 175a, 175b, 177b  
 iterations: 14, 15, 17, 91d, 91d, 91d, 91d, 91d, 91d, 91d, 91d, 58d, 91d, 100c, 105d, 107c, 117b, 136b, 136c, 166b, 172, 179b, 179c, 194b, 195a, 195b, 196c, 198c, 208b, 209a, 209b, 210b, 211, 214a, 214b, 214c, 214d, 215d, 215e, 234, 237, 238a, 238b, 239  
 iv: 124b, 125b, 125c, 125d, 125e, 126b, 127b, 127c  
 j: 31c, 49d, 50, 51c, 52b, 78c, 85b, 86b, 94d, 96c, 86b, 86b, 96e, 97b, 97c, 97d, 102b, 103a, 103b, 105d, 107c, 111c, 113b, 128c, 141c, 143b, 148b, 149b, 158b, 159b, 188b, 191b, 192a, 192d, 193b, 202a, 203a, 205b, 205c, 206a, 211, 215d, 215e, 251a, 251d, 251e, 252d, 253d, 254c, 254e, 254f, 256b, 256c, 257c, 257e, 257h, 306d, 307a, 307b, 307c, 312, 313d, 315b, 359c, 359d, 359e, 360a, 360b, 360c, 360d, 361c, 361d, 362c, 362d, 362e, 362f, 363c, 363f, 364a, 364b, 364c, 364d, 364e, 365a, 365d, 365e  
 jacobi: 76a, 83a, 83a, 86d, 96b, 98a, 133a, 141c, 143b, 148b, 149b, 154c, 156b, 160  
 jacobian: 31c, 31c, 31c, 110c, 111a, 111c, 113b, 222a, 223b, 223c, 224b, 224c, 224e, 225a, 225b, 226a, 226c, 227b, 228b, 229d, 230a, 232a, 232b, 233a, 233b, 324g, 325a, 325b, 328a, 328b, 328c, 328d, 328e, 329a, 329d, 329e, 331d, 335b, 336b, 336d, 337b, 338a, 338c, 339a, 339c  
 jacobi\_rotation: 318a, 318b, 318c, 318d  
 join\_division: 49c, 50, 97c  
 k: 47a, 47c, 48a, 86b, 86c, 226c, 227b, 228b, 229a, 233d, 287b, 330d, 47a, 330e, 331a  
 kinematics: 218a, 234, 322a, 327d, 331d, 332  
 lambda: 221a, 223c, 224e, 225b, 226a, 319d, 324c, 324d, 325a, 326c, 331d  
 last: 252c, 258c, 258e, 261g, 262d, 263c, 263d, 267d, 268b, 269b, 269d, 272a, 272b, 273b, 273b, 272d, 272e, 273a, 273b, 273c, 273d, 273b, 274b, 274c, 274d, 274e, 275a, 275b, 275c, 275d, 276a, 276b, 276c, 278h, 279b  
 lcm: 52b, 308d, 309a, 356f, 357a, 357c, 357e  
 linalg: 75a, 234, 311a, 315b, 319d  
 local\_avg\_chi2: 91d, 93c, 100c, 103d, 117b, 119a, 121d, 166b, 172, 173d, 174a, 175f, 177b  
 local\_integral: 91d, 93c, 94a, 100c, 103d, 117b, 121d, 166b, 172, 174a, 175f, 176a, 177b  
 local\_std\_dev: 91d, 93c, 94a, 100c, 103d, 117b, 121d, 166b, 172, 174a,

[175f](#), [176a](#), [177b](#)  
 logical\_fmt: [61a](#), [61b](#), [62](#), [141c](#), [143a](#), [143b](#)  
 lorentzian: [188b](#), [189b](#), [189c](#)  
 lorentzian\_normalized: [188b](#), [189a](#), [189b](#)  
 lu\_decompose: [311b](#), [312](#), [314b](#), [319d](#)  
 luxury\_state: [279a](#), [280c](#)  
 luxury\_state\_integer: [279a](#), [279b](#), [279c](#), [280c](#)  
 luxury\_stateless: [278h](#), [279b](#), [279e](#), [280a](#)  
 luxury\_state\_real: [279c](#), [280c](#)  
 luxury\_static: [279d](#), [280c](#)  
 luxury\_static\_integer: [279d](#), [279e](#), [279f](#), [280c](#)  
 luxury\_static\_real: [279f](#), [280c](#)  
 m: [39c](#), [40a](#), [46a](#), [46c](#), [40a](#), [47a](#), [47c](#), [48a](#), [57f](#), [58d](#), [308e](#), [308f](#), [40a](#), [309a](#),  
[327c](#), [332](#), [357b](#), [357c](#), [357d](#), [357e](#)  
 map\_domain: [76d](#), [78b](#), [78c](#)  
 marshal\_div\_history: [67b](#), [67c](#), [158b](#)  
 marshal\_div\_history\_size: [67b](#), [68a](#), [158b](#), [159a](#)  
 marshal\_division: [66a](#), [66b](#), [154c](#)  
 marshal\_division\_size: [66a](#), [66c](#), [154c](#), [156a](#)  
 marshal\_raw\_state: [267b](#), [267c](#), [267d](#), [268c](#), [269a](#), [269b](#), [270a](#)  
 marshal\_raw\_state\_size: [267b](#), [267c](#), [268a](#), [268d](#), [269c](#), [270b](#)  
 marshal\_state: [267b](#), [267c](#), [267d](#), [268b](#), [269b](#)  
 marshal\_state\_size: [267b](#), [267c](#), [268a](#), [269c](#)  
 massless\_isotropic\_decay: [229a](#), [330c](#), [330d](#)  
 midpoint: [296b](#), [297b](#), [298a](#), [298b](#), [299](#)  
 midpoint1: [298b](#), [298c](#), [298d](#)  
 midpoint2: [298b](#), [298c](#), [298e](#)  
 more\_pancake\_than\_cigar: [124b](#), [124c](#), [125a](#)  
 mpi90: [163c](#), [198c](#), [216a](#), [216b](#), [234](#), [341a](#), [343a](#)  
 mpi90\_abort: [341b](#), [342c](#), [342e](#), [343a](#)  
 mpi90\_broadcast: [17](#), [166b](#), [172](#), [175f](#), [176a](#), [185b](#), [186a](#), [198c](#), [216b](#), [234](#),  
[238b](#), [350d](#), [350e](#)  
 mpi90\_broadcast\_double: [350e](#), [351b](#)  
 mpi90\_broadcast\_double\_array: [350e](#), [351b](#), [352b](#), [353a](#), [353d](#)  
 mpi90\_broadcast\_double\_array2: [350e](#), [353a](#)  
 mpi90\_broadcast\_double\_array3: [350e](#), [353d](#)  
 mpi90\_broadcast\_integer: [350e](#), [351a](#)  
 mpi90\_broadcast\_integer\_array: [350e](#), [351a](#), [351d](#), [352d](#), [353c](#)  
 mpi90\_broadcast\_integer\_array2: [350e](#), [352d](#)  
 mpi90\_broadcast\_integer\_array3: [350e](#), [353c](#)  
 mpi90\_broadcast\_logical: [350e](#), [351c](#)

mpi90\_broadcast\_logical\_array: [350e](#), [351c](#), [352c](#), [353b](#), [354](#)  
 mpi90\_broadcast\_logical\_array2: [350e](#), [353b](#)  
 mpi90\_broadcast\_logical\_array3: [350e](#), [354](#)  
 mpi90\_finalize: [198c](#), [216b](#), [234](#), [341b](#), [342d](#)  
 mpi90\_init: [198c](#), [216b](#), [234](#), [341b](#), [342a](#)  
 mpi90\_print\_error: [341b](#), [342b](#), [342c](#), [343a](#)  
 mpi90\_rank: [17](#), [165b](#), [165d](#), [166a](#), [166b](#), [168a](#), [169c](#), [169d](#), [170c](#), [170d](#),  
[171a](#), [171b](#), [172](#), [178d](#), [179a](#), [180d](#), [181a](#), [181b](#), [181c](#), [182a](#), [182b](#), [182c](#),  
[182d](#), [185b](#), [186a](#), [198c](#), [216b](#), [234](#), [341b](#), [344a](#)  
 mpi90\_receive: [175b](#), [183c](#), [187](#), [344b](#), [346d](#)  
 mpi90\_receive\_double: [346d](#), [347a](#)  
 mpi90\_receive\_double\_array: [346d](#), [347a](#), [348a](#), [348c](#)  
 mpi90\_receive\_double\_array2: [346d](#), [348c](#)  
 mpi90\_receive\_double\_pointer: [349a](#), [350c](#)  
 mpi90\_receive\_integer: [346c](#), [346d](#)  
 mpi90\_receive\_integer\_array: [346c](#), [346d](#), [347b](#), [348b](#)  
 mpi90\_receive\_integer\_array2: [346d](#), [348b](#)  
 mpi90\_receive\_integer\_pointer: [349a](#), [349b](#)  
 mpi90\_receive\_pointer: [349a](#), [349a](#), [349a](#), [184b](#), [344b](#), [349a](#)  
 mpi90\_send: [175a](#), [183b](#), [184a](#), [186c](#), [344b](#), [344c](#)  
 mpi90\_send\_double: [344c](#), [344e](#)  
 mpi90\_send\_double\_array: [344c](#), [344e](#), [345c](#), [346a](#)  
 mpi90\_send\_double\_array2: [344c](#), [346a](#)  
 mpi90\_send\_integer: [344c](#), [344d](#)  
 mpi90\_send\_integer\_array: [344c](#), [344d](#), [345a](#), [345d](#)  
 mpi90\_send\_integer\_array2: [344c](#), [345d](#)  
 mpi90\_size: [17](#), [166b](#), [172](#), [198c](#), [216b](#), [234](#), [341b](#), [343c](#)  
 mpi90\_status: [183c](#), [184b](#), [187](#), [346b](#), [346c](#), [347a](#), [347b](#), [347e](#), [348a](#), [348b](#),  
[348c](#), [349b](#), [350c](#)  
 multi\_channel: [194a](#), [195a](#), [196c](#), [208a](#), [209a](#), [214c](#)  
 multi\_channel\_generator: [210a](#), [211](#), [214d](#)  
 ndim: [76d](#), [78c](#), [80](#), [82](#), [85a](#), [85b](#), [91a](#), [91d](#), [94d](#), [95b](#), [104c](#), [127b](#), [127c](#),  
[143b](#), [144](#), [149b](#), [154c](#), [156a](#), [156b](#), [158b](#), [159a](#), [159b](#), [160](#), [315b](#), [76d](#), [76d](#),  
[318a](#), [368d](#), [368e](#)  
 ng: [37b](#), [38b](#), [39b](#), [39c](#), [42](#), [49d](#), [50](#), [42](#), [51a](#), [42](#), [51d](#), [52b](#), [52c](#), [57b](#), [57d](#),  
[57f](#), [58d](#), [61a](#), [62](#), [63d](#), [64b](#), [66b](#), [67a](#), [67c](#), [68b](#), [68c](#), [69b](#), [80](#), [42](#), [82](#), [83d](#),  
[102b](#), [103b](#), [360b](#), [360c](#)  
 norm: [125e](#), [126b](#), [202a](#), [202b](#)  
 num\_calls: [79c](#), [79c](#), [79c](#), [79c](#), [79c](#), [79c](#), [24a](#), [24c](#), [79c](#), [79c](#), [79c](#), [79c](#), [79c](#),  
[79c](#), [79c](#), [79c](#), [79c](#), [79c](#), [79c](#), [79c](#), [49d](#), [51a](#), [51d](#), [52c](#), [76a](#), [76d](#), [79a](#), [79c](#),  
[80](#), [81b](#), [82](#), [96b](#), [110a](#), [114b](#), [115a](#), [115c](#), [116b](#), [116d](#), [79c](#), [79c](#), [117b](#), [120a](#),

136c, 141c, 143b, 146a, 146b, 148b, 149b, 152a, 152c, 154c, 156b, 160,  
 161b, 162a, 165b, 165d, 166a, 170c, 170d, 171a, 172, 179c, 186a  
 num\_cells: 80, 82, 83a, 94d, 96b  
 num\_div: 42, 42, 42, 39c, 40b, 42, 42, 42, 42, 47a, 42, 49d, 50, 51c, 51d,  
 52b, 57d, 57f, 58b, 58d, 62, 63a, 64b, 66b, 67a, 67c, 68b, 69b, 76a, 76d,  
 78d, 79a, 79c, 80, 81b, 82, 91a, 42, 94d, 95b, 97a, 114b, 115c, 116b, 116d,  
 137c, 138, 139c, 141c, 143b, 148b, 149b, 154c, 156b, 160, 161a, 165b,  
 165d, 166a, 170c, 170d, 171a  
 numeric\_jacobian: 113a, 113b  
 object: 174b, 175a, 175b, 175c, 175d  
 one\_to\_two\_massive: 326a, 326b, 326c  
 one\_to\_two\_massless: 326a, 326b, 326d  
 on\_shell: 325a, 325b, 326c, 327a, 327c  
 outer\_product: 83e, 87c, 88a, 308b, 308c, 312  
 phase\_space: 222b, 223d, 226c, 228b, 229d, 327d  
 phase\_space\_volume: 220b, 229a, 234, 331b, 331c  
 phi: 31a, 31a, 31a, 110c, 111a, 111c, 112, 113b, 133d, 179a, 188b, 190,  
 191a, 192a, 192d, 193a, 193b, 201, 205c, 206b, 207d, 211, 221a, 221b,  
 222b, 224a, 224b, 224e, 225a, 225b, 226a, 230a, 325a, 325b, 326c, 326d,  
 327b, 328c, 328d, 330d, 330e, 331d, 335b, 336a, 336b, 336d, 337a, 338a,  
 338b, 338c, 339a, 339b  
 phi1: 230b, 231b, 233d  
 phi12: 230b, 230c, 233d, 234  
 phi2: 230a, 230b, 231c, 233d  
 phi21: 230b, 231a, 233d, 234  
 print\_LIPS3\_m5i2a3: 229c, 230a  
 print\_history: 58c, 58d, 59b, 105d  
 print\_history,: 59b  
 print\_results: 196a, 196b, 196c  
 probabilities: 57f, 58a, 58b  
 probability: 48b, 48c, 89c, 363d, 363e, 363f  
 psi: 201, 204, 205c  
 quadrupole: 79c, 79c, 79c, 79c, 79c, 76a, 76d, 79a, 79c, 80, 81b, 91a, 96a,  
 114b, 115c, 116b, 116d, 137c, 138, 139c, 141c, 143b, 148b, 149b, 154c,  
 156b, 160, 165b, 165d, 166a, 170c, 170d, 171a  
 quadrupole\_division: 49a, 49b, 91a, 363d, 363e, 365a  
 r: 43b, 44a, 86d, 87d, 112, 126a, 126b, 127a, 127b, 132d, 133a, 133d, 134b,  
 135b, 135c, 189b, 190, 191a, 193b, 247, 249f, 272a, 272b, 272e, 273a, 275b,  
 275c, 275d, 276e, 277a, 277b, 330d, 331a, 331d, 335b, 336a, 336b, 336d,  
 337a, 338a, 338b, 338c, 339a, 339b, 361c, 361d  
 raise\_exception: 49d, 50, 51a, 88c, 89a, 94a, 95a, 103d, 119a, 121d,



[133c](#), [135b](#), [135c](#), [166b](#), [173d](#), [174a](#), [176a](#), [177b](#), [244d](#), [244e](#), [245b](#), [358c](#),  
[358d](#), [358e](#)  
random\_LIPS3: [329b](#), [329c](#)  
random\_LIPS3\_unit: [329b](#), [329c](#), [329d](#)  
random\_LIPS3\_unit\_massless: [329b](#), [329c](#), [329e](#)  
read\_division: [60d](#), [60e](#), [143b](#)  
read\_division\_name: [60d](#), [60e](#), [63c](#)  
read\_division\_raw: [60d](#), [60e](#), [149b](#)  
read\_division\_raw\_name: [60d](#), [60e](#), [65b](#)  
read\_division\_raw\_unit: [60d](#), [60e](#), [64b](#)  
read\_division\_unit: [60d](#), [60e](#), [62](#), [63c](#), [65b](#)  
read\_grid\_name: [140a](#), [141a](#), [145d](#), [180a](#), [180c](#), [181c](#)  
read\_grid\_raw\_name: [140b](#), [141b](#), [151b](#)  
read\_grid\_raw\_unit: [140b](#), [141b](#), [149b](#), [151b](#), [152c](#)  
read\_grids\_name: [140a](#), [141a](#), [148a](#), [181d](#), [181f](#), [182d](#)  
read\_grids\_raw\_name: [140b](#), [141b](#), [154a](#)  
read\_grids\_raw\_unit: [140b](#), [141b](#), [152c](#), [154a](#)  
read\_grids\_unit: [140a](#), [141a](#), [146b](#), [148a](#), [181d](#), [181f](#), [182b](#)  
read\_grid\_unit: [140a](#), [141a](#), [143b](#), [145d](#), [146b](#), [180a](#), [180c](#), [181a](#)  
read\_raw\_state\_name: [263a](#), [263b](#), [267a](#)  
read\_raw\_state\_unit: [263a](#), [263b](#), [263d](#), [264b](#), [267a](#)  
read\_state\_array: [263d](#), [264b](#), [264e](#), [264f](#), [265c](#), [265d](#)  
read\_state\_name: [263a](#), [263b](#), [266e](#)  
read\_state\_unit: [263a](#), [263b](#), [263d](#), [266e](#)  
real\_array\_state: [276b](#), [276c](#), [278a](#), [278e](#)  
real\_array\_stateless: [274e](#), [275a](#), [276b](#), [276c](#), [277d](#), [277e](#), [278b](#), [278f](#)  
real\_array\_static: [277d](#), [277e](#), [278a](#), [278e](#)  
real\_state: [275c](#), [275d](#), [278a](#), [278e](#)  
real\_stateless: [272e](#), [273a](#), [275c](#), [275d](#), [277a](#), [277b](#), [278b](#), [278f](#)  
real\_static: [277a](#), [277b](#), [278a](#), [278e](#)  
rebin: [39c](#), [45c](#), [47a](#), [47b](#)  
rebinning\_weights: [40a](#), [45c](#), [46a](#), [46b](#), [49b](#), [57f](#)  
record\_efficiency: [44b](#), [44e](#), [87c](#), [133a](#), [361f](#), [362b](#), [362e](#)  
record\_integral: [44b](#), [44c](#), [87c](#), [361e](#), [362a](#), [362c](#)  
record\_variance: [44b](#), [44d](#), [88a](#), [361e](#), [362a](#), [362d](#)  
refine\_division: [45b](#), [45c](#), [91a](#), [91b](#), [363a](#), [363b](#), [363c](#)  
reshape\_division: [38a](#), [39c](#), [82](#), [359a](#), [359b](#), [360c](#)  
rigid\_division: [56b](#), [57b](#), [82](#), [83d](#), [85b](#), [96b](#), [363d](#), [363e](#), [364d](#)  
s\_buffer: [251f](#), [272c](#), [273b](#), [276e](#), [277a](#), [277b](#), [277c](#), [277d](#), [277e](#), [279d](#),  
[279e](#), [279f](#)  
s\_buffer\_end: [272c](#), [273b](#), [276e](#), [277a](#), [277b](#), [277c](#), [277d](#), [277e](#), [279e](#)





sum\_chi2: 76a, 79a, 88c, 92, 95c, 98a, 110a, 114b, 115c, 120a, 120c, 141c,  
 143b, 146a, 146b, 148b, 149b, 152a, 152c, 154c, 156b, 160, 161b, 186a  
 sum\_division: 49c, 51b, 55b, 97d  
 sum\_f: 86c, 87c, 87d, 88a  
 sum\_f2: 86c, 87c, 87d, 88a  
 sum\_integral: 76a, 79a, 88c, 92, 95c, 98a, 110a, 114b, 115c, 120a, 120c,  
 141c, 143b, 146a, 146b, 148b, 149b, 152a, 152c, 154c, 156b, 160, 161b,  
 186a  
 summarize\_division: 57e, 57f, 104e, 365b, 365c, 365e  
 sum\_weights: 76a, 79a, 83e, 84b, 88c, 92, 95c, 98a, 110a, 114b, 115c, 120a,  
 120c, 141c, 143b, 146a, 146b, 148b, 149b, 152a, 152c, 154c, 156b, 160,  
 161b, 186a  
 sum\_x: 88a  
 sum\_xx: 88a  
 surface: 189b, 340a, 340b  
 s\_virginal: 251f, 252a, 256a, 276d  
 swap: 223a, 305e, 306a, 306d, 307b, 307c, 312, 313d, 356c  
 swap\_integer: 305e, 306a, 306b  
 swap\_integer\_array: 356b, 356c, 356d  
 swap\_real: 305e, 306a, 306c  
 swap\_real\_array: 356b, 356c, 356e  
 tao\_random\_copy: 249a, 259e, 261c, 271b  
 tao\_random\_create: 16, 24a, 196c, 198c, 213, 216b, 234, 259a, 248g, 248h,  
 249e, 249f, 259a, 271b, 281d  
 tao\_random\_destroy: 248i, 260d, 271b  
 tao\_random\_flush: 249b, 259d, 259f, 262b, 262d, 271b  
 tao\_random\_luxury: 248d, 248e, 248f, 279g, 271b, 278h, 279g, 281a  
 tao\_random\_marshal: 267b, 267c, 282b  
 tao\_random\_marshal\_size: 267b, 267c, 282b  
 tao\_random\_number: 277f, 86d, 112, 131, 133a, 133c, 134b, 135b, 135c,  
 193a, 193b, 215e, 229d, 234, 247, 248b, 249e, 249f, 271b, 277f, 278d, 281a,  
 281c, 281d, 281e, 282a, 282b, 319d, 329d, 329e, 331d, 332  
 tao52\_random\_numbers: 271a, 283  
 tao\_random\_numbers: 270d, 23b, 75a, 270d, 270d, 131, 163c, 192b, 196c,  
 198b, 198c, 207b, 215c, 218a, 234, 270d, 271a, 280d, 282c, 283, 319d,  
 327d, 331d, 332  
 tao\_random\_raw\_state: 248a, 258b, 258b, 258b, 249e, 249f, 258b, 252b,  
258b, 258c, 258d, 258e, 259f, 260a, 260b, 260c, 261b, 262a, 262b, 262c,  
 264a, 264b, 266d, 267a, 268c, 268d, 269a, 270a, 270b, 270c  
 tao\_random\_read: 249c, 263a, 271b, 282a  
 tao\_random\_seed: 196c, 198c, 213, 216b, 234, 255d, 248c, 253c, 255d,

271b, 255d, 276d, 281a, 281c, 282a, 282b, 319d  
 tao\_random\_state: 14, 15, 16, 17, 258c, 23c, 258c, 258c, 85a, 258c, 91d, 100c, 112, 117b, 132d, 133d, 136b, 136c, 137c, 138, 139c, 166b, 172, 178d, 179a, 179b, 179c, 192d, 193a, 193b, 194b, 195a, 196c, 198c, 207d, 208b, 209a, 210b, 211, 215d, 229d, 234, 258c, 258c, 248a, 258c, 258c, 258c, 249e, 249f, 252c, 258c, 258e, 258c, 259d, 259e, 259f, 260b, 261a, 261g, 262b, 262c, 262d, 263c, 263d, 266c, 266e, 267d, 268a, 268b, 269b, 269c, 269d, 275b, 275c, 275d, 276a, 276b, 276c, 279a, 279b, 279c, 280d, 282c, 329d, 329e  
 tao\_random\_test: 249d, 271b, 280d, 282c, 283  
 tao\_random\_unmarshal: 267b, 267c, 282b  
 tao\_random\_write: 249c, 262e, 271b, 282a  
 tao\_test: 283  
 two\_to\_three\_massless: 324e, 324f, 325b  
 unit: 61a, 62, 63b, 63c, 63d, 64b, 65a, 65b, 73a, 73c, 105d, 107b, 107c, 126a, 127a, 141c, 143b, 145c, 145d, 146a, 146b, 147, 148a, 148b, 149b, 151a, 151b, 152a, 152c, 153, 154a, 180d, 181a, 182a, 182b, 249c, 263c, 263d, 264a, 264b, 264c, 264e, 265a, 265c, 265e, 266c, 266d, 266e, 267a, 296b, 297b, 299, 310c, 315b, 319a, 319c, 319d, 330a, 330b  
 unmarshal\_div\_history: 67b, 68b, 159b  
 unmarshal\_division: 66a, 67a, 156b  
 unmarshal\_raw\_state: 267b, 267c, 268b, 269a, 269d, 270c  
 unmarshal\_state: 267b, 267c, 268b, 269d  
 utils: 37a, 75a, 131, 163c, 218a, 234, 292a, 302a, 311a, 319d  
 value\_spread: 289b, 290b, 290e  
 value\_spread\_percent: 49b, 57f, 290c, 290e  
 vamp: 2, 75b, 75b, 75b, 23b, 75b, 58d, 70a, 70c, 75a, 75b, 88c, 105d, 107c, 131, 75b, 75b, 163b, 75b, 75b, 75b, 188b, 192b, 196c, 198c, 201, 207a, 215c, 218a, 244e  
 vamp\_apply\_equivalences: 89d, 89d, 117b  
 vamp\_average\_iterations: 93a, 91d, 93a, 93b, 100c, 117b, 119a, 121b, 166b, 173d, 174a, 177b  
 vamp\_average\_iterations\_grid: 92, 93a, 93b  
 vamp\_average\_iterations\_grids: 120c, 121a, 121b  
 vamp\_broadcast\_grids: 183a, 186a  
 vamp\_broadcast\_many\_grids: 185a, 185c  
 vamp\_broadcast\_one\_grid: 185a, 185b, 185c  
 vamp\_check\_jacobian: 112, 111b, 112, 192d, 207d  
 vamp\_copy\_grid: 160, 76b, 160, 161b, 367b, 367c, 367d  
 vamp\_copy\_grids: 161b, 114a, 161b, 367f, 367g, 367h  
 vamp\_copy\_history: 162b, 104a, 162b, 368b, 368c, 369b

[illegible]

141c, 143b, 145c, 145d, 148b, 149b, 151a, 151b, 154c, 156a, 156b, 160,  
 161a, 165b, 165d, 166a, 166b, 168a, 178d, 179b, 180d, 181a, 181b, 181c,  
 183b, 183c, 184a, 184b, 185b, 185c, 188b, 189c, 192a, 194b, 201, 203a,  
 206b, 208b, 210b, 218a, 226c, 227b, 228b, 229a, 233d, 234, 366d, 366e,  
 366f, 367a, 367d, 367e  
 vamp\_grids: 110a, 110a, 110a, 110a, 110a, 110a, 110a, 110a, 110a, 110a,  
 110a, 110a, 110a, 110a, 89d, 91b, 110a, 110a, 114b, 115a, 115c, 116b,  
 116d, 117b, 120a, 120c, 121c, 122b, 133d, 136c, 146a, 146b, 147, 148a,  
 152a, 152c, 153, 154a, 161b, 162a, 169e, 169f, 170c, 170d, 171a, 171b,  
 171c, 172, 179a, 179c, 182a, 182b, 182c, 182d, 195a, 209a, 211, 234, 367h,  
 368a  
 vamp\_grid\_type: 22, 31c, 70b, 70c, 75a, 75b, 122d  
 vamp\_history: 103c, 103c, 91d, 100c, 103c, 104c, 104d, 104e, 105d, 107b,  
 107c, 117b, 122b, 136b, 136c, 137c, 138, 139c, 158b, 159a, 159b, 162b,  
 163a, 166b, 169c, 169d, 172, 179b, 179c, 186c, 187, 194b, 195b, 208b,  
 209b, 210b, 211, 234, 368d, 368e, 369a, 369b, 369c, 369d, 369e  
 vampi: 164a, 164a, 164a, 164a, 163b, 163c, 164a, 164a, 164a, 198b, 198c,  
 216a, 216b, 234  
 vamp\_integrate\_grid: 137a, 137b, 137c, 138, 139c  
 vamp\_integrate\_region: 137a, 137b, 138  
 vamp\_integratex\_region: 139a, 139b, 139c  
 vampi\_tests: 198b, 198c  
 vamp\_join\_grid: 94b, 15, 17, 94b, 94b, 94c, 98b, 100c, 166b  
 vamp\_join\_grid\_multi: 94b, 94c, 100a  
 vamp\_join\_grid\_single: 94b, 94c, 97b, 100a  
 vamp\_marshall\_grid: 154c, 29, 154b, 154c, 183b, 184a, 185b  
 vamp\_marshall\_grid\_size: 156a, 156a, 29, 154b, 156a, 183b, 184a, 185b  
 vamp\_marshall\_history: 158b, 158a, 158b, 186c  
 vamp\_marshall\_history\_size: 159a, 158a, 159a, 186c  
 vamp\_multi\_channel: 110c, 110b, 110c, 188b, 192a  
 vamp\_multi\_channel0: 111a, 110b, 111a, 201, 206b  
 vamp\_next\_event\_multi: 132b, 132c, 133d, 178a, 178c, 179a  
 vamp\_next\_event\_single: 132b, 132c, 132d, 133d, 178a, 178c, 178d  
 vamp\_nullify\_covariance: 84a, 83c, 84a, 117b, 173b, 177b, 366b, 366c,  
 366e  
 vamp\_nullify\_variance: 84c, 83c, 84c, 117b, 173b, 177b, 366b, 366c,  
 367a  
 vamp\_parallel\_mpi: 163c, 163c, 163c, 163c, 164a  
 vamp\_print\_covariance: 128b, 128c  
 vamp\_print\_histories: 105b, 105c, 107b, 168b, 169b, 169d  
 vamp\_print\_history: 105b, 105b, 105c, 168b, 169a, 169b, 194b, 195b,

[208b](#), [209b](#), [210b](#), [211](#), [237](#), [238a](#), [238b](#), [239](#)  
vamp\_print\_one\_history: [105b](#), [105c](#), [105d](#), [107b](#), [168b](#), [169b](#), [169c](#)  
vamp\_probability: [89b](#), [89c](#), [110c](#), [218a](#), [233d](#)  
vamp\_read\_grid: [141a](#), [140a](#), [141a](#), [143b](#), [180a](#), [180b](#), [180c](#)  
vamp\_read\_grid\_raw: [140b](#), [141b](#)  
vamp\_read\_grids: [141a](#), [140a](#), [141a](#), [146b](#), [181d](#), [181e](#), [181f](#)  
vamp\_read\_grids\_raw: [140b](#), [141b](#), [152c](#)  
vamp\_receive\_grid: [17](#), [183c](#), [30](#), [166b](#), [174b](#), [175b](#), [183a](#), [183c](#), [184b](#)  
vamp\_receive\_history: [175b](#), [186b](#), [187](#)  
vamp\_reduce\_channels: [120a](#), [117b](#), [119b](#), [120a](#), [174a](#), [177b](#)  
vamp\_refine\_grid: [14](#), [15](#), [17](#), [91a](#), [84d](#), [91a](#), [91d](#), [100c](#), [117b](#), [166b](#), [173c](#)  
vamp\_refine\_grids: [84d](#), [91b](#)  
vamp\_refine\_weights: [121c](#), [120b](#), [121c](#), [170a](#), [170b](#), [171b](#), [195b](#), [209b](#),  
[211](#), [238b](#)  
vamp\_reshape\_grid: [81b](#), [81b](#), [79a](#), [81a](#), [81b](#), [116d](#), [164b](#), [165a](#), [166a](#)  
vamp\_reshape\_grid\_internal: [80](#), [81a](#), [81b](#), [91a](#)  
vamp\_reshape\_grids: [116d](#), [115c](#), [116c](#), [116d](#)  
vamp\_rest: [75a](#), [75b](#)  
vamp\_rigid\_divisions: [15](#), [17](#), [83d](#), [83c](#), [83d](#), [100c](#), [166b](#)  
vamp\_sample\_grid: [91d](#), [14](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [24b](#), [24c](#), [91d](#), [91d](#),  
[91d](#), [91d](#), [84d](#), [91d](#), [91d](#), [91d](#), [91d](#), [91d](#), [136b](#), [137c](#), [164b](#), [165a](#), [166b](#),  
[177b](#), [179b](#), [194b](#), [208b](#), [210b](#), [237](#), [238a](#), [239](#)  
vamp\_sample\_grid0: [14](#), [15](#), [85a](#), [17](#), [85a](#), [84d](#), [85a](#), [91d](#), [100c](#), [85a](#), [119a](#),  
[85a](#), [136b](#), [136c](#), [166b](#), [85a](#), [173d](#), [179b](#), [179c](#), [85a](#)  
vamp\_sample\_grid\_parallel: [100c](#), [100c](#), [100b](#), [100c](#)  
vamp\_sample\_grids: [117b](#), [117a](#), [117b](#), [136c](#), [170a](#), [170b](#), [117b](#), [172](#), [195b](#),  
[209b](#), [211](#), [238b](#)  
vamp\_send\_grid: [17](#), [183b](#), [30](#), [166b](#), [174b](#), [175a](#), [183a](#), [183b](#), [183b](#), [183b](#),  
[184a](#)  
vamp\_send\_history: [175a](#), [186b](#), [186c](#)  
vamp\_serial\_mpi: [163b](#), [163b](#), [163b](#), [163c](#), [164a](#)  
vamp\_stat: [37a](#), [75a](#), [131](#), [289a](#)  
vamp\_sum\_channels: [122d](#), [122c](#), [122d](#)  
vamp\_terminate\_history: [104d](#), [103d](#), [104a](#), [104d](#), [119a](#), [121d](#), [166b](#),  
[173d](#), [174a](#), [177b](#), [368b](#), [368c](#), [369a](#)  
vamp\_test0: [200b](#), [208b](#), [209b](#), [213](#)  
vamp\_test0\_functions: [201](#), [207b](#), [215c](#)  
vamp\_tests: [192b](#), [196c](#), [192b](#)  
vamp\_tests0: [207a](#), [215c](#), [216a](#)  
vamp\_unmarshal\_grid: [156b](#), [29](#), [154b](#), [156b](#), [183c](#), [184b](#), [185b](#)  
vamp\_unmarshal\_history: [159b](#), [158a](#), [159b](#), [187](#)



vamp\_update\_weights: 116b, 116a, 116b, 121c, 170a, 170b, 171a  
vamp\_warmup\_grid: 136b, 136b, 136a, 136b, 178a, 178b, 179b, 210b  
vamp\_warmup\_grids: 136c, 136a, 136c, 178a, 178b, 179c, 211  
vamp\_write\_grid: 141a, 140a, 141a, 180a, 180b, 180c, 194b, 208b, 237,  
238a  
vamp\_write\_grid\_raw: 140b, 141b  
vamp\_write\_grids: 141a, 140a, 141a, 181d, 181e, 181f, 195b, 209b, 238b  
vamp\_write\_grids\_raw: 140b, 141b  
vamp\_write\_histories: 105b  
vamp\_write\_history: 105b, 105c  
vamp\_write\_one\_history: 105b  
var\_f: 44d, 87d, 87d, 88a, 362d  
volume\_division: 56b, 57a, 83a, 96b, 363d, 363e, 364c  
w: 51a, 95a, 188b, 192a, 195b, 201, 206b, 209b, 211, 228a, 228b  
wgt: 43b, 43c, 44a, 86d, 86d, 86d, 86d, 87a, 87d, 87d, 132d, 133a, 133d,  
135a, 135b, 135c, 361c, 361d  
wgt0: 129c, 130b  
wgt1: 129c, 130b  
wgts: 86d, 87d, 112, 132d, 133a  
write\_division: 60d, 60e, 141c  
write\_division\_name: 60d, 60e, 63b  
write\_division\_raw: 60d, 60e, 148b  
write\_division\_raw\_name: 60d, 60e, 65a  
write\_division\_raw\_unit: 60d, 60e, 63d  
write\_division\_unit: 60d, 60e, 61a, 63b, 65a  
write\_grid\_name: 140a, 141a, 145c, 180a, 180c, 181b  
write\_grid\_raw\_name: 140b, 141b, 151a  
write\_grid\_raw\_unit: 140b, 141b, 148b, 151a, 152a  
write\_grids\_name: 140a, 141a, 147, 181d, 181f, 182c  
write\_grids\_raw\_name: 140b, 141b, 153  
write\_grids\_raw\_unit: 140b, 141b, 152a, 153  
write\_grids\_unit: 140a, 141a, 146a, 147, 181d, 181f, 182a  
write\_grid\_unit: 140a, 141a, 141c, 145c, 146a, 180a, 180c, 180d  
write\_histogram: 210b, 211, 293b, 293c, 296b, 299  
write\_histogram1: 293c, 293d, 296b  
write\_histogram2: 293c, 293d, 299  
write\_histogram1\_unit: 293c, 297a, 297b  
write\_history: 58c, 58d, 59b, 107c  
write\_raw\_state\_name: 262e, 262f, 266d  
write\_raw\_state\_unit: 262e, 262f, 263c, 264a, 266d  
write\_state\_array: 263c, 264a, 264c, 264d, 265a, 265b

write\_state\_name: [262e](#), [262f](#), [266c](#)  
 write\_state\_unit: [262e](#), [262f](#), [263c](#), [266c](#)  
 wx: [233c](#), [233d](#), [238b](#)  
 x: [23a](#), [31a](#), [31b](#), [31c](#), [86d](#), [86d](#), [86d](#), [86d](#), [86d](#), [86d](#), [86d](#), [37b](#), [38b](#), [39a](#), [39b](#), [39c](#),  
[40b](#), [43b](#), [43c](#), [44a](#), [45c](#), [86d](#), [47a](#), [48a](#), [48c](#), [49d](#), [50](#), [51d](#), [52c](#), [54c](#), [54d](#),  
[55a](#), [55d](#), [56a](#), [56c](#), [57c](#), [57f](#), [58b](#), [58d](#), [59d](#), [61a](#), [62](#), [63a](#), [63d](#), [64b](#), [66b](#),  
[66c](#), [67a](#), [68c](#), [69a](#), [86d](#), [86d](#), [87a](#), [87c](#), [87d](#), [89c](#), [110c](#), [111a](#), [111c](#), [112](#),  
[113b](#), [122d](#), [129b](#), [130b](#), [132d](#), [133a](#), [133d](#), [178d](#), [179a](#), [189a](#), [189b](#), [189c](#),  
[190](#), [191a](#), [191b](#), [192a](#), [192d](#), [193a](#), [193b](#), [202a](#), [203a](#), [204](#), [205a](#), [205b](#),  
[205c](#), [206a](#), [206b](#), [207d](#), [210b](#), [211](#), [221a](#), [221b](#), [222b](#), [224a](#), [224c](#), [225b](#),  
[226a](#), [226c](#), [227b](#), [228b](#), [229a](#), [229d](#), [230c](#), [231a](#), [233d](#), [234](#), [246](#), [248f](#), [252b](#),  
[252d](#), [253d](#), [253e](#), [254a](#), [254b](#), [254c](#), [254d](#), [254e](#), [254f](#), [255a](#), [255c](#), [257c](#),  
[257e](#), [257f](#), [257g](#), [257h](#), [258a](#), [258b](#), [258d](#), [262a](#), [264a](#), [264b](#), [265c](#), [268c](#),  
[268d](#), [269a](#), [270a](#), [270b](#), [270c](#), [275b](#), [275c](#), [275d](#), [276a](#), [276b](#), [276c](#), [285c](#),  
[287b](#), [287c](#), [289c](#), [290a](#), [290b](#), [290d](#), [290e](#), [294b](#), [295b](#), [298d](#), [298e](#), [308c](#),  
[328a](#), [328b](#), [329d](#), [329e](#), [330d](#), [331a](#), [335b](#), [336a](#), [336d](#), [337a](#), [337b](#), [338a](#),  
[338b](#), [339a](#), [339b](#), [339c](#), [361c](#), [361d](#), [364a](#)  
 x5: [224c](#), [226a](#), [229d](#), [230c](#), [231a](#), [232a](#), [232b](#)  
 x\_mid: [43b](#), [86d](#), [87d](#), [88a](#), [361c](#)  
 x\_new: [47a](#), [48a](#)

## O.2 Refinements

⟨API documentation [247](#)⟩  
 ⟨Accept distribution among  $n$  workers [103a](#)⟩  
 ⟨Adjust Jacobian [222a](#)⟩  
 ⟨Adjust grid and other state for new num\_calls [82](#)⟩  
 ⟨Adjust  $h\%div$  iff necessary [105a](#)⟩  
 ⟨Allocate or resize the divisions [97a](#)⟩  
 ⟨Alternative to `basic.f90` [24d](#)⟩  
 ⟨Application in Rambo mode [239](#)⟩  
 ⟨Application in massless single channel mode [238a](#)⟩  
 ⟨Application in multi channel mode [238b](#)⟩  
 ⟨Application in single channel mode [237](#)⟩  
 ⟨ $A' = P^T(\phi; p, q) \cdot A \cdot P(\phi; p, q)$  [317c](#)⟩  
 ⟨Bail out if any (`d == NaN`) [46c](#)⟩  
 ⟨Bail out if exception `exc` raised [96d](#)⟩  
 ⟨Body of `create*_array2_pointer` [303b](#)⟩  
 ⟨Body of `create*_array_pointer` [303a](#)⟩  
 ⟨Body of `mpi90_broadcast*_array` [352a](#)⟩

<Body of mpi90\_receive\_\*\_array 347c>  
 <Body of mpi90\_receive\_\*\_pointer 349c>  
 <Body of mpi90\_send\_\*\_array 345b>  
 <Body of multi\_channel 195b>  
 <Body of tao\_random\_\* 272b>  
 <Body of tao\_random\_\*\_array 273d>  
 <Boost and rescale the vectors 331a>  
 <Bootstrap the x and xl buffers 257d>  
 <Bootstrap the x buffer 253d>  
 <Check optional arguments in vamp\_sample\_grid0 89a>  
 <Choose a x and calculate f(x) 133a>  
 <Cleanup in vamp\_test0 215f>  
 <Clenshaw's recurrence formula 285b>  
 <Collect integration and grid optimization data for current cell 88a>  
 <Collect integration and grid optimization data for x from f 87b>  
 <Collect results of vamp\_sample\_grid0 88b>  
 <Combine the rest of gs onto g 98a>  
 <Constants in divisions 64a>  
 <Constants in vamp 149a>  
 <Constants in vamp\_equivalences 71c>  
 <Construct  $\hat{R}(\theta; i, j)$  126a>  
 <Copy results of vamp\_sample\_grid to dummy variables 93c>  
 <Copy results of vamp\_sample\_grids to dummy variables 175f>  
 <Copy the rest of g to the gs 95b>  
 <Copyleft notice 1>  
 <Count up cell, exit if done 85b>  
 <Declaration of 30-bit tao\_random\_numbers 264d>  
 <Declaration of 52-bit tao\_random\_numbers 265b>  
 <Declaration of 30-bit tao\_random\_numbers types 258b>  
 <Declaration of 52-bit tao\_random\_numbers types 258d>  
 <Declaration of coordinates procedures 335a>  
 <Declaration of cross\_section procedures 219d>  
 <Declaration of divisions procedures 38a>  
 <Declaration of divisions procedures (removed from WHIZARD) 59c>  
 <Declaration of divisions types 37b>  
 <Declaration of exceptions procedures 244b>  
 <Declaration of exceptions types 243b>  
 <Declaration of histograms procedures 293b>  
 <Declaration of histograms types 292b>  
 <Declaration of kinematics procedures 322b>  
 <Declaration of kinematics types 324g>

<Declaration of `linalg` procedures 311b>  
 <Declaration of `mpi90` procedures 341b>  
 <Declaration of `mpi90` types 346b>  
 <Declaration of `phase_space` procedures 329b>  
 <Declaration of `phase_space` types 328a>  
 <Declaration of procedures in `vamp_tests0` 208a>  
 <Declaration of procedures in `vamp_tests` 192c>  
 <Declaration of procedures in `vamp_tests0` (broken?) 207c>  
 <Declaration of `specfun` procedures 284b>  
 <Declaration of `stest_functions` procedures 287a>  
 <Declaration of `tao_random_numbers` 251b>  
 <Declaration of `tao_random_numbers` (unused luxury) 280a>  
 <Declaration of `utils` procedures 302b>  
 <Declaration of `vamp` procedures 76b>  
 <Declaration of `vamp` procedures (removed from *WHIZARD*) 113a>  
 <Declaration of `vamp` types 103c>  
 <Declaration of `vamp_equivalences` procedures 71e>  
 <Declaration of `vamp_equivalences` types 71a>  
 <Declaration of `vamp_grid_type` types 76a>  
 <Declaration of `vampi` procedures 164b>  
 <Declaration of `vampi` types 169e>  
 <Declaration of `vamp_stat` procedures 289b>  
 <Decode `channel` into `ch` and `p(:)` 205b>  
 <Determine  $\phi$  for the Jacobi rotation  $P(\phi; p, q)$  with  $A'_{pq} = 0$  317a>  
 <Distribute complete grids among processes 173b>  
 <Distribute each grid among processes 177b>  
 <Estimate `waste` of processor time 177a>  
 <Execute `command` 215b>  
 <Execute `command` in `vamp_test` (never defined)>  
 <Execute tests in `vamp_test0` 214a>  
 <Exit `iterate` if accuracy has been reached 94a>  
 <Exit `iterate` if accuracy has been reached (MPI) 176a>  
 <Fill `state` from `x` 255c>  
 <Fork a pseudo stratified sampling division 52c>  
 <Fork a pure stratified sampling division 51d>  
 <Fork an importance sampling division 51a>  
 <Gather exceptions in `vamp_sample_grid_parallel` 102a>  
 <Generate isotropic null vectors 330e>  
 <Get  $\cos \theta$  and  $\sin \theta$  from `evecs` 125e>  
 <Get `x` in the current cell 86d>  
 <Handle `g%calls_per_cell` for `d == 0` 95a>

<Handle local\_error 342c>  
 <Handle local\_error (no mpi90\_abort) 342b>  
 <Handle optional pancake and cigar 126c>  
 <Idioms 98b>  
 <Implementation of 30-bit tao\_random\_numbers 251a>  
 <Implementation of 52-bit tao\_random\_numbers 256b>  
 <Implementation of coordinates procedures 335b>  
 <Implementation of cross\_section procedures 220a>  
 <Implementation of divisions procedures 38b>  
 <Implementation of divisions procedures (removed from WHIZARD) 44e>  
 <Implementation of exceptions procedures 244c>  
 <Implementation of histograms procedures 293f>  
 <Implementation of kinematics procedures 323a>  
 <Implementation of linalg procedures 312>  
 <Implementation of mpi90 procedures 342a>  
 <Implementation of phase\_space procedures 329d>  
 <Implementation of procedures in vamp\_tests0 208b>  
 <Implementation of procedures in vamp\_tests 193a>  
 <Implementation of procedures in vamp\_tests (broken?) 192d>  
 <Implementation of procedures in vamp\_tests0 (broken?) 207d>  
 <Implementation of specfun procedures 285c>  
 <Implementation of stest\_functions procedures 287b>  
 <Implementation of tao\_random\_numbers 251f>  
 <Implementation of utils procedures 303c>  
 <Implementation of vamp procedures 76d>  
 <Implementation of vamp procedures (removed from WHIZARD) 113b>  
 <Implementation of vamp\_equivalences procedures 71d>  
 <Implementation of vampi procedures 165b>  
 <Implementation of vampi procedures (doesn't work with MPICH yet) 184a>  
 <Implementation of vamp\_stat procedures 289c>  
 <Implementation of vamp\_test0\_functions procedures 202a>  
 <Implementation of vamp\_test\_functions procedures 189a>  
 <Increment  $k$  until  $\sum m_k \geq \Delta$  and keep the surplus in  $\delta$  47c>  
 <Initialize a virginal random number generator 276d>  
 <Initialize clusters 129b>  
 <Initialize num\_rot 318e>  
 <Initialize stratified sampling 42>  
 <Insure that associated (g%map) == .false. 145a>  
 <Insure that associated (g%mu\_x) == .false. 145b>  
 <Insure that size (g%div) == ndim 144>  
 <Insure that ubound (d%x, dim=1) == num\_div 63a>

<Interface declaration for `func` 22>  
 <Interface declaration for `ihp` 31b>  
 <Interface declaration for `jacobian` 31c>  
 <Interface declaration for `phi` 31a>  
 <Interfaces of 30-bit `tao_random_numbers` 277f>  
 <Interfaces of 52-bit `tao_random_numbers` 278d>  
 <Interfaces of divisions procedures 60e>  
 <Interfaces of divisions procedures (removed from WHIZARD) 362b>  
 <Interfaces of exceptions procedures 358d>  
 <Interfaces of histograms procedures 293c>  
 <Interfaces of kinematics procedures 322c>  
 <Interfaces of `mpi90` procedures 344c>  
 <Interfaces of `phase_space` procedures 329c>  
 <Interfaces of `tao_random_numbers` 255d>  
 <Interfaces of `tao_random_numbers` (unused luxury) 279g>  
 <Interfaces of `utils` procedures 302c>  
 <Interfaces of `vamp` procedures 93b>  
 <Interfaces of `vampi` procedures 169b>  
 <Interpolate the new  $x_i$  from  $x_k$  and  $\delta$  48a>  
 <Join closest clusters 129c>  
 <Join importance sampling divisions 51b>  
 <Join pseudo stratified sampling divisions 54a>  
 <Join pure stratified sampling divisions 52a>  
 <Kill `m4(1)` builtins 355b>  
 <Load `a` and refresh state 251c>  
 <Load 52-bit `a` and refresh state 256c>  
 <Local variables in `vamp_sample_grid0` 86a>  
 <MPI communication example 29>  
 <MPI communication example' 30>  
 <Maybe accept unweighted event 133c>  
 <Maybe accept unweighted multi channel event 135b>  
 <Maybe accept unweighted multi channel event (old version) 135c>  
 <Module `vamp_test0_functions` 201>  
 <Module `vamp_test_functions` 188b>  
 <Module `vamp_tests` 192b>  
 <Modules used by `vamp_test0` 215c>  
 <Modules used by `vamp_tests0` 207b>  
 <Parallel implementation of  $S_n = S_0(rS_0)^n$  (HPF) 15>  
 <Parallel implementation of  $S_n = S_0(rS_0)^n$  (MPI) 17>  
 <Parallel usage of  $S_n = S_0(rS_0)^n$  (HPF) 16>  
 <Parameters in `mpi90` (never defined)>

<Parameters in tao\_random\_numbers 250a>  
 <Parameters in tao\_random\_numbers (alternatives) 250b>  
 <Parameters in tao\_random\_test 280e>  
 <Parameters in utils 309c>  
 <Parameters in vampi 165c>  
 <Parameters local to tao\_random\_seed 253a>  
 <Parse the commandline in vamp\_test and set command (never defined)>  
 <Perform more tests of tao\_random\_numbers 281d>  
 <Perform simple tests of tao\_random\_numbers 281a>  
 <Perform the Jacobi rotation resulting in  $A'_{pq} = 0$  316>  
 <Prepare array buffer and done, todo, chunk 274b>  
 <Pull u into the intervall [3, 4] 285a>  
 <Read command line and decode it as command (never defined)>  
 <Receive the result for channel #ch at the root 175b>  
 <Reload buffer or exit 274d>  
 <Reset counters in vamp\_sample\_grid0 85c>  
 <Resize arrays, iff necessary 40b>  
 <Return optional arguments in lu\_decompose 313d>  
 <Sample calls\_per\_cell points in the current cell 86c>  
 <Sample g%g0%grids(ch) 173d>  
 <Sample the grid g%grids(ch) 119a>  
 <Select channel from weights 134b>  
 <Serial implementation of  $S_n = S_0(rS_0)^n$  14>  
 <Set default for domain 343b>  
 <Set defaults for source, tag and domain 346e>  
 <Set i, delta\_x, x, and wgt from xi 43c>  
 <Set i(1), i(2) to the axes of the optimal plane 125b>  
 <Set i(1), i(2) to the axes of the optimal plane (broken!) 125c>  
 <Set iv to the index of the optimal eigenvector 124b>  
 <Set j to minloc(key) 307a>  
 <Set m to (1, 1, ...) or to rebinning weights from d%variance 40a>  
 <Set nu to num or size(v) 274a>  
 <Set rev to reverse or .false. 306e>  
 <Set  $(s_2, t_1, \phi, \cos \theta_3, \phi_3)$  from  $(x_1, \dots, x_5)$  221a>  
 <Set  $(s_2, t_1, \phi, \cos \theta_3, \phi_3)$  from  $(x_1, \dots, x_5)$  (massless case) 221b>  
 <Set subspace to the axes of the optimal plane 127c>  
 <Set up integrand and region in vamp\_test0 215e>  
 <Set up s and t 253f>  
 <Set up seed\_value from seed or DEFAULT\_SEED 253c>  
 <Setup to fork a pseudo stratified sampling division 52b>  
 <Setup to fork a pure stratified sampling division 51c>



<Shift `s` or `t` and exit if `t ≤ 0` 255b>  
 <Ship `g%g0%grids` from the root to the assigned processor 174b>  
 <Ship the result for channel `#ch` back to the root 175a>  
 <Specific procedures for 30-bit `tao_random_number` 278a>  
 <Specific procedures for 52-bit `tao_random_number` 278e>  
 <Specific procedures for `tao_random_copy` 261f>  
 <Specific procedures for `tao_random_create` 259c>  
 <Specific procedures for `tao_random_luxury` 280c>  
 <Specific procedures for `tao_random_seed` 255f>  
 <Step last and reload buffer iff necessary 272d>  
 <Test `a(1) = A.2027082` 281b>  
 <Trace results of `vamp_sample_grid` 103d>  
 <Trace results of `vamp_sample_grids` 121d>  
 <Trivial `ktest.f90` 332>  
 <Types in `cross_section` 224b>  
 <Unconditionally accept weighted event 133b>  
 <Unconditionally accept weighted multi channel event 135a>  
 <(Unused) Interfaces of `phase_space` procedures 329g>  
 <Update last, done and todo and set new chunk 274c>  
 <Update `num_rot` 318f>  
 <Variables in 30-bit `tao_random_numbers` 250c>  
 <Variables in 52-bit `tao_random_numbers` 255g>  
 <Variables in `cross_section` 218b>  
 <Variables in divisions 45d>  
 <Variables in exceptions 243c>  
 <Variables in histograms 293e>  
 <Variables in `mpi90` (never defined)>  
 <Variables in `tao_random_numbers` 266a>  
 <Variables in `utils` 310b>  
 <Variables in `vamp` 78a>  
 <Variables in `vamp_test0` 215a>  
 <Variables local to 52-bit `tao_random_seed` 257b>  
 < $V' = V \cdot P(\phi; p, q)$  318d>  
 <XXX Implementation of `cross_section` procedures 219e>  
 <XXX Variables in `cross_section` 219a>  
 <application.f90 218a>  
 <basic.f90 23a>  
 <52-bit  $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) 257g>  
 <52-bit  $p(z) \rightarrow zp(z)$  (modulo 2 and  $z^K + z^L + 1$ ) 258a>  
 <call `copy_division` (`gs%div(j)`, `g%div(j)`) 96e>  
 <call `fork_division` (`g%div(j)`, `gs%div(j)`, `g%calls_per_cell`, ...) 96c>



<call join\_division (g%div(j), gs%div(j)) 97c>  
 <call sum\_division (g%div(j), gs%div(j)) 97d>  
 < $c_0/2, c_1, c_2, \dots, c_{15}$  for  $\Gamma(x)$  286a>  
 <constants.f90 241b>  
 <coordinates.f90 334>  
 <ctest.f90 131>  
 <divisions.f90 37a>  
 <eps = 1 313a>  
 <eps = - eps 313b>  
 <exceptions.f90 243a>  
 <f = wgt \* func (x, weights, channel), *iff* x inside true\_domain 87a>  
 <f90.m4 355c>  
 <f95.m4 355a>  
 <histograms.f90 292a>  
 <kinematics.f90 322a>  
 <ktest.f90 331d>  
 <la\_sample.f90 319d>  
 <linalg.f90 311a>  
 < $m_a \leftrightarrow m_b, m_1 \leftrightarrow m_2$  for channel #1 222c>  
 <mpi90.f90 341a>  
 <pivots = 0 and eps = 0 313c>  
 < $p_1 \leftrightarrow p_2$  for channel #2 223a>  
 <products.f90 321>  
 < $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) 254a>  
 < $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) (DEK, C) 254e>  
 < $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) (DEK, Fortran) 254c>  
 < $p(z) \rightarrow p(z)^2$  (modulo 2 and  $z^K + z^L + 1$ ) (alternative) 254d>  
 < $p(z) \rightarrow zp(z)$  (modulo 2 and  $z^K + z^L + 1$ ) 255a>  
 <specfun.f90 284a>  
 <stest.f90 286b>  
 <tao52\_random\_numbers.f90 271a>  
 <tao\_random\_numbers.f90 270d>  
 <tao\_test.f90 283>  
 <utils.f90 302a>  
 <vamp0\_\* => vamp\_\* 165a>  
 <vamp.f90 70a>  
 <vampi.f90 163b>  
 <vampi\_test.f90 198a>  
 <vampi\_test0.f90 216a>  
 <vamp\_kinds.f90 241a>  
 <vamp\_stat.f90 289a>

$\langle \text{vamp\_test.f90 } 188\text{a} \rangle$   
 $\langle \text{vamp\_test0.f90 } 200\text{b} \rangle$   
 $\langle \text{vamp\_test.out } 200\text{a} \rangle$   
 $\langle \text{vamp\_test0.out } 217 \rangle$   
 $\langle \text{weights: } \alpha_i \rightarrow w_{\max,i} \alpha_i \text{ } 134\text{a} \rangle$

# INDEX

deficiencies in Fortran90 and F, 76,  
86  
deficiencies of Fortran90 that have  
been fixed in Fortran95, 247  
dependences on external modules,  
86  
  
Fortran problem, 70  
Fortran sucks, 109  
functional programming rules, 109  
  
IEEE hacks, 46  
inconvenient F constraints, 113, 314  
  
more empirical studies helpful, 129  
  
optimizations not implemented yet,  
102  
  
possible error by DEK, 254  
Problems with MPICH, 183, 184  
  
remove from finalized program, 332  
  
system dependencies, 46, 266  
  
theoretical question, 254  
  
unfinished business, 123

## *Acknowledgements*

## BIBLIOGRAPHY

- [1] G. P. Lepage, J. Comp. Phys. **27**, 192 (1978).
- [2] G. P. Lepage, *VEGAS – An Adaptive Multi-dimensional Integration Program*, Cornell preprint, CLNS-80/447, March 1980.
- [3] T. Ohl, *Vegas Revisited: Adaptive Monte Carlo Integration Beyond Factorization*, hep-ph/9806432, Preprint IKDA 98/15, Darmstadt University of Technology, 1998.
- [4] D. E. Knuth, *Literate Programming*, Vol. 27 of *CSLI Lecture Notes* (Center for the Study of Language and Information, Leland Stanford Junior University, Stanford, CA, 1991).
- [5] N. Ramsey, IEEE Software **11**, 97 (1994).
- [6] American National Standards Institute, *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*, New York, 1978.
- [7] International Standards Organization, *ISO/IEC 1539:1991, Information technology — Programming Languages — Fortran*, Geneva, 1991.
- [8] International Standards Organization, *ISO/IEC 1539:1997, Information technology — Programming Languages — Fortran*, Geneva, 1997.
- [9] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.1*, Rice University, Houston, Texas, 1994.
- [10] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Rice University, Houston, Texas, 1997.
- [11] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, Technical Report CS-94230, University of Tennessee, Knoxville, Tennessee, 1994.
- [12] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener, *Fortran 95 Handbook*, The MIT Press, Cambridge, MA, 1997.

- [13] Michael Metcalf and John Reid, *The F Programming Language*, (Oxford University Press, 1996).
- [14] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA, 1994.
- [15] D. E. Knuth, *Seminumerical Algorithms* (third edition), Vol. 2 of *The Art of Computer Programming*, (Addison-Wesley, 1997).
- [16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edition, (Cambridge University Press, 1992)
- [17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in Fortran77: The Art of Scientific Computing*, Volume 1 of *Fortran Numerical Recipes*, 2nd edition, (Cambridge University Press, 1992)
- [18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in Fortran90: The Art of Parallel Scientific Computing*, Volume 2 of *Fortran Numerical Recipes*, (Cambridge University Press, 1992)
- [19] S. Kawabata, Comp. Phys. Comm. **41**, 127 (1986).
- [20] MINAMI-TATEYA Group, *GRACE Manual*, KEK Report 92-19.
- [21] S. Veseli, Comp. Phys. Comm. **108**, 9 (1998).
- [22] R. Kleiss, R. Pittau, *Weight Optimization in Multichannel Monte Carlo*, Comp. Phys. Comm. **83**, 141 (1994).
- [23] George Marsaglia, *The Marsaglia Random Number CD-ROM*, FSU, Dept. of Statistics and SCRI, 1996.
- [24] Y. L. Luke, *Mathematical Functions and their Approximations*, Academic Press, New York, 1975.
- [25] R. Kleiss, W. J. Stirling, S. D. Ellis, *A New Monte Carlo Treatment of Multiparticle Phase Space at High Energies*, Comp. Phys. Comm. **40**, 359 (1986).