BU-HEP-96-10

QCDF90: a set of Fortran 90 modules for a high-level, efficient implementation of QCD simulations

Indranil Dasgupta[1] , **Andrea Ruben Levi**[2] , **Vittorio Lubicz**[3]

and

Claudio Rebbi[4]

Department of Physics, Boston University

590 Commonwealth Avenue, Boston, MA 02215, USA

May 8, 1996

Abstract

We present a complete set of Fortran 90 modules that can be used to write very compact, efficient, and high level QCD programs. The modules define fields (gauge, fermi, generators, complex, and real fields) as abstract data types, together with simpler objects such as SU(3) matrices or color vectors. Overloaded operators are then defined to perform all possible operations between the fields that may be required in a QCD simulation. QCD programs written using these modules need not have cumbersome subroutines and can be very simple and transparent. This is illustrated with two simple example programs.

[1] e-mail: dgupta@budoe.bu.edu
[2] e-mail: leviar@budoe.bu.edu
[3] e-mail: lubicz@weyl.bu.edu
[4] e-mail: rebbi@bu.edu

PROGRAM SUMMARY

Title of program: QCDF90
Computer for which the program is designed: Any computer
Computers under which the program has been tested: Silicon Graphics Indigo and PowerChallengeArray, and IBM R6000 58H *Installations:* Boston University, Center for Computational Science and Department of Physics.
Operating systems under which the program has been tested: IRIX 6.1, and AIX.
Programming language used: Fortran 90
No. of bits in a word: 64
No. of lines in distributed program, including test data, etc: 7806
Keywords: QCD, lattice gauge theory
Nature of physical problem: Non-perturbative computations in QCD
Memory required to execute with typical data: Varies according to the applications. Scales proportionally to the lattice volume $NX * NY * NZ * NT$. On a 16^4 lattice, the example codes quenched.f90 and propagator.f90 use approximately 110 Mbytes and 140 Mbytes respectively.
Typical running time: Varies according to the applications. The example codes quenched.f90 and propagator.f90 take approximately 45 microsec to update an SU(3) link, 8 microsec to calculate a plaquette, and 20 microsec for a CG step per link, using a 16^4 lattice, on an SGI Power-Challenge per node.

LONG WRITE-UP

1 Introduction

The computer simulation of quantum fluctuations (cf. for instance [1], [2], [3]) has been one of the most powerful tools for obtaining information about the non-perturbative properties of quantum field theories in general, and, especially, of Quantum Chromodynamics (QCD) (good accounts of progress in this field of research can be found in the proceedings of the yearly international symposia on lattice gauge theories [4]; see also [5]). These simulations, which deal with matrix and vector fields defined over a four-dimensional space-time lattice, involve huge number of variables and are very demanding in computer resources. Therefore, good payoffs can be obtained in this domain of applications from the development of highly-efficient code. On the other hand, even greater gains can be achieved through the invention of better algorithms, which is made much easier by the availability of high-level, structured programming tools. High-level programming tools are also invaluable for extracting physical results from the data collected in the simulations, which typically requires experimenting with different types of data analysis and involves substantial amounts of code development.

With the twofold goal of facilitating the development of algorithms and applications for lattice QCD, and of maintaining good code performance, we have taken advantage of the possibilities offered by Fortran90 to write a set of modules for a high-level, yet efficient implementation of QCD simulations. Our end product is described in this long write-up, whose main purpose is to provide researchers with all the information needed to use our modules. Since this effectively makes the long write-up a reference document, it is indeed, and necessarily, "long". We have nevertheless striven to be concise, in order to save space and, especially, because we felt that a concise document would make it easier for the user to find the relevant information. Most of the times

the functionality provided by our modules will be obvious. For instance, if `f1` and `f2` are two variables of type fermi_field (see later for the precise definition), `f1+f2` will have as components the sum of the components of the two fields. Similarly, if `g1` and `g2` are variables of type gauge_field, `g1*g2` will have as components the matrix products of the components of `g1` and `g2`. In other instances, however, we had to use a bit of creativity in adapting the symbols of the language to the definitions of some further useful overloaded operators. Thus, if `f1` and `f2` are again variables of type fermi_field, `f1//f2` will be for us a variable of type gauge_field having for components the dyadic formed by the vector components of the two fermi_fields. For all these less obvious definitions, there is no substitute to reading the sections of this article, where all of our overloaded operators are carefully documented.

We expect that most of the users of our modules will be practitioners of lattice gauge theory, and as such already quite knowledgeable about the type of variables that enter QCD simulations. Having in mind, however, that some of the users might be application scientists called on to benchmark code with which they are not too familiar, we decided to include in this write-up a very concise description of the data structures encountered in QCD simulations. This is presented in the next section, which summarizes the kinematics that has been used for lattice QCD since the pioneering work of Wilson [6]. The section that follows discusses the all-important notion of parallel transport in presence of a gauge field and our implementation of parallel transport via a generalization of the C-shift operation, which we call "U-shift". Sections 4, 5 and 6 deal with algorithmic issues related to the ordering of the data, with the description of the data types, and with some further considerations of programming and efficiency. The remaining sections are devoted to a detailed description of our modules and of the functionality which they provide.

2 Geometry and variables

We consider a four-dimensional lattice with extent $\mathtt{NX}, \mathtt{NY}, \mathtt{NZ}$ and $\mathtt{NT}$ in the four directions. We will assume that $\mathtt{NX}, \mathtt{NY}, \mathtt{NZ}, \mathtt{NT}$ are all even. A lattice site will be labeled by four integer valued variables $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ with

$$\mathtt{0} \leq \mathtt{x} < \mathtt{NX}, \quad \mathtt{0} \leq \mathtt{y} < \mathtt{NY}, \quad \mathtt{0} \leq \mathtt{z} < \mathtt{NZ}, \quad \mathtt{0} \leq \mathtt{t} < \mathtt{NT} \ . \tag{1}$$

When convenient, we will denote the collection of these four indices by $\mathbf{x}$.

We will assume periodic boundary conditions.

The physical system is defined in terms of two types of variables (also called the dynamical variables): the gauge fields and the Fermi fields.

The components of a gauge field are 3×3 unitary, unimodular matrices (i.e. elements of the group $SU(3)$, the so called "color" group) defined over the oriented links of the lattice. Later we will see that programming considerations demand a more involved layout of data, but, conceptually, a gauge field can be considered as a multidimensional array of complex variables

$$\mathtt{U}(\mathtt{3}, \mathtt{3}, \mathtt{0} : \mathtt{NX} - \mathtt{1}, \mathtt{0} : \mathtt{NY} - \mathtt{1}, \mathtt{0} : \mathtt{NZ} - \mathtt{1}, \mathtt{0} : \mathtt{NT} - \mathtt{1}, \mathtt{4}) \ , \tag{2}$$

where the first two indices of the generic array element $\mathtt{U}(\mathtt{i}, \mathtt{j}, \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}, \mathtt{m})$ are the indices of the $SU(3)$ matrix, whereas $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ label a lattice site and $\mathtt{m} = \mathtt{1} \ldots \mathtt{4}$ labels one of the four lattice links having origin at the site and oriented in the positive $\mathtt{m}$ direction. When convenient we will use the more compact notation $U^{\mu}_{ij,\mathbf{x}}$ to denote the gauge field elements, or $U^{\mu}_{\mathbf{x}}$ to denote the whole matrix defined over the link (in this compact notation we follow the common practice of using a Greek letter to denote the direction of the link). Another useful notation consists in representing by $\hat{\mu}$ a four-vector having its μ component equal to 1, all other components equal to zero. With this notation, one can say that the gauge variable $U^{\mu}_{\mathbf{x}}$ is defined over the oriented link from $\mathbf{x}$ to $\mathbf{x} + \hat{\mu}$.

The components of a Fermi field are defined over the sites of the lattice. They are 3-dimensional complex vectors with respect to the matrices of the color group and carry an additional spin index $\mathtt{s}$ ranging from 1 to 4. Thus the data layout of a Fermi field can be represented conceptually in terms of an array of complex variables

$$\mathtt{f}(3, 0 : \mathtt{NX} - 1, 0 : \mathtt{NY} - 1, 0 : \mathtt{NZ} - 1, 0 : \mathtt{NT} - 1, 4) \ , \tag{3}$$

where the first index of the generic array element $\mathtt{f}(\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}, \mathtt{s})$ is the color index, $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ label the site and $\mathtt{s}$ is the spin index. When convenient we will use a more compact notation $\psi_{i,\mathbf{x},s}$ for the components of a Fermi field, or $\psi_{\mathbf{x},s}$ for the whole color vector, or even just $\psi_{\mathbf{x}}$.

In the field theoretical definition of the physical system the components of a Fermi field would be anticommuting elements of a Grassmann algebra with integration. The rules of integration over elements of a Grassmann algebra ψ_a, $\bar{\psi}_b$ (a and b stand for any complete set of indices) have, as their most important consequence, the formula

$$\int \prod_a (d\bar{\psi}_a \, d\psi_a) \exp[\sum_{a,b} \bar{\psi}_a A_{a,b} \psi_b] = \mathrm{Det}[A] \ . \tag{4}$$

In computational applications $\mathrm{Det}[A]$ or its derivatives with respect to the dynamical variables are calculated by means of ordinary complex variables ϕ_a, $\bar{\phi}_b$ making use of the identity

$$\mathrm{Det}[A] = \int \prod_a (\frac{d\bar{\phi}_a \, d\phi_a}{\pi}) \exp[\sum_{a,b} \bar{\phi}_a [A^{-1}]_{a,b} \phi_b] \ . \tag{5}$$

Thus effectively one deals with arrays of complex variables as in (3).

3 The notion of U-shift

The gauge field serves to define the transport of dynamical variables between neighboring sites. Gauge theories are characterized by the property of local gauge invariance. In the present context this means that it is always possible to redefine the Fermi variables by an $SU(3)$ transformation

$$\psi_{i,\mathbf{x},s} \to \psi'_{i,\mathbf{x},s} = \sum_j G_{ij,\mathbf{x}} \psi_{j,\mathbf{x},s} \ , \tag{6}$$

where the elements of the gauge transformation $G_{ij,\mathbf{x}}$ are $SU(3)$ matrices defined over the sites. All of the physical quantities must remain invariant under such transformations.

It is clear that, if the Fermi fields transform according to (6) with a $G_{ij,\mathbf{x}}$ that changes from site to site, a straightforward finite difference (as one would use in the approximation of a derivative)

$$(\Delta\psi)_{i,\mathbf{x},s} = \psi_{i,\mathbf{x}+\hat{\mu},s} - \psi_{i,\mathbf{x},s} \tag{7}$$

will produce meaningless results. Rather, one should "transport" the variable $\psi_{\mathbf{x}+\hat{\mu}}$ from the site $\mathbf{x}+\hat{\mu}$ to the site $\mathbf{x}$ by means of the gauge variable $U^\mu_{\mathbf{x}}$ defining a shifted variable

$$\psi^{\text{shifted}}_{i,\mathbf{x},s} = \sum_j U^\mu_{ij,\mathbf{x}} \psi_{j,\mathbf{x}+\hat{\mu},s} \tag{8}$$

and then define a gauge covariant finite difference

$$(D\psi)_{i,\mathbf{x},s} = \psi^{\text{shifted}}_{i,\mathbf{x},s} - \psi_{i,\mathbf{x},s} \ . \tag{9}$$

Under a gauge transformation the gauge field itself changes according to

$$U^\mu_{ij,\mathbf{x}} \to U'^\mu_{ij,\mathbf{x}} = \sum_{kl} G_{ik,\mathbf{x}} U^\mu_{kl,\mathbf{x}} [G^{-1}]_{lj,\mathbf{x}+\hat{\mu}} \ . \tag{10}$$

From Eqs. (9,6,10) one can verify that under a gauge transformation the gauge covariant finite difference changes like ψ itself:

$$(D\psi)_{i,\mathbf{x},s} \to (D\psi')_{i,\mathbf{x},s} = \sum_j G_{ij,\mathbf{x}} (D\psi)_{j,\mathbf{x},s} \ . \tag{11}$$

Thus the gauge covariant finite difference is a meaningful construct and quantities such as its magnitude or the scalar product $\sum_i \bar{\psi}_{i,\mathbf{x},s} (D\psi)_{i,\mathbf{x},s'}$ are gauge invariant and thus physically well defined.

It is clear from the above that a circular shift (C-shift) of an array such as $\mathtt{f}(3, \mathtt{0:NX-1}, \mathtt{0:NY-1}, \mathtt{0:NZ-1}, \mathtt{0:NT-1}, 4)$ will generally be complemented by multiplication by an element of the gauge field. We will therefore define a U-shift operation in the following manner.

A U-shift with positive direction parameter $\mu = 1 \ldots 4$ of the Fermi field $\psi_{i,\mathbf{x},s}$ produces the array $\psi^{\text{shifted}}_{i,\mathbf{x},s}$ as given by Eq. (8).

A U-shift with negative direction parameter $\mu' = -\mu = -1 \ldots -4$ of the Fermi field $\psi_{i,\mathbf{x},s}$ produces the array

$$\psi^{\text{shifted}}_{i,\mathbf{x},s} = \sum_j U^{\dagger\mu}_{ij,\mathbf{x}-\hat{\mu}} \psi_{i,\mathbf{x}-\hat{\mu},s} \ , \tag{12}$$

this latter equation being motivated by the fact that the transport factor over a link crossed in the negative direction is the Hermitian adjoint (or equivalently the inverse, with a unitary group $U^\dagger = U^{-1}$) of the transport factor for the positively oriented link.

We define a U-shift for the gauge field as well. Since the gauge field elements have two color indices which should be associated with the beginning and end of the link (cf. the gauge transformation properties of the gauge field variables Eq. (10)) the U-shift of a gauge field will involve two matrix multiplications. Moreover, it will be convenient to define its action on a generic gauge field, denoted below by V, not necessarily identical to U. The idea is that in general there will be several variables with the properties of a gauge field (see the type definitions below) but there will always be one well defined

"master gauge field", denoted by U, which will serve to define the transport of all gauge dependent variables. With this in mind the action of a U-shift on a gauge field is defined as follows.

A U-shift with positive direction parameter $\mu = 1 \ldots 4$ of the gauge field $V^{\nu}_{ij,\mathbf{x}}$ produces the array

$$V^{\mathrm{shifted},\nu}_{i,\mathbf{x}} = \sum_{kl} U^{\mu}_{ik,\mathbf{x}} V^{\nu}_{kl,\mathbf{x}+\hat{\mu}} U^{\dagger\mu}_{lj,\mathbf{x}+\hat{\nu}} \ . \tag{13}$$

A U-shift with negative direction parameter $\mu' = -\mu = -1 \ldots -4$ of the gauge field $V^{\nu}_{ij,\mathbf{x}}$ produces the array

$$V^{\mathrm{shifted},\nu}_{i,\mathbf{x}} = \sum_{kl} U^{\dagger\mu}_{ik,\mathbf{x}-\hat{\mu}} V^{\nu}_{kl,\mathbf{x}-\hat{\mu}} U^{\mu}_{lj,\mathbf{x}-\hat{\mu}+\hat{\nu}} \ . \tag{14}$$

When acting on field variables which carry no color index (we will define such field variables below) the U-shift reduces to an ordinary C-shift.

4 Even and odd components of field variables

All lattice sites can be subdivided into "even" and "odd" sites according to whether the sum of the integer valued coordinates $\mathtt{x} + \mathtt{y} + \mathtt{z} + \mathtt{t}$ is even or odd (checkerboard subdivision). Correspondingly all field variables can be divided into even and odd variables (for a gauge field variable we base the subdivision on the origin of the link over which the variable is defined, i.e. the $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ indices of the array (2)). With periodic boundary conditions and with an even lattice size in all directions, a C-shift or a U-shift of an even field variable produces an odd field variable and vice versa. There are many algorithms which demand, especially in the context of a parallel implementation, that even and odd variables be treated separately. For example, in a Monte Carlo simulation algorithm all variables at even sites can be upgraded simultaneously while those at odd sites are kept fixed and vice versa. We will accommodate these algorithmic demands by defining all of our field variables

as arrays of even or odd field variables. We will do so by taking advantage of the type definition as follows. All field variables will be defined through a type. The first component of the type will be an integer variable `parity` which will take values `0` and `1` for variables defined over even and odd sites respectively. It will also be convenient to use the value -1 to characterize a field with parity undefined.

For the gauge variables it will be convenient to include in the type a single μ component (of definite parity, of course). Thus, in addition to the variable `parity`, the type will contain an integer variable `dir`, taking values $1 \ldots 4$, to denote the direction of the link (i.e. the value of the index μ). It will be convenient to let `dir` also take the value `0`, to characterize 3×3 complex matrices which are defined over the sites rather than over the links, such as the $SU(3)$ matrices of the gauge transformation in Eq. (6).

Finally the type will then contain an array, denoted by `fc` (for field component) which will contain all the field variables defined over the sites of a given parity.

Insofar as the indexing of the array is concerned, this is to a large extent arbitrary, provided that the mapping between the array indices and the actual Cartesian coordinates of the site is well defined. For instance, one could collapse two neighboring "time" slices into a single one and use indices $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ where `t` ranges now from `0` to $\mathtt{NT2} - 1 = \mathtt{NT}/2 - 1$. On the other hand, with many architectures efficiency considerations recommend that the indices $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}$ be fused into a single index, spanning the range `0` to $\mathtt{NXYZT2} - 1 = \mathtt{NX} * \mathtt{NY} * \mathtt{NZ} * \mathtt{NT}/2 - 1$. This will typically be the case when, because of a vectorized or superscalar architecture, the instructions are pipelined and longer arrays give rise to better performance. In principle a good optimizing compiler should recognize when the individual loops over the $\mathtt{x}, \mathtt{y}, \mathtt{z}$ and `t` indices can be fused into a single one and take advantage of this possibility. However, some compilers may be able to fuse only a limited number of nested loops or, alternatively, this type of optimization may be

hindered by the the presence of further indices or by a large number of instructions within the loops. Since the use of types and overloaded operators makes the actual indexing of the arrays transparent to the user, we decided to use a single index to label all of the sites of a definite parity. This index is constructed by going through the sites of definite parity in a lexicographic order, increasing the x coordinate first, then y, z and t, but, as stated above, the ordering of the sites is largely immaterial. For all those operations which are performed locally over the sites, the detailed mapping between the index and the geometry of the lattice is clearly irrelevant. It is, of course, of consequence for the implementation of the shift operations and for accessing the component of a field at a definite Cartesian site. For such purposes we provide the specifics of the mapping through some global variables and an initialization subroutine. We define the following global variables:

```
INTEGER, DIMENSION(0:NX-1,0:NY-1,0:NZ-1,0:NT-1) ::  xyzt_index
INTEGER, DIMENSION(0:NX-1,0:NY-1,0:NZ-1,0:NT-1) ::  xyzt_parity
INTEGER, DIMENSION(0:NXYZT2-1,0:1,4) ::  xyzt_cartesian
INTEGER, DIMENSION(0:NXYZT2-1,0:1,8) ::  xyzt_neighbor
LOGICAL shift_initialized
```

where the parameter `NXYZT2` $=$ `NX`$*$`NY`$*$`NZ`$*$`NT`$/2$ equals one half of the total number of sites in the lattice.

The arrays defined above are initialized by executing the subroutine `shift_initialization`. The variable `shift_initialized` is initialized to `.FALSE.` All of the function calls which implement the overloaded shift operators check the value of `shift_initialized`. If this is `.FALSE.`, the subroutine `shift_initialization` is called and the arrays are properly initialized. Before returning, `shift_initialization` sets `shift_initialized` to `.TRUE.` From this moment on the arrays can be used to establish the mapping between the Cartesian coordinates and the indices within the sublattices of definite parity. The programmer wishing to use these arrays before any shift operation is performed can, of course, initialize them directly via a call to `shift_initialization`.

The array component `xyzt_index(x,y,z,t)` gives the index of the field component defined over the site with Cartesian coordinates `x,y,z,t`. `xyzt_parity(x,y,z,t)` gives the parity of the site (`xyzt_parity(x,y,z,t)` = `x+y+z+t MOD 2`). `xyzt_cartesian(i,p,m)` gives the Cartesian coordinate (`x,y,z,t` for `m=1,2,3,4` respectively) of the site with index `i` and parity `p`. Finally `xyzt_neighbor(i,p,m)` gives the index of the nearest neighbor site in direction `m` of a site with index `i` and parity `p`. The convention is that the values `m=1,2,3,4` correspond to the nearest neighbor in the forward x, y, z, t directions, whereas `m=5,6,7,8` correspond to the nearest neighbor in the backward x, y, z, t directions, respectively.

5 Types

We define the following F90 types.

5.1 Type gauge_field

```
TYPE gauge_field
  INTEGER parity
  INTEGER dir
  COMPLEX(REAL8),DIMENSION(3,3,0:NXYZT2-1)::fc
END TYPE
```

As discussed in the previous section, a variable of type gauge_field contains the components of a gauge field defined over all the links of direction `dir` emerging from the lattice sites of a given `parity`. The field component `fc(i,j,xyzt)` is an array of double precision complex variables (the kind `REAL8` is defined in the module "precisions", see below), where `i,j` are the indices of the $SU(3)$ matrix and `xyzt` labels the site within the subset of sites of a definite parity.

5.2 Type full_gauge_field

```
TYPE full_gauge_field
  TYPE(gauge_field), DIMENSION(0:1,4) ::  uc
END TYPE
```

A variable of type full_gauge_field is meant to store an entire gauge field configuration, i.e. 8 variables of type gauge_field corresponding to the two parity components and the 4 direction components of a full gauge field. Although the `parity` and `dir` components of the individual `uc(i,j)` components can be given any value, good programming practice recommends that one sets `uc(i,j)%parity = i`, `uc(i,j)%dir = j`.

5.3 Type fermi_field

```
TYPE fermi_field
  INTEGER parity
  COMPLEX(REAL8),DIMENSION(3,0:NXYZT2-1,4)::fc
END TYPE
```

A variable of type fermi_field contains the components of a Fermi field defined over the lattice sites of a given `parity`. The field component `fc(i,xyzt,s)` is an array of double precision complex variables, where `i` is the color index, `xyzt` labels the site and `s` is the spin index of the field.

5.4 Type complex_field

```
TYPE complex_field
  INTEGER parity
  COMPLEX(REAL8),DIMENSION(0:NXYZT2-1)::fc
END TYPE
```

The type complex_field is introduced to store an array of complex numbers `fc(xyzt)` defined over the lattice sites of a given `parity`. Although one could also store such variables in an array of complex numbers, defining a type has

the advantage that one can record the parity of the field and that it becomes possible to define overloaded operators (intrinsic operations on intrinsic types cannot be overloaded). A similar remark applies to the type real_field defined below.

5.5 Type real_field

```
TYPE real_field
  INTEGER parity
  REAL(REAL8),DIMENSION(0:NXYZT2-1)::fc
END TYPE
```

The type real_field is introduced to store an array of real numbers `fc`(`xyzt`) defined over the lattice sites of a given `parity`.

5.6 Type generator_field

```
TYPE generator_field
  INTEGER parity
  INTEGER dir
  REAL(REAL8),DIMENSION(8,0:NXYZT2-1)::fc
END TYPE
```

Although for computational purposes it is useful to store the components of an $SU(3)$ gauge field as 3×3 complex matrices, a general $SU(3)$ matrix is a function of only 8 real independent parameters. In particular, given an 8-dimensional real vector with components v_k one can associate to it the $SU(3)$ matrix

$$U_{ij} = \left[\exp\left(\imath \sum_{k=1}^{8} v_k \lambda^k\right) \right]_{ij} , \tag{15}$$

where the matrices λ^k form a basis in the space of Hermitian traceless 3×3 matrices and satisfy the equations $\mathrm{Tr}(\lambda^k \lambda^{k'}) = 0$ for $k \neq k'$, $\mathrm{Tr}(\lambda^k)^2 = 2$.

The term group generator is commonly used to refer to a traceless Hermitian matrix, such as

$$H_{ij} = \sum_{k=1}^{8} v_k \lambda_{ij}^k \tag{16}$$

in Eq. (15). For some algorithms it is convenient to deal directly with the components v_k of a generator, rather with the exponentiated matrix U or the Hermitian matrix H. For this reason we provide the type generator_field, aimed at storing generator components defined over the sites of a given `parity`. Since generators are frequently associated to gauge field variables, we give the type generator_field a `dir` component as well.

5.7 Type matrix

```
TYPE matrix
  COMPLEX(REAL8),DIMENSION(3,3)::mc
END TYPE
```

The type matrix is defined for programming convenience, in order to allow for the overloading of operators and assignments. For instance, it makes it possible to define an operation `g*m`, where the variables `g` and `m` are of type gauge_field and matrix respectively, which implements the matrix product of the components of a gauge field times a constant matrix.

6 Programming and efficiency considerations

6.1 One layer versus two layer data structure

Conceptually our variables would be most naturally defined in terms of a two layer data structure. At the bottom layer we would find objects such as a single $SU(3)$ matrix or a single color vector, i.e. three dimensional complex vector. Overloaded operations such as matrix multiplication or multiplication of a matrix times a color vector would also be defined. At the top layer we

would then use these objects to define extended fields, such as the gauge field, consisting of an array of objects of type matrix. Operators among the objects of the top layer would be built from the elemental operators already defined at the bottom layer. However appealing, this organization of the data would almost certainly imply a huge penalty in efficiency. It is indeed reasonable to expect that the compiler will implement overloaded operations in terms of function calls. In a two layer structure, then, an operation such as the addition of two Fermi fields would be implemented via repeated calls, site by site, to the function which adds the color vector components of the two fields. It is clear that this use of function calls at very low granularity would imply a heavy computational burden. The only way to regain efficiency would be to inline the function calls implementing the elemental operations. While in principle this is possible, it is not reasonable to expect that compilers would generally allow inlining of function calls that implement operations among derived data types over which they have little direct control. For this reason we decided to forfeit the possibility of defining a two layer data structure, however conceptually pleasing this may be, and organized all of our data into a single layer of user defined types. Thus the types which we introduce to define extended fields are, essentially, F90 arrays complemented with one or two variables (`parity`, `dir`) specifying their attributes. As a consequence the computational cost for the use of overloaded operators between our data structures should not be any bigger than the cost of a call to a function or subroutine that manipulates large arrays. On the other hand, the advantages we gain in code structure and ease of programming are truly remarkable.

6.2 Overloaded assignments

The use of overloaded operators may imply the creation of more temporaries and, consequently, more motion of data than a straightforward implementation of operations among arrays. Consider for example the following operation among variables of type fermi_field:

$$\texttt{f1} = \texttt{f1} + \texttt{f2} + \texttt{f3} \ . \tag{17}$$

(We will formally define the addition of Fermi fields later, but it will perform the obvious operation of adding the `fc` components of the fields.)

With ordinary arrays the compiler might put the result of `f1` + `f2` in a temporary `t1` and then add `t1` and `f3` placing the result in `f1`. Thus there would be two write-to-memory operations per component of the arrays. (A good optimizing compiler could even use registers, dispensing with the creation of the temporary and of one of the copies to memory.) However, if the overloaded addition of Fermi fields is implemented via function calls, what we expect to happen is that the function implementing `f1` + `f2` places the result into a temporary `t1` returning the address of the corresponding data structure to the calling program. The compiler at this point will probably copy `t1` into a temporary `t2`, since it would not be safe to pass the addresses of `t1` and `f3` to the add function which will likely put the result into `t1` again. Finally, the result will be copied into `f1`. If implemented in this manner, the entire operation involves four write-to-memory operations: to `t1`, to `t2`, to `t1` again and to `f1`. (Of course, all of the above is implementation dependent. As far as we know, F90 does not specify how the variables should be passed in function calls. An operating system could let the calling program pass to a function the address where it expects the result, making the call `a` = `function`(`b`, `c`) effectively identical to `CALL subroutine`(`a`, `b`, `c`). In this case the composite operation (17) could be implemented with two copies to memory only.)

The procedure could be drastically simplified through the use of an overloaded assignment + =. Instruction (17) could be written

$$\texttt{f1} \mathrel{+}= \texttt{f2} + \texttt{f3} \ , \tag{18}$$

which the compiler would implement by issuing first a call to a function that adds `f2` and `f3` returning the result in `t1`. The addresses of `f1` and `t1` would then be passed to a subroutine, e.g. `plus_eq`(`a`, `b`) that implements

the operation `f1` = `f1` + `t1` among the components of the data types. The required number of copies to memory would be only two.

In order to allow for these possible gains in efficiency, we have defined a large set of overloaded assignments, which will be detailed in the description of the module "assign" given below. Since F90 permits only the use of the = symbol for the assignment, we have implemented its overloading by defining two global variables: a character variable `assign_type` and an integer variable `assign_spec` (for assign specification, introduced to accommodate assignments of a more elaborate nature). The default values of these variables are $'='$ and `0`. They are initialized with these values and reset to their default values at the end of all overloaded assignments. We follow this procedure to avoid the occurrence of accidental erroneous assignments. When `assign_type` equals $'='$ the result of the assignment between variables of identical type is the expected copy of the data structure at the r.h.s. into the variable at the l.h.s.. (We also define overloaded $'='$ assignments between variables of different type; the results of such assignments are explained in the description of the module "assign".) Overloaded assignments such as `a` + = `b` are obtained by setting `assign_type` (and possibly `assign_spec`) to the appropriate value immediately before the assignment. We recommend the following pattern for the instructions:

$$\texttt{assign\_type} ='+'; \quad \texttt{a} = \texttt{b} \tag{19}$$

or (this implements a U-shift from direction `n`)

$$\texttt{assign\_type} ='\texttt{u}'; \quad \texttt{assign\_spec} = \texttt{n}; \quad \texttt{a} = \texttt{b} \tag{20}$$

The overloaded assignments are implemented via case constructs, which make reference to the values of the global variables `assign_type`, `assign_spec`. A simplified version of the code for an assignment would be as follows:

```
SUBROUTINE typea_eq_typeb(a,b)
```

```
  TYPE(typea), INTENT(INOUT) ::  a
  TYPE(typeb), INTENT(IN) ::  b
  SELECT CASE(assign_type)
  CASE('=')
    implements a=b
  CASE('+')
    implements a=a+b
  CASE DEFAULT
    returns an error message and stops execution if the value
    of assign_type does not correspond to any defined assignment
  END SELECT
  assign_type='=';   assign_spec=0
END SUBROUTINE typea_eq_typeb
```

We wish to emphasize that the structure of data and operations which we have introduced may still cause loss of efficiency with some compilers, even with an optimizing one. It might happen that code performing the same calculations as a code written in terms of our data structures, but formulated without use of any derived data types, is converted, upon compilation, into a more efficient executable. However, we designed our data structure and defined our operators and assignments in a way which should present no barrier to a highly efficient, parallelizing compilation. It will be an interesting experiment to verify how different compilers respond to it.

7 Modules

7.1 Module precisions

This module defines two kind parameters, `REAL8` and `LONG`. These parameters store the kind of an 8-byte floating point variable and of an 8-byte integer variable. They are used to render the kind definitions machine independent. `INTEGER(LONG)` variables are used only for the parallel generation of pseudorandom numbers in a system independent way (cf. the module "random_numbers"). If 8-byte integers are not supported by the architecture, the

module random_numbers should be modified to run with shorter integers or to use system supplied parallel pseudorandom numbers, and the definition of LONG should be changed accordingly.

7.2 Module global_module

This module defines the integer constants NX, NY, NZ and NT which specify the size of the lattice. NX, NY, NZ, NT must all be even. It defines the reduced temporal extent NT2 = NT/2, and the products NXYZT = NX * NY * NZ * NT, NXYZT2 = NX * NY * NZ * NT2. It also defines for convenience the constants NCGV = 9 * NXYZT2, NCFV = 12 * NXYZT2, NRGV = 2 * NCGV, NRFV = 2 * NCFV, NRGEV = 8 * NXYZT2, which are equal to the number of complex or, respectively, real variables in the fc components of the types gauge_field, fermi_field and generator_field.

All of the types introduced in Sect. 5 are declared in this module.

Finally the module declares a few global variables, namely, the master gauge field:

```
TYPE(full_gauge_field)  u
```

the assignment variables (cf. Sect. 6.2):

```
CHARACTER  assign_type
INTEGER  assign_spec
```

the arrays xyzt_index, xyzt_parity, xyzt_cartesian, xyzt_neighbor and the logical variable shift_initialized, already mentioned in Sect. 4,
the context logical array, used in conditional operations (cf. the module "conditionals"):

```
LOGICAL, DIMENSION(0 : NXYZT2 - 1) :: context
```

and the variables used for the generation of pseudorandom numbers (see the module "random_numbers"):

```
INTEGER  seed_a, seed_b
INTEGER, DIMENSION(0 : NXYZT2 - 1) :: seeds
```

The module contains the subroutine shift_initialization (see Sect. 4).

7.3 Module constants

This module defines some useful parameters, making them available to all program units which use it. Namely, the following real constants are defined: `PI` (π), `PI2` ($\pi/2$), `TWOPI` (2π), `SQRT2` ($\sqrt{2}$), `SQRT22` ($\sqrt{2}/2$), `SQRT3` ($\sqrt{3}$), `SQRT33` ($\sqrt{3}/3$), `TWOSQRT33` ($2\sqrt{3}/3$), the complex constant `IU` ($\imath$), and the arrays:

`COMPLEX(REAL8), DIMENSION(3,3) :: ZERO_m, UNIT, IU_m`

`COMPLEX(REAL8), DIMENSION(3) :: ZERO_v`

`REAL(REAL8), DIMENSION(8) :: ZERO_ge`

`COMPLEX(REAL8), DIMENSION(3,3,8) :: LAMBDA`

`COMPLEX(REAL8), DIMENSION(4,4,5) :: GAMMA`

`UNIT` and `IU_m` are set equal to the unit matrix, and to $\imath$ times the unit matrix, respectively. `ZERO_m`, `ZERO_v`, `ZERO_ge` have all components equal to zero. The array `LAMBDA` stores the components of the λ matrices:
`LAMBDA(i,j,k)` $= \lambda^k_{i,j}$,
and the array `GAMMA` stores the components of Dirac's γ matrices,
`GAMMA(s1,s2,m)` $= \gamma^m_{s1,s2}$, $m = 1 \ldots 5$, in our chosen representation.
(We follow the convention $\gamma^5 = \gamma^1\gamma^2\gamma^3\gamma^4$.)

Notice that we do not make any distinction between upper and lower indices for the λ and γ matrices: $\lambda^k = \lambda_k$, $\gamma^m = \gamma_m$ and the use of upper or lower indices is only dictated by notational convenience.

7.4 Module field_algebra

This module defines several overloaded operators that perform arithmetic operations between fields and other variables. We describe here all the operations which are defined. For conciseness we introduce notational conventions. We use the symbols `g`, `u`, `f`, `c`, `r`, `ge` and `m` to denote variables of type gauge_field, full_gauge_field, fermi_field, complex_field, real_field, generator_field and matrix, respectively, and the symbols `complex` and `real` to

denote a complex or real variable of kind REAL8 (cf. Sect. 7.1). When necessary, we will use subscripts, e.g. $\mathtt{f}_1, \mathtt{f}_2$, to distinguish between two variables of the same type.

All operators obey the following general rules:

i) When the result of the operation is a field, if the two operands have a parity component, the parity of the result is the parity of the operands if they have the same parity, otherwise it is undefined (i.e. $= -1$). If a single operand has a parity component, then the parity of the result takes the same value. A similar rule applies to the direction component of the variables of type gauge_field and generator_field: if both operands have the same dir or a single operand carries a dir component, then the dir component of the result is set to this value. Otherwise it is set to 0.

ii) When the operator acts between fields, the operation is performed site by site and the result is again a variable of field type. When the operator acts between a variable of type field and a global variable (i.e. m, complex and real) the site variable is combined with the global variable. For example, the operations $\mathtt{c} = \mathtt{c}_1 + \mathtt{c}_2$ and $\mathtt{c} = \mathtt{c}_1 +$ complex would be implemented as

```
DO xyzt=0,NXYZT2-1
  c%fc(xyzt)=c1%fc(xyzt)+c2%fc(xyzt)
END DO
```

and

```
DO xyzt=0,NXYZT2-1
  c%fc(xyzt)=c1%fc(xyzt)+complex
END DO
```

respectively.

The following operations are defined and have the obvious meaning, implicit in the symbol:

$\mathtt{g}_1 + \mathtt{g}_2$, $\mathtt{g}_1 - \mathtt{g}_2$, $\mathtt{g}_1 * \mathtt{g}_2$, $\mathtt{g} * \mathtt{f}$, $\mathtt{f} * \mathtt{g}$, $\mathtt{g} * \mathtt{c}$, $\mathtt{c} * \mathtt{g}$, $\mathtt{g}/\mathtt{c}$,
$\mathtt{g} * \mathtt{r}$, $\mathtt{r} * \mathtt{g}$, $\mathtt{g}/\mathtt{r}$, $\mathtt{g} + \mathtt{m}$, $\mathtt{m} + \mathtt{g}$, $\mathtt{g} - \mathtt{m}$, $\mathtt{m} - \mathtt{g}$, $\mathtt{g} * \mathtt{m}$, $\mathtt{m} * \mathtt{g}$,
$\mathtt{g} *$ complex, complex $* \mathtt{g}$, g/complex, $\mathtt{g} *$ real, real $* \mathtt{g}$, g/real;

$\mathtt{f}_1+\mathtt{f}_2$, $\mathtt{f}_1-\mathtt{f}_2$, $\mathtt{f}*\mathtt{c}$, $\mathtt{c}*\mathtt{f}$, $\mathtt{f}/\mathtt{c}$, $\mathtt{f}*\mathtt{r}$, $\mathtt{r}*\mathtt{f}$, $\mathtt{f}/\mathtt{r}$, $\mathtt{f}*\mathtt{m}$, $\mathtt{m}*\mathtt{f}$, $\mathtt{f}*\mathtt{complex}$, $\mathtt{complex}*\mathtt{f}$, $\mathtt{f}/\mathtt{complex}$, $\mathtt{f}*\mathtt{real}$, $\mathtt{real}*\mathtt{f}$, $\mathtt{f}/\mathtt{real}$;

$\mathtt{c}_1+\mathtt{c}_2$, $\mathtt{c}_1-\mathtt{c}_2$, $\mathtt{c}_1*\mathtt{c}_2$, $\mathtt{c}_1/\mathtt{c}_2$, $\mathtt{c}+\mathtt{r}$, $\mathtt{r}+\mathtt{c}$, $\mathtt{c}-\mathtt{r}$, $\mathtt{r}-\mathtt{c}$, $\mathtt{c}*\mathtt{r}$, $\mathtt{r}*\mathtt{c}$, $\mathtt{c}/\mathtt{r}$, $\mathtt{r}/\mathtt{c}$, $\mathtt{c}+\mathtt{complex}$, $\mathtt{complex}+\mathtt{c}$, $\mathtt{c}-\mathtt{complex}$, $\mathtt{complex}-\mathtt{c}$, $\mathtt{c}*\mathtt{complex}$, $\mathtt{complex}*\mathtt{c}$, $\mathtt{c}/\mathtt{complex}$, $\mathtt{complex}/\mathtt{c}$, $\mathtt{c}+\mathtt{real}$, $\mathtt{real}+\mathtt{c}$, $\mathtt{c}-\mathtt{real}$, $\mathtt{real}-\mathtt{c}$, $\mathtt{c}*\mathtt{real}$, $\mathtt{real}*\mathtt{c}$, $\mathtt{c}/\mathtt{real}$, $\mathtt{real}/\mathtt{c}$;

$\mathtt{r}_1+\mathtt{r}_2$, $\mathtt{r}_1-\mathtt{r}_2$, $\mathtt{r}_1*\mathtt{r}_2$, $\mathtt{r}_1/\mathtt{r}_2$, $\mathtt{r}+\mathtt{real}$, $\mathtt{real}+\mathtt{r}$, $\mathtt{r}-\mathtt{real}$, $\mathtt{real}-\mathtt{r}$, $\mathtt{r}*\mathtt{real}$, $\mathtt{real}*\mathtt{r}$, $\mathtt{r}/\mathtt{real}$, $\mathtt{real}/\mathtt{r}$;

$\mathtt{ge}_1+\mathtt{ge}_2$, $\mathtt{ge}_1-\mathtt{ge}_2$, $\mathtt{ge}*\mathtt{r}$, $\mathtt{r}*\mathtt{ge}$, $\mathtt{ge}/\mathtt{r}$, $\mathtt{ge}*\mathtt{real}$, $\mathtt{real}*\mathtt{ge}$, $\mathtt{ge}/\mathtt{real}$;

$\mathtt{m}_1+\mathtt{m}_2$, $\mathtt{m}_1-\mathtt{m}_2$, $\mathtt{m}_1*\mathtt{m}_2$, $\mathtt{m}*\mathtt{complex}$, $\mathtt{complex}*\mathtt{m}$, $\mathtt{m}/\mathtt{complex}$, $\mathtt{m}*\mathtt{real}$, $\mathtt{real}*\mathtt{m}$, $\mathtt{m}/\mathtt{real}$;

We do not provide any clarification about the operations listed above (it would be truly superfluous) but for the observation that the symbol $*$ implies matrix multiplication when acting between operands of type gauge_field or matrix, and matrix by vector or vector by matrix when one of the operand is a fermi_field and the other a gauge_field or a matrix. Notice that there is no implicit complex conjugation of the vector at the r.h.s. of a vector by matrix multiplication, i.e. $\mathtt{f}=\mathtt{f}_1*\mathtt{m}$ translates into

```
DO s=1,4
DO xyzt=0,NXYZT2-1
DO i=1,3
  f%fc(i,xyzt,s)=f1%fc(1,xyzt,s)*m%mc(1,i)  &
      +f1%fc(2,xyzt,s)*m%mc(2,i) +f1%fc(3,xyzt,s)*m%mc(3,i)
END DO
END DO
END DO
```

The following additional operations have a special meaning:

`g1/g2` :
the gauge field `g1` is multiplied, site by site, by the Hermitian adjoint of the gauge field `g2` (the notation is motivated by the fact that, with unitary matrices, the Hermitian adjoint of a matrix is also its inverse; however, there is no restriction that the variables stored in a gauge field must represent unitary matrices).
`m/g2` and `g1/m` : same as above, but with `m` a matrix rather than a gauge field.
`g1//g2` : the Hermitian adjoint of the gauge field `g1` is multiplied, site by site, by the gauge field `g2`.
`g1//m` and `m//g2` : same as above, but with `m` a matrix rather than a gauge field.
`f/g` and `g//f` : the Fermi field `f` is right or left multiplied, site by site, by the Hermitian adjoint of the gauge field `g`.
`f/m` and `m//f` : same as above, but with `m` a matrix rather than a gauge field.
`f1*f2` :
this operation returns a complex field having as site components the scalar product, taken over the color and the spin indices, of the complex conjugate of `f1` and `f2`. Explicitly, `c = f1*f2` would be implemented as

```
DO xyzt=0,NXYZT2-1
  c%fc(xyzt)=0
DO s=1,4
DO i=1,3
  c%fc(xyzt)=c%fc(xyzt)    &
      +CONJG(f1%fc(i,xyzt,s))*f2%fc(i,xyzt,s)
END DO
END DO
END DO
```

`f1//f2` :
this operation returns a variable of type gauge_field having as site components the dyadic (over the color indices) of `f1` and the complex conjugate of `f2`. The

spin indices are summed over. Explicitly, $\mathtt{g} = \mathtt{f}_1 // \mathtt{f}_2$ would be implemented as

```
DO xyzt=0,NXYZT2-1
DO i=1,3
DO j=1,3
  g%fc(i,j,xyzt)=f1%fc(i,xyzt,1) *CONJG(f2%fc(j,xyzt,1))
  DO s=2,4
    g%fc(i,j,xyzt)=g%fc(i,j,xyzt)   &
        +f1%fc(i,xyzt,s)*CONJG(f2%fc(j,xyzt,s))
  END DO
END DO
END DO
END DO
```

$\mathtt{ge}_1 * \mathtt{ge}_2$: this operation returns a real field having as site components the scalar product of the site components of the generators.

$\mathtt{g}_1$.Dot.$\mathtt{g}_2$: this operation returns a real field having as site components the the real part of the trace of the product of the Hermitian adjoint of the site components of the gauge field $\mathtt{g}_1$ with the site components of the gauge field $\mathtt{g}_2$.

The following named operators are also defined:

i.Gamma.f , where i is a scalar integer. This operation returns a Fermi field having as site components the product of a single γ matrix or of a pair of γ matrices times the site components $\psi_{\mathbf{x}}$ of the Fermi field f. Our convention is as follows. The integer variable i can take value 1 through 5 or value $10 * i_1 + i_2$, where i_1 and i_2 can again range from 1 to 5. In the former case the operator implements the product $\gamma_i \psi_{\mathbf{x}}$. In the latter case the pair i_1, i_2 stands for two indices labeling a matrix $\gamma_{i_1\, i_2}$, where $\gamma_{i_1\, i_2} = \frac{i}{2}[\gamma_{i_1}\gamma_{i_2} - \gamma_{i_2}\gamma_{i_1}]$, $\gamma_{i\,5} = -\gamma_{5\,i} = \gamma_i\gamma_5$ with $i, i_1, i_2 = 1 \ldots 4$, and the operator implements the product $\gamma_{i_1\, i_2}\psi_{\mathbf{x}}$. Thus, for instance, $\mathtt{i} = 25; \mathtt{f1} =$ i.Gamma.f2 would implement $\psi_{1\mathbf{x}} = \gamma_2\gamma_5\psi_{2\mathbf{x}}$. Products of γ matrices have been explicitly incorporated in the definition of the .Gamma. operator because they are frequently encountered in the evaluation of matrix elements of fermionic variables.

`f.Gamma.i` , where `i` is a scalar integer. This operation returns a Fermi field having as site components the product of site components of the Fermi field `f` times a single γ matrix or of a pair of γ matrices, following the same convention about the values of `i` as above.
`i.Lambda.g` , where `i` is a scalar integer. This operation returns a gauge field having as site components the product of the matrix λ_i times the site components of the gauge field `g`.
`g.Lambda.i` , where `i` is a scalar integer. This operation returns a gauge field having as site components the product of the site components of the gauge field `g` times the matrix λ_i.

In addition we define the following unary operators:

`.I.`, `.Minus.`, `.Conjg.`, `.Adj.`, `.Ctr.` `.Tr.` `.Sqrt.` and `.Exp.`

When acting on a variable of type gauge_field, fermi_field or complex_field `.I.` returns $\imath$ times the variable. When acting on a variable of type real_field it returns a complex field given by $\imath$ times the real field. This is introduced for efficiency, since the operator is implemented by switching real and imaginary parts with the appropriate change of sign, rather than through a complex multiplication.

When acting on a variable of type gauge_field, fermi_field, complex_field, real_field or generator_field, `.Minus.` returns the negative of the variable.

When acting on a variable of type gauge_field, fermi_field, complex_field or matrix `.Conjg.` returns the complex conjugate of the variable, i.e. a variable whose complex components are the complex conjugate of the original one.

When acting on a variable of type gauge_field or matrix `.Adj.` returns the Hermitian adjoint of the variable.

When acting on a variable of type gauge_field or matrix `.Ctr.` returns a complex_field or complex number, respectively, equal to the trace (at each site in the case of a field) of the operand.

When acting on a variable of type gauge_field or matrix `.Tr.` returns a real_field or real number, respectively, equal to the real part of the trace (at each site in the case of a field) of the operand.

When acting on a variable of type real_field .Sqrt. returns a real_field having as site components the square root of the absolute value of the site components of the operand. At the same time the global variable context is set to .TRUE. at all sites where the operand is non-negative and to .FALSE. at all other sites.

When acting on a variable of type real_field .Exp. returns a real_field having as site components the exponential of the site components of the operand.

7.5 Modules assign_isotype1, assign_isotype2, assign_isotype3 and assign_mixed

These modules define the normal assignment and a variety of overloaded assignments which are defined for efficiency (cf. Sect. 6.2 above) and programming convenience. They are presented as four separate modules (assign_isotype1, assign_isotype2 and assign_isotype3 define assignments between variables of the same type, assign_mixed between variables of different type) to reduce the overall length of the individual modules. We reproduce here all the available assignments. We use the notational conventions we introduced at the beginning of Sect. 7.4. Namely, we use the symbols g, u, f, c, r, ge and m to denote variables of type gauge_field, full_gauge_field, fermi_field, complex_field, real_field, generator_field and matrix, respectively, and the symbols complex and real to denote a complex or real variable of kind REAL8 (cf. Sect. 7.1). Also, we use subscripts, e.g. $\mathtt{f}_1, \mathtt{f}_2$, to distinguish between two variables of the same type.

When the assignment relies on the the fact that the global variables assign_type and assign_spec have a value different from their default values $' ='$ and 0, we will denote this fact by the using the combined symbols assign_type(assign_spec) = to denote the assignment. For example, we would use $\mathtt{f}_1 \; + = \; \mathtt{f}_2$ or $\mathtt{f}_1 \; \mathtt{U(2)} = \mathtt{f}_2$ to denote assignments which in the actual coding would be implemented as

`assign_type='+'; f1=f2` , or
`assign_type='U'; assign_spec=2; f1=f2` , respectively.

A general rule is that all assignments set the global variables `assign_type` and `assign_spec` equal to their default values $'='$ and 0, no matter what the assignment does. As discussed in Sect. 6.2, this is done in order to avoid the accidental use of erroneous assignments.

For the `parity` component, the rule is that, if the destination is not an operand in the assignment (i.e. it is a variable with strict `INTENT(OUT)`), the `parity` component (if present) of the variable at the l.h.s. of the assignment (destination) is set equal to the `parity` of the variable at the r.h.s. of the assignment (source), or set to -1 if the source has no `parity`. Similarly, when the destination is not an operand in the assignment and has a `dir` component, this is set equal to the `dir` of the source or to `0` if the source has no `dir`. An exception to the rule above about the `parity` component occurs with the `assign_type` $='$ `u`$'$, `assign_type` $='$ `w`$'$ and `assign_type` $='$ `x`$'$ assignments, which copy into the destination a shifted source. In this case, if the `parity` of the source is defined, the `parity` of the destination is set to the opposite value.

If the destination is an operand in the assignment (i.e. it is a variable with `INTENT(INOUT)`) `parity` and `dir` are treated in a manner similar to what happens in the definition of the operators implemented by the overloaded assignment. Typically, if the destination and the other operand have the same `parity`, this is preserved, otherwise the `parity` of the destination is set to -1 (undefined). An exception is found in the assignments `U` $=$ and `W` $=$ which implement the sum of the destination with a shifted operand, in which case the `parity` of the destination is preserved if the other operand has the opposite `parity` (as is the case in a geometrically meaningful operation) and is returned undefined otherwise.

In what follows we list all of the available assignments and define their action, appending a few words of explanation when appropriate. When the

assignment is not followed by further clarifications, it means that it is a straightforward assignment (with `assign_type` $'='$) copying the content of the source into the destination. Also, whenever the assignment implements operations which can be performed by using overloaded operators, we illustrate its action simply by reformulating it in terms of these operators. We refer to the sections detailing the modules where the overloaded operators are defined for clarification of their action.

The assignments are listed in order of destination type, first, and then of source type. The ordering of the types is the same as their order of introduction in Sect. 5.

Available assignments:

$\mathtt{g}_1 = \mathtt{g}_2$

$\mathtt{g}_1 \mathrel{+}= \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_1 + \mathtt{g}_2)$

$\mathtt{g}_1 \mathrel{-}= \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_1 - \mathtt{g}_2)$

$\mathtt{g}_1 \mathrel{*}(0) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_1 * \mathtt{g}_2)$

$\mathtt{g}_1 \mathrel{*}(-1) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_2 * \mathtt{g}_1)$

Notice how the `assign_spec` variable is used, above and immediately below, to specify the order of the operands in the non-commutative matrix multiplication.

$\mathtt{g}_1 /(0) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_1/\mathtt{g}_2)$

$\mathtt{g}_1 /(-1) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_2//\mathtt{g}_1)$

$\mathtt{g}_1\ \mathtt{u}(\mathtt{dir}) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{dir.Ushift.g}_2)$

$\mathtt{g}_1\ \mathtt{U}(\mathtt{dir}) = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{g}_1 + (\mathtt{dir.Ushift.g}_2))$

$\mathtt{g}_1\ \mathtt{t} = \mathtt{g}_2$ ($\mathtt{g}_1 = \mathtt{g}_2$ where `context` is `.TRUE.`)

$\mathtt{g}_1\ \mathtt{f} = \mathtt{g}_2$ ($\mathtt{g}_1 = \mathtt{g}_2$ where `context` is `.FALSE.`)

$\mathtt{g}_1\ \mathtt{A} = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{.Adj.g}_2)$

$\mathtt{g}_1\ \mathtt{C} = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{.Conjg.g}_2)$

$\mathtt{g}_1\ \mathtt{I} = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{.I.g}_2)$

$\mathtt{g}_1\ \mathtt{M} = \mathtt{g}_2 \qquad (\mathtt{g}_1 = \mathtt{.Minus.g}_2)$

$\mathtt{g} = \mathtt{u}$ ($\mathtt{u}\%\mathtt{uc}(\mathtt{g}\%\mathtt{parity}, \mathtt{g}\%\mathtt{dir})$ is copied to $\mathtt{g}\%\mathtt{fc}$)

g t = u (same as above, but only where context is .TRUE.)

g f = u (same as two lines above, but only where context is .FALSE.)

g = ge (g = .Matrix.ge)

g E = ge (g = .Exp.ge)

g = m (all elements of g are set equal to m)

g *(0) = m (g = g * m)

g *(−1) = m (g = m * g)

g * = complex (g = g * complex)

g * = real (g = g * real)

g / = complex (g = g/complex)

g / = real (g = g/real)

u = g (g%fc is copied to u%uc(g%parity, g%dir))

u t = g (same as above, but only where context is .TRUE.)

u f = g (same as two lines above, but only where context is .FALSE.)

$u_1 = u_2$

f *(0) = g (f = f * g)

f *(−1) = g (f = g * f)

f /(0) = g (f = f/g)

f /(−1) = g (f = g//f)

(Note the function played by assign_spec in the four preceding assignments.)

$f_1 = f_2$

$f_1 + = f_2$ ($f_1 = f_1 + f_2$)

$f_1 - = f_2$ ($f_1 = f_1 - f_2$)

f_1 u(dir) = f_2 (f_1 = dir.Ushift.f_2)

f_1 U(dir) = f_2 ($f_1 = f_1$ + (dir.Ushift.f_2))

f_1 w(dir) = f_2 (f_1 = dir.Wshift.f_2)

f_1 W(dir) = f_2 ($f_1 = f_1$ + (dir.Wshift.f_2))

f_1 x(dir) = f_2 (f_1 = dir.Xshift.f_2)

f_1 X(dir) = f_2 ($f_1 = f_1$ + (dir.Xshift.f_2))

$\mathtt{f_1\ C = f_2} \qquad (\mathtt{f_1 = .Conjg.f_2})$

$\mathtt{f_1\ I = f_2} \qquad (\mathtt{f_1 = .I.f_2})$

$\mathtt{f_1\ M = f_2} \qquad (\mathtt{f_1 = .Minus.f_2})$

$\mathtt{f *= c} \qquad (\mathtt{f = f * c})$

$\mathtt{f\ /= c} \qquad (\mathtt{f = f/c})$

$\mathtt{f *= r} \qquad (\mathtt{f = f * r})$

$\mathtt{f\ /= r} \qquad (\mathtt{f = f/r})$

$\mathtt{f *= complex} \qquad (\mathtt{f = f * complex})$

$\mathtt{f *= real} \qquad (\mathtt{f = f * real})$

$\mathtt{f\ /= complex} \qquad (\mathtt{f = f/complex})$

$\mathtt{f\ /= real} \qquad (\mathtt{f = f/real})$

$\mathtt{c = g} \qquad (\mathtt{c = .Ctr.g})$

$\mathtt{c_1 = c_2}$

$\mathtt{c_1 += c_2} \qquad (\mathtt{c_1 = c_1 + c_2})$

$\mathtt{c_1 -= c_2} \qquad (\mathtt{c_1 = c_1 - c_2})$

$\mathtt{c_1 *= c_2} \qquad (\mathtt{c_1 = c_1 * c_2})$

$\mathtt{c_1\ /= c_2} \qquad (\mathtt{c_1 = c_1/c_2})$

$\mathtt{c_1\ C = c_2} \qquad (\mathtt{c_1 = .Conjg.c_2})$

$\mathtt{c_1\ M = c_2} \qquad (\mathtt{c_1 = .Minus.c_2})$

$\mathtt{c_1\ I = c_2} \qquad (\mathtt{c_1 = .I.c_2})$

$\mathtt{c = r}$

$\mathtt{c += r} \qquad (\mathtt{c = c + r})$

$\mathtt{c -= r} \qquad (\mathtt{c = c - r})$

$\mathtt{c *= r} \qquad (\mathtt{c = c * r})$

$\mathtt{c\ /= r} \qquad (\mathtt{c = c/r})$

$\mathtt{c\ M = r} \qquad (\mathtt{c = .Minus.r})$

$\mathtt{c = complex}$

$\mathtt{c += complex} \qquad (\mathtt{c = c + complex})$

$\mathtt{c -= complex} \qquad (\mathtt{c = c - complex})$

$\mathtt{c *= complex} \qquad (\mathtt{c = c * complex})$

c / = complex (c = c/complex)

c M = complex (c = .Minus.complex)

c = real

c + = real (c = c + real)

c − = real (c = c − real)

c ∗ = real (c = c ∗ real)

c / = real (c = c/real)

c M = real (c = .Minus.real)

r = g (r = .Tr.g)

r = f (r = f ∗ f)

r = c the elements of r are set equal to the real part of the elements of c

$r_1 = r_2$

$r_1 + = r_2$ $(r_1 = r_1 + r_2)$

$r_1 - = r_2$ $(r_1 = r_1 - r_2)$

$r_1 * = r_2$ $(r_1 = r_1 * r_2)$

$r_1 / = r_2$ $(r_1 = r_1/r_2)$

r_1 M = r_2 $(r_1 = .Minus.r_2)$

r_1 R = r_2 $(r_1 = .Sqrt.r_2)$

r_1 E = r_2 $(r_1 = .Exp.r_2)$

r = real

r + = real (r = r + real)

r − = real (r = r − real)

r ∗ = real (r = r ∗ real)

r / = real (r = r/real)

r M = real (r = .Minus.real)

ge = g (ge = .Generator.g)

$ge_1 = ge_2$

$ge_1 + = ge_2$ $(ge_1 = ge_1 + ge_2)$

$ge_1 - = ge_2$ $(ge_1 = ge_1 - ge_2)$

ge_1 M = ge_2 $(ge_1 = .Minus.ge_2)$

$\texttt{ge}_1\ \texttt{S} = \texttt{ge}_2 \qquad (\texttt{ge}_1 = .\texttt{Sq}.\texttt{ge}_2)$

$\texttt{ge} * = \texttt{r} \qquad (\texttt{ge} = \texttt{ge} * \texttt{r})$

$\texttt{ge} \ / = \texttt{r} \qquad (\texttt{ge} = \texttt{ge}/\texttt{r})$

$\texttt{ge} * = \texttt{real} \qquad (\texttt{ge} = \texttt{ge} * \texttt{real})$

$\texttt{ge} \ / = \texttt{real} \qquad (\texttt{ge} = \texttt{ge}/\texttt{real})$

The following assignments perform global reductions, either absolute or restricted to the lattice sites where `context` is .TRUE. or .FALSE.:

$\texttt{complex} = \texttt{c} \qquad (\texttt{complex} = \sum_{\texttt{xyzt}} \texttt{c}(\texttt{xyzt}))$

$\texttt{complex t} = \texttt{c} \qquad (\texttt{complex} = \sum_{\texttt{WHERE(context(xyzt))}} \texttt{c}(\texttt{xyzt}))$

$\texttt{complex f} = \texttt{c} \qquad (\texttt{complex} = \sum_{\texttt{WHERE(.NOT.context(xyzt))}} \texttt{c}(\texttt{xyzt}))$

$\texttt{real} = \texttt{c} \qquad (\texttt{real} = \sum_{\texttt{xyzt}} \texttt{Real}[\texttt{c}(\texttt{xyzt})])$

$\texttt{real t} = \texttt{c} \qquad (\texttt{real} = \sum_{\texttt{WHERE(context(xyzt))}} \texttt{Real}[\texttt{c}(\texttt{xyzt})])$

$\texttt{real f} = \texttt{c} \qquad (\texttt{real} = \sum_{\texttt{WHERE(.NOT.context(xyzt))}} \texttt{Real}[\texttt{c}(\texttt{xyzt})])$

$\texttt{real} = \texttt{r} \qquad (\texttt{real} = \sum_{\texttt{xyzt}} \texttt{r}(\texttt{xyzt}))$

$\texttt{real t} = \texttt{r} \qquad (\texttt{real} = \sum_{\texttt{WHERE(context(xyzt))}} \texttt{r}(\texttt{xyzt}))$

$\texttt{real f} = \texttt{r} \qquad (\texttt{real} = \sum_{\texttt{WHERE(.NOT.context(xyzt))}} \texttt{r}(\texttt{xyzt}))$

7.6 Module shifts

This module defines the operators .Cshift., .Ushift., .Wshift. and .Xshift. The left operand for all these operators is an integer `m` which must take one of the values $1, 2, 3, 4, -1, -2, -3, -4$ and specifies the direction and orientation of the shift. The right operand can be any variable of field type for .Cshift.. It can be any variable of field type with the exception of the type generator_field for .Ushift., while it must be a variable of type fermi_field for .Wshift. and .Xshift. The `parity` component of the right operand must be defined, i.e. take value `0` or `1`. If the `parity` is not defined or if `m` does not take one of the values specified above the function call implementing the operator returns an error message and stops the program. All of these operators return a field variable of the same type as the left operand and opposite `parity`. If

the right operand is of the type gauge_field or generator_field and thus has also a `dir` component, this is passed to the result unchanged.

.`Cshift`. implements an ordinary C-shift of the field, but with respect to the Cartesian geometry of the lattice. This is why the parity is interchanged. Given a site with Cartesian coordinates $\mathbf{x}$ in the sublattice of the parity of the result, the operator copies into the corresponding element of the result the element of the right operand which is defined over the lattice site $\mathbf{x} + s\hat{\mu}$, where s is the sign of `m` and $\mu = \mathrm{abs}(\mathtt{m})$.

.`Ushift`. moves the data in a manner similar to .`Cshift`., but with the inclusion of the appropriate transport factors, defined in terms of the global field `U` (cf. "global module"). For the variable of type gauge_field and fermi_field the U-shift operation has been defined in Sect. 3 (cf. Eqs. (13), (14), and Eqs. (8), (12), for gauge fields and Fermi fields, respectively). The action of a U-shift on variables of type complex_field and real_field reduces to a C-shift. The U-shift of a generator field is not normally encountered in QCD simulations and for this reason it is not explicitly implemented here. It can be implemented by using the functionality provided by the module generator_algebra to re-express the generator field as a field of hermitian matrices (i.e. of type gauge_field), shifting the latter, and converting it again to a generator_field.

The operator .`Wshift`. acts only on Fermi fields and it is a combination of a .`Ushift`. and the product with a γ matrix. Precisely, if we again define s to be the sign of `m` and define μ or `mu` to be the absolute value of `m`, then the operation `f2 = m.Wshift.f1` is equivalent to

```
f2=m.Ushift.f1-s*(mu.Gamma.(m.Ushift.f1))
```

Equivalently

$$f_{2,\mathbf{x}} = (1 - \gamma_\mu) U^\mu_{\mathbf{x}} f_{1,\mathbf{x}+\hat{\mu}} \tag{21}$$

for positive `m`, and

$$f_{2,\mathbf{x}} = (1 + \gamma_\mu) U^{\dagger\mu}_{\mathbf{x}-\hat{\mu}} f_{1,\mathbf{x}-\hat{\mu}} \tag{22}$$

for negative `m`.

The operator .`Xshift`. also acts only on Fermi fields and it is equivalent to a W-shift bracketed by two matrices γ_5 (where $\gamma_5 = \gamma_1\gamma_2\gamma_3\gamma_4$):

$$f_{2,\mathbf{x}} = \gamma_5(1-\gamma_\mu)\gamma_5 U^\mu_{\mathbf{x}} f_{1,\mathbf{x}+\hat{\mu}} \tag{23}$$

for positive `m`, and

$$f_{2,\mathbf{x}} = \gamma_5(1+\gamma_\mu)\gamma_5 U^{\dagger\mu}_{\mathbf{x}-\hat{\mu}} f_{1,\mathbf{x}-\hat{\mu}} \tag{24}$$

for negative `m`.

The reason for the explicit introduction of the W-shift and X-shift operators is that the combinations (21-24) appear in the Wilson discretization of the Dirac operator, which is a very widely used lattice Dirac operator, and related algebra, and that the calculation of the l.h.s. of Eqs. (21-24) is one of the most time consuming tasks of any QCD simulation. Moreover, the combinations $1 \pm \gamma_\mu$ appearing in (21-24) are projection operators, which effectively limits the U-shift to a subspace of the spin space of dimensionality two. Thus a direct implementation of the W-shift, rather than via a combination of the .`Ushift`. and .`Gamma`. operators, entails substantial advantages of efficiency.

7.7 Module Dirac_operator

The (Wilson) lattice Dirac operator, acting on a Fermi field f_1, produces a Fermi field f_2, given by

$$f_{2,\mathbf{x}} = \sum_\mu [(1-\gamma_\mu)U^\mu_{\mathbf{x}} f_{1,\mathbf{x}+\hat{\mu}} + (1+\gamma_\mu)U^{\dagger\mu}_{\mathbf{x}-\hat{\mu}} f_{1,\mathbf{x}-\hat{\mu}}] \tag{25}$$

It is obvious from this equation that the lattice Dirac operator only connects components of Fermi fields of opposite parity. The unary operator .`Dirac`. accepts as operand a variable of type fermi_field, which must have a definite `parity`, and returns a variable of the same type and opposite `parity` given

by the action of the lattice Dirac operator (25) on the operand. The unary operator .XDirac. implements the action of the Dirac operators bracketed by two matrices γ_5, i.e. , if f is a variable of type fermi_field, .XDirac.f returns the same results as i5.Gamma.(.XDirac.(i5.Gamma.f)), where the integer variable i5 equals 5.

In this module the operators .Dirac. and .XDirac. are implemented using the operators .Wshift. and .Xshift., introduced in the module field_algebra. We have defined them as separate operators for convenience of coding and also because, the application of these operators being the most CPU intensive part for the majority of applications, this module isolates the code whose optimization would produce the largest returns. A programmer striving for exceptional efficiency might want to code this module as a highly optimized, self-standing implementation of the lattice Dirac operator. Even if this route is chosen, we are certain that the advantages of having a module written at a higher level against which to compare the results of the optimized module are not lost on the practicing computational scientist.

7.8 Module generator_algebra

This module defines the unary operators .Matrix., .Generator., .Sq. and .Exp. which perform some special operations involving generator fields. The operators accept arguments of the type generator_field or gauge_field and return as result a variable of one of these types. The parity and dir components of the argument are passed on to the result.

.Matrix. accepts an argument ge of type generator_field and returns a result v of type gauge_field containing, site by site, the Hermitian matrix

$$v_{ij,\mathbf{x}} = \sum_k \lambda^k_{ij}\, ge_{k,\mathbf{x}} \ . \tag{26}$$

.Generator. accepts an argument v of the type gauge_field and returns a result ge of type generator_field containing, site by site, the traceless, antihermitian part of the argument:

$$ge_{k,\mathbf{x}} = -\frac{\imath}{4}\sum_{ij}\lambda^k_{ij}\,(v_{ji,\mathbf{x}} - v^*_{ij,\mathbf{x}})\ . \tag{27}$$

Notice that .Generator. and .Matrix. are not inverse operators. However .Generator.(.Matrix.(IU∗ge)) does return ge, while .Matrix.(.Generator.v) returns the traceless, antihermitian part of v: $v_{AH} = (v - v^\dagger)/(2\imath)$.

.Sq. accepts an argument of type generator_field $\mathtt{ge}_1$ and returns a result $\mathtt{ge}_2$ of the same type containing, site by site, the generator corresponding to the traceless part of the square of .Matrix.ge1:

$$(ge_2)_{l,\mathbf{x}} = \frac{1}{2}\sum_{ijk}[\lambda^l_{ij}\,(\sum_m \lambda^m_{jk}\,(ge_1)_{m,\mathbf{x}})\,(\sum_n \lambda^n_{ki}\,(ge_1)_{n,\mathbf{x}})]\ . \tag{28}$$

.Exp. accepts an argument ge of type generator_field and returns a result v of type gauge_field containing, site by site, the exponentiated generator component:

$$v_{ij,\mathbf{x}} = [\,\exp{(\imath\sum_k \lambda^k\, ge_{k,\mathbf{x}})}]_{ij}\ . \tag{29}$$

The algorithm used for this exponentiation deserves a few words of explanation. Let us define $H = \sum_k \lambda^k_{ij}\, ge_{k,\mathbf{x}}$, $q = \mathrm{Tr}H^2$ and $p = \mathrm{Det}H = (\mathrm{Tr}H^3)/3$. From the characteristic equation (recall that $\mathrm{Tr}H = 0$)

$$H^3 - \frac{q}{2}H - pI = 0\ , \tag{30}$$

I being the identity matrix, satisfied by H and therefore by its eigenvalues h_n, we can easily calculate the eigenvalues as

$$h_n = a\cos[\alpha + 2\pi(n-1)/3], \quad n = 1,2,3\ , \tag{31}$$

with $a = \sqrt{2q/3}$, $\alpha = [\cos^{-1}(4p/a^3)]/3$. We order the eigenvalues so that $|h_1| \geq |h_2| \geq |h_3|$.

In a basis where H is diagonal, it is easy to express it as a linear combination of two matrices of type "λ^3" (one 0 eigenvalue) and "λ^8" (two degenerate eigenvalues), respectively. Of the six different ways in which this can be done we use the decomposition

$$H = S + K \tag{32}$$

with $S = \mathrm{diag}(-h_1 - 2h_2, h_1 + 2h_2, 0)$, $K = \mathrm{diag}(2(h_1 + h_2), -h_1 - h_2, -h_1 - h_2)$.

By using the eigenvalues determined above, it is straightforward to express S in the form

$$S = c_1 H + c_2(H^2 - qI/3) \ . \tag{33}$$

Through their dependence on the eigenvalues and Eq. (31), however, c_1 and c_2 are functions of the invariants q and p only. It follows that Eqs. (33) and (32) provide a decomposition into two matrices of type "λ^3" and "λ^8" irrespective of the basis. On the other hand, with a matrix of type "λ^3" it is straightforward to calculate $\exp(\imath S)$ expressing it as a linear combination of I, S and S^2. Similarly $\exp(\imath K)$ can be expressed as a linear combination of I and K. $\exp(\imath H)$ can be finally calculated as product of the two commuting matrices $\exp(\imath S)$ and $\exp(\imath K)$.

It is very important to have an efficient algorithm for the exponentiation of a matrix, since this operation can be a time consuming component of several QCD calculations. The algorithm outlined above has been implemented in the module "generator_algebra" by performing a substantial amount of the algebra directly in terms of generator components and inlining all of the operations. The exponentiation can thus be done with a reasonably contained number of arithmetic operations, in particular approximately 300 explicit real multiplications. By way of comparison, just one product of 3×3 complex matrices requires 108 real multiplications (i.e. 27 complex multiplications – these could also be performed with 81 real multiplications, but then with a much larger number of adds).

7.9 Module random_numbers

This module implements a parallelizable version of the unix pseudorandom number generator erand48 and provides added functionality.

erand48 is a congruential pseudorandom number generator based on the iterative formula

$$s_{i+1} = a_1 * s_i + b_1 \mod m \ , \tag{34}$$

where a_1 = `0x5DEECE66D`, b_1 = `0xB`, $m = 2^{48}$, s_i and s_{i+1} are integers of at least 48 bits of precision. The "seeds" s_i are converted to real pseudorandom numbers r_i with uniform distribution between 0 and 1 by $r_i = 2^{-48}\, s_i$.

As presented above, the algorithm is intrinsically serial. However it follows from Eq. (34) that the N^{th} iterate s_{i+N} is still of the form $s_{i+N} = a_N * s_i + b_N \mod 2^{48}$ with integers a_N and b_N which are uniquely determined by a_1, b_1. The module takes advantage of this fact and of the existence of the global variable `seeds` (cf. global_module) to generate pseudorandom numbers in a parallelizable fashion.

The module defines the following unary operators: `.Seed.`, `.Rand.`, `.Gauss.` and `.Ggauss.`.

`.Seed.` must be used to initialize the generation of pseudorandom numbers. When invoked with an argument `saved_seed` of kind `LONG` (8-byte integer, defined in the module precisions) `.Seed.` sets the global variable `seeds` to the sequence (34) beginning with `saved_seed` and also sets the global variables `seed_a`, `seed_b` to the appropriate multiplier and constant term, a_N and b_N, needed to produce increments by $N = NX*NY*NZ*NT/2$ in the sequence of pseudorandom numbers. It returns `.TRUE.`.

When acting on a logical variable equal to `.TRUE.`, `.Seed.` returns the current seed (=`seeds(0,0,0,0)`), which must be used to restart the sequence of pseudorandom numbers. If the argument is `.FALSE.`, `.Seed.` returns 0.

The unary operator `.Rand.`, if invoked with a real argument `real` of kind `REAL8`, returns a real_field of pseudorandom numbers with uniform distribution between 0 and `real`. At the same time it upgrades the global variables `seeds` using the multiplier a_N (i.e. `seed_a`) and constant term b_N

(i.e. seed_b). It follows that subsequent calls to .Rand. produce real fields with the same distribution of pseudorandom numbers which one would have obtained invoking erand48 sequentially within nested DO loops:

```
DO xyzt=0,NXYZT2-1
...
```

The parity of the results is undefined.

If .Rand. has an argument of type real_field, it returns a real_field of pseudorandom numbers uniformely distributed between 0 and the corresponding component of the argument. The parity of the result is the same as the parity of the argument.

The unary operator .Gauss. returns a real field of pseudorandom numbers with gaussian distribution of mean square deviation equal to the argument of .Gauss. and upgrades the global variable seeds. The argument can again be a variable of kind REAL8 or of type real_field and the parity of the result is undefined or equal to the parity of the argument, respectively.

The unary operator .Ggauss. works like .Gauss. but fills with gaussian random numbers the components of a generator_field, setting its direction equal to 0. Precisely, the instruction ge = .Ggauss.r, although in the module it is implemented differently, would be equivalent to

```
DO i=1,8
  auxr=.Gauss.r
  ge%fc(i,:,:,:,:)=auxr%fc
END DO
ge%parity=auxr%parity
ge%dir=0
```

where auxr is a variable of type real_field

This module assumes the availability of long (8-byte) integers and the fact that a multiplication of long integers will return the lowest 8 bytes of the product (i.e. $a * b \bmod 2^{64}$) without producing an arithmetic error when the product exceeds the maximum long integer. If these assumptions are not

satisfied, the module should be replaced with some other suitable module. Also, we would like to point out that the algorithm of Eq. (34) produces pseudorandom numbers of reasonably good quality and period ($\approx 10^{14}$). However, a computer capable of 100 Gflops sustained running a program that makes use of one pseudorandom number every thousand floating point operations would exhaust the whole set of pseudorandom numbers in one million seconds, which is not a very long time. Thus for calculations that are very computer intensive or which demand pseudorandom numbers of exceptionally good quality, the module should be modified to meet the more stringent demands. Two improvements which can be made with minimal changes would consist in the use of a larger m in Eq. (34) (with appropriate a_1 and b_1) and/or of a reshuffle of the pseudorandom numbers produced by the algorithm. Of course, one could also make use of the F90 RANDOM_NUMBER subroutine, but the results would no longer be machine independent.

7.10 Module conditionals

This module defines six overloaded relational operators, >, >=, <, <=, ==, / =, and the operator .`Xor`.

The relational operators take as operands two real_fields or one real_field and one real variable of kind `REAL8`. They return a logical variable which is set to .`TRUE`. if the two fields have the same (defined) parity or if the single field operand has defined parity, and is set to .`FALSE`. otherwise. At the same time the global variable `context` is set to .`TRUE`. at all sites where the relation is satisfied, to .`FALSE`. at all other sites. For example, the function implementing the relational operator `a` > `b`, with `a` and `b` of type real_field, could contain a line: `context` = `a`%`fc` > `b`%`fc` , which produces the action mentioned above.

The operator .`Xor`. accepts as operands a pair of fields of the same type and returns a field, also of the same type, having as elements the corresponding elements of the first operand at the sites where the global variable

context is .TRUE., the elements of the second operand at the sites where context is .FALSE.. For clarification, the function g_xor_g implementing the operation g1.Xor.g2, where g1 and g2 are fields of type gauge_field, would contain the code

```
DO j=1,3
DO i=1,3
  WHERE(context)
    g_xor_g%fc(i,j,:,:,:,:)=g1%fc(i,j,:,:,:,:)
  ELSEWHERE
    g_xor_g%fc(i,j,:,:,:,:)=g2%fc(i,j,:,:,:,:)
  END WHERE
END DO
END DO
```

The parity of the field returned by .Xor. is the common parity of the two operands if they have the same parity, otherwise it is undefined. In addition, for operands of type gauge_field, the dir component of the returned field is the common dir of the operands if they have the same dir, otherwise it is set to 0.

The operators provided by the module "conditionals" can be very conveniently used to select elements out of two fields according to some local condition, an operation which lies at the foundation of stochastic simulation techniques.

7.11 Subroutine write_conf and read_conf

The file "write_read_conf.f90" contains two subroutines which serve to store and retrieve an entire SU(3) gauge field configuration, written in a portable, compressed ASCII format. Only the first two columns of the gauge field matrices are stored, because the third one can be recovered from the unitarity and unimodularity constraints. The write_conf subroutine takes advantage of the fact that all of the elements of the gauge field matrices have magnitude smaller or equal to 1 to re-express their real and imaginary parts as 48bit

integers. These integers are then written in base 64, with the digits being given by the ASCII collating sequence starting from 0. Thus, 8 characters are needed to represent either the real or the imaginary part of the gauge field matrix elements and an entire gauge field matrix is represented by 96 ASCII characters, without loss of numerical information. A detailed description of the contents of the file generated by write_conf and of the standard used for coding the information is written, as a header, at the beginning of the file itself. This makes the file with the compressed gauge configuration portable and usable, irrespective of the availability of the write_conf and read_conf subroutines or of a separate description of their implementation.

8 Example code

In order to illustrate the power of the modules we developed, we present here two programs which implement a multihit Metropolis simulation of quenched QCD and the calculation of a quark propagator. Anybody familiar with the complexity of the programs for lattice QCD simulations will appreciate the conciseness of our examples. It is also to be noticed that a large amount of the code in the programs performs peripheral functions, such as initialization and printout. If we consider the Metropolis simulation program, for instance, the section of the code which performs the actual upgrading steps consists of only 28 lines! It is our hope that researchers interested in using our modules will find it easy to become familiar with their functionality and that, not being hindered by inessential programming burdens, they will thus be able to make much faster progress in the development of far-reaching QCD applications.

8.1 quenched.f90

```
!     Program Qcdf90_quenched

!     Copyright by Indranil Dasgupta, Andrea R. Levi, Vittorio Lubicz
!     and Claudio Rebbi  -  Boston University  -  May 1996
```

```
!    This program may be freely copied and used as long as this notice
!    is retained.

PROGRAM Qcdf90_quenched

   USE precisions
   USE constants
   USE global_module
   USE field_algebra
   USE generator_algebra
   USE conditionals
   USE shift
   USE random_numbers
   USE assign_mixed
   USE assign_isotype1

   IMPLICIT NONE
   TYPE(gauge_field)::  staple,g_old,g_new
   TYPE(real_field)::  plaq_old,plaq_new,bf_ratio,rand
   TYPE(generator_field)::  ge
   TYPE(matrix) ::  zero_matrix, unit_matrix
   LOGICAL l_test,l_seed
   REAL(REAL8) clock_dcl,clock_upd,clock_plaq
   REAL(REAL8) beta,saved_beta,hp,av_plaq,aux,range_small,range_unit
   CHARACTER(LEN=64) in_filename,out_filename
   CHARACTER(LEN=16) id
   INTEGER(LONG) saved_seed,inp_seed
   INTEGER clock_rate,clock_1,clock_2
   INTEGER hotcoldread,save,num_upd,p,m,sign,nu,i,hit,num_hit

!  input variables:
   WRITE (*,'("Lattice size:  ",4I5)') NX,NY,NZ,NT
   WRITE (*,'("Enter beta:      ")',ADVANCE='NO')
   READ *,beta
   WRITE (*,'("Enter number of updates:      ")',ADVANCE='NO')
   READ *,num_upd
   WRITE (*,'("Select the starting configuration.    Enter 0 for&
```

```
              &a hot start ")')
   WRITE (*,'("1 for a cold start, 2 to read from Disk:  ")',ADVANCE='NO')
   READ *,hotcoldread

!  other useful variables:
   num_hit=6                !  number of Metropolis multiple hits
   range_unit=1._REAL8      !  unitary range for the random numbers
   range_small=0.1_REAL8    !  range for the random numbers
   inp_seed=1               !  input seed for random numbers generator
   zero_matrix%mc=ZERO_m    !  0 matrix for initializing the staple
   in_filename= 'configuration.in'
   out_filename='configuration.out'

!  initializing system clock
   CALL SYSTEM_CLOCK(clock_1,clock_rate)
   clock_dcl=1._REAL8/clock_rate
   clock_upd=0._REAL8
   clock_plaq=0._REAL8

!  initializing random generator and gauge configuration
   SELECT CASE(hotcoldread)
   CASE(0)
     l_seed=.Seed.inp_seed
     DO p=0,1
     DO m=1,4
       ge=.Ggauss.range_unit
       u%uc(p,m)=.Exp.ge
       u%uc(p,m)%parity=p
       u%uc(p,m)%dir=m
     END DO
     END DO
   CASE(1)
     l_seed=.Seed.inp_seed
     unit_matrix%mc=UNIT
     DO p=0,1
     DO m=1,4
       u%uc(p,m)=unit_matrix
```

```
      u%uc(p,m)%parity=p
      u%uc(p,m)%dir=m
    END DO
    END DO
  CASE(2)
    CALL read_conf(saved_beta,id,hp,saved_seed,in_filename)
    IF(inp_seed==0) THEN
      WRITE (*,'("saved_seed=",I15)') saved_seed
      l_seed=.Seed.saved_seed
    ELSE
      l_seed=.Seed.inp_seed
      WRITE (*,'("seed re-initialized")')
    ENDIF
  CASE DEFAULT
    WRITE (*,'("hotcoldread must only be 0,1 or 2")')
    STOP
  END SELECT

  DO i=1,num_upd                          !  Main Loop

!  Metropolis update
     CALL SYSTEM_CLOCK(clock_1)
     DO p=0,1
     DO m=1,4

       !  Staple
       staple=zero_matrix
       staple%parity=p
       staple%dir=m
       DO nu=1,4
         IF(nu.EQ.m) CYCLE
         DO sign=-1,1,2
           staple=staple+((nu*sign).Ushift.u%uc(1-p,m))
         END DO
       END DO

       g_old=u%uc(p,m)
```

```
        DO hit=1,num_hit
          plaq_old=g_old.Dot.staple
          ge=.Ggauss.range_small
          ge%parity=p
          ge%dir=m
          g_new=(.Exp.ge)*g_old
          plaq_new=g_new.Dot.staple
          bf_ratio=.Exp.(beta/3._REAL8*(plaq_new-plaq_old))
          rand=.Rand.range_unit
          l_test=rand<bf_ratio
          assign_type='t'; g_old=g_new
        END DO
        u%uc(p,m)=g_old
      END DO
      END DO
      CALL SYSTEM_CLOCK(clock_2)
      clock_upd=clock_upd+(clock_2-clock_1)*clock_dcl

!  Plaquette
      CALL SYSTEM_CLOCK(clock_1)
      av_plaq=0._REAL8
      DO p=0,1
      DO m=1,3
        DO nu=m+1,4
          aux=u%uc(p,m).Dot.(nu.Ushift.u%uc(1-p,m))
          av_plaq=av_plaq+aux
        END DO
      END DO
      END DO
      av_plaq=av_plaq/REAL(18*NXYZT,REAL8)
      CALL SYSTEM_CLOCK(clock_2)
      clock_plaq=clock_plaq+(clock_2-clock_1)*clock_dcl
     WRITE (*,'("iteration ",I5," av.  plaq.= ",F10.6)') i,av_plaq

   END DO                                ! End Main Loop

!  Save configuration on disk
```

```
   WRITE (*,'("Save configuration on disk ?  (Yes=1, &
               &No=0):  ")',ADVANCE='NO')
   READ *,save
   IF(save==1) THEN
     WRITE (*,'("saving the configuration")')
     id='conf 0.0.0'
     hp=0.0
     l_seed=.TRUE.
     saved_seed=.Seed.l_seed
     WRITE (*,'("  saved_seed = ",I15)') saved_seed
     CALL write_conf(beta,id,hp,saved_seed,out_filename)
   ENDIF


!  Print timing
   WRITE (*,'("Av.  upgrade time in microsecs per link",F9.3)') &
            (1000000*clock_upd)/(4*NXYZT*num_upd)
   WRITE (*,'("Av.  measure time in microsecs per plaquette",F9.3)')&
            (1000000*clock_plaq)/(6*NXYZT*num_upd)

   END
```

8.2 propagator.f90

```
!    Program Qcdf90_propagator

!    Copyright by Indranil Dasgupta, Andrea R. Levi, Vittorio Lubicz
!    and Claudio Rebbi  -  Boston University  -  January 1996
!    This program may be freely copied and used as long as this notice
!    is retained.

PROGRAM Qcdf90_propagator

   USE precisions
   USE constants
```

```
   USE global_module
   USE field_algebra
   USE generator_algebra
   USE conditionals
   USE shift
   USE dirac
   USE random_numbers
   USE assign_mixed
   USE assign_isotype1
   USE assign_isotype2

   IMPLICIT NONE
   TYPE(fermi_field)::  psi,chi,grad,h,m_h,mp_m_h
   REAL(REAL8) clock_dcl,clock_cg
   REAL(REAL8) kappa,tolerance,residue,saved_beta,hp
   REAL(REAL8) alpha,old_alpha,g_2,g_old_2,beta_cg,old_beta_cg
   REAL(REAL8) h_a_h,norm_psi
   CHARACTER(LEN=64) in_filename
   CHARACTER(LEN=16) id
   INTEGER(LONG) saved_seed
   INTEGER clock_rate,clock_1,clock_2
   INTEGER iter,nsteps,niter,init_niter,stop_flag,init_stop_flag
   INTEGER i,xyzt,s

!  input variables:
   WRITE (*,'("Enter kappa:      ")',ADVANCE='NO')
   READ *,kappa
   WRITE (*,'("Enter max numbers of cg steps:      ")',ADVANCE='NO')
   READ *,nsteps
   WRITE (*,'("Enter tolerance:      ")',ADVANCE='NO')
   READ *,tolerance
                              !  the conjugated gradient will
                              !  run until the residue<tolerance
                              !  or for a maximum of nsteps

!  other useful variables:
   in_filename= 'configuration.in'
```

```
   init_stop_flag=2
   init_niter=4

!  gauge configuration is read from the disk
   CALL read_conf(saved_beta,id,hp,saved_seed,in_filename)

!  the source chi (in the even sites) is set arbitrarly in this example
   DO i=1,3
   DO s=1,4
   DO xyzt=0,NXYZT2-1
     chi%fc(i,xyzt,s)=(1._REAL8,1._REAL8)/SQRT(REAL(24*NXYZT2,REAL8))
   END DO
   END DO
   END DO
   chi%parity=0

!  psi must be initialized as the starting trial configuration.
!  the simplest choice is psi=chi
   psi=chi

!  initializing system clock
   CALL SYSTEM_CLOCK(clock_1,clock_rate)
   clock_dcl=1._REAL8/clock_rate

!    Calculate psi as the solution of:  M*psi=chi,
!    where M is the fermion matrix, and chi is a given source.
!    The residue is printed to monitor the convergence.

    stop_flag=init_stop_flag
    niter=init_niter
    iter=0
    m_h=psi-(kappa**2)*(.Dirac.(.Dirac.psi))
    mp_m_h=m_h-(kappa**2)*(.Xdirac.(.Xdirac.m_h))
    grad=chi-mp_m_h
    g_2=grad*grad
    h=grad
    norm_psi=psi*psi
```

```
residue=g_2/norm_psi
WRITE (*,'("residue= ",F20.16," at step:",I5)') residue,iter
old_alpha=0._REAL8
old_beta_cg=1._REAL8

DO iter=1,nsteps
  m_h=h-(kappa**2)*(.Dirac.(.Dirac.h))
  h_a_h=m_h*m_h
  beta_cg=g_2/h_a_h
  psi=psi+beta_cg*h
  norm_psi=psi*psi
  IF(mod(iter,niter)==0 .AND. g_2/norm_psi<tolerance) THEN
    stop_flag=stop_flag-1
    m_h=psi-(kappa**2)*(.Dirac.(.Dirac.psi))
    mp_m_h=m_h-(kappa**2)*(.Xdirac.(.Xdirac.m_h))
    grad=chi-mp_m_h
    g_2=grad*grad
    h=grad
    g_old_2=g_2
    residue=g_2/norm_psi
    WRITE (*,'("residue= ",F20.16," at step:",I5)') residue,iter
    IF(stop_flag == 0) EXIT
  ELSE
    mp_m_h=m_h-(kappa**2)*(.Xdirac.(.Xdirac.m_h))
    grad=grad-beta_cg*mp_m_h
    g_old_2=g_2
    g_2=grad*grad
    alpha=g_2/g_old_2
    h=grad+alpha*h
    norm_psi=psi*psi
    residue=g_2/norm_psi
    IF(mod(iter,niter) == 0) THEN
       WRITE (*,'("residue= ",F20.16," at step:",I5)') residue,iter
    ENDIF
    old_beta_cg=beta_cg
    old_alpha=alpha
  END IF
```

```
    END DO
    CALL SYSTEM_CLOCK(clock_2)
    clock_cg=(clock_2-clock_1)*clock_dcl

!test solution:
   m_h=psi-(kappa**2)*(.Dirac.(.Dirac.psi))
   mp_m_h=m_h-(kappa**2)*(.Xdirac.(.Xdirac.m_h))
   grad=chi-mp_m_h
   norm_psi=psi*psi
   g_2=grad*grad
   residue=g_2/norm_psi
   WRITE (*,'("final residue= ",F20.16)') residue

!  Print timing
   WRITE (*,'("Cg time per iteration per link in microsecs",F9.3)')
&
         (1000000*clock_cg)/(iter*4*NXYZT)

   END
```

8.3 Compilation and sample run output

The code has been tested and compiled on a SGI PowerChallengeArray with 90 MHz processor nodes, using IRIX 6.1 Fortran 90, with a single processor -O3 optimization flags or with the flags -O3 -pfa -mp to implement multi-processing; on a SGI Indigo using the IRIX 6.1 Fortran 90; and on the IBM R6000 58H model 7013 at 55 MHz, with the xlf90 IBM compiler using the -O3 optimization flags.

The run of the example programs produce the following outputs when running on a single processor of the SGI PowerChallengeArray.

Output of quenched.f90

```
Lattice size:          8    8    8    8
```

```
Enter beta:      6.0
Enter number of updates:      15
Select the starting configuration.    Enter 0 for a hot start
1 for a cold start, 2 to read from Disk:  1
iteration     1 av.  plaq.=   0.849923
iteration     2 av.  plaq.=   0.773278
iteration     3 av.  plaq.=   0.727001
iteration     4 av.  plaq.=   0.699791
iteration     5 av.  plaq.=   0.677709
iteration     6 av.  plaq.=   0.664358
iteration     7 av.  plaq.=   0.654980
iteration     8 av.  plaq.=   0.645880
iteration     9 av.  plaq.=   0.638568
iteration    10 av.  plaq.=   0.635049
iteration    11 av.  plaq.=   0.631868
iteration    12 av.  plaq.=   0.628131
iteration    13 av.  plaq.=   0.624450
iteration    14 av.  plaq.=   0.621757
iteration    15 av.  plaq.=   0.619540
Save configuration on disk ?  (Yes=1, No=0):  1
saving the configuration
  saved_seed = 182618478903297
Av.  upgrade time in microsecs per link  275.533
Av.  measure time in microsecs per plaquette    8.624
```

Output of propagator.f90

```
Enter kappa:      0.155
Enter max numbers of cg steps:      2000
Enter tolerance:      1.e-14
residue=   0.2417309784323778 at step:        0
residue=   0.0047398663491857 at step:        4
residue=   0.0007379739612745 at step:        8
residue=   0.0001969957469930 at step:      12
residue=   0.0000640891236947 at step:      16
residue=   0.0000235032623593 at step:      20
residue=   0.0000079006298388 at step:      24
```

```
residue=   0.0000030559375109 at step:      28
residue=   0.0000014286464825 at step:      32
residue=   0.0000006891018149 at step:      36
residue=   0.0000003502268064 at step:      40
residue=   0.0000002296348520 at step:      44
residue=   0.0000001023143421 at step:      48
residue=   0.0000000357544848 at step:      52
residue=   0.0000000114945632 at step:      56
residue=   0.0000000034230144 at step:      60
residue=   0.0000000010030646 at step:      64
residue=   0.0000000005153800 at step:      68
residue=   0.0000000003507320 at step:      72
residue=   0.0000000002150572 at step:      76
residue=   0.0000000000880532 at step:      80
residue=   0.0000000000289906 at step:      84
residue=   0.0000000000095210 at step:      88
residue=   0.0000000000042035 at step:      92
residue=   0.0000000000016253 at step:      96
residue=   0.0000000000004389 at step:    100
residue=   0.0000000000001207 at step:    104
residue=   0.0000000000000298 at step:    108
residue=   0.0000000000000066 at step:    112
residue=   0.0000000000000022 at step:    116
final residue=   0.0000000000000022
Cg time per iteration per link in microsecs   21.548
```

Acknowledgments

This research was supported in part under DOE grant DE-FG02-91ER40676. We are grateful to the Center of Computational Science and the Office of Information Technology for support and access to the Boston University supercomputer facility. V.L. acknowledges the support of an INFN post-doctoral fellowship.

References

[1] M. Creutz, L. Jacobs and C. Rebbi, Phys. Rep. **95** 201, (1983).

[2] "Lattice Gauge Theories and Monte Carlo Simulations," C. Rebbi ed., World Scientific Publishing Co., Singapore (1983).

[3] "Quantum Fields on the Computer", edited by M. Creutz, World Scientific, Singapore (1992).

[4] Proceedings of *Lattice 90-95*, Nucl. Phys. B (Proc. Suppl.) 20, 1991; 26, 1992; 30, 1993; 34, 1994; 42, 1995; 47, 1996.

[5] A. H. Muller, B. Müller C. Rebbi and W. H. Smith, Quantum Chromodynamics in "Particle Physics - Perspectives and Opportunities", Report of the DPF Committee on Long-Term Planning, R. Peccei, M. Zeller, D. Cassal, J. Bagger, R. Cahn, P. Grannis and F. Sciulli, eds., World Scientific Pub. Co., Singapore, (1995).

[6] K. Wilson, Phys. Rev. **D10** 2445, (1974).