tmLQCD Package Documentation

Contents

1 Theoretical Background

1.1 QCD on a lattice

Quantum Chromodynamics on a hyper-cubic Euclidean space-time lattice of size $L^3 \times T$ with lattice spacing a is formally described by the action

$$S = S_{\mathrm{G}}[U] + a^4 \sum_x \bar{\psi}\ D[U]\ \psi \tag{1}$$

with S_{G} some suitable discretisation of the the Yang-Mills action $F_{\mu\nu}^2/4$ [1]. The particular implementation we are using can be found below in section 4.2 and consists of plaquette and rectangular shaped Wilson loops with particular coefficients. D is a discretisation of the Dirac operator, for which Wilson originally proposed [2] to use the so called Wilson Dirac operator

$$D_W[U] = \frac{1}{2}\left[\gamma_\mu\left(\nabla_\mu + \nabla_\mu^*\right) - a\nabla_\mu^*\nabla_\mu\right] \tag{2}$$

with ∇_μ and ∇_μ^* the forward and backward gauge covariant difference operators, respectively:

$$\begin{aligned}\nabla_\mu\psi(x) &= \frac{1}{a}\Big[U(x, x+a\hat{\mu})\psi(x+a\hat{\mu}) - \psi(x)\Big]\,,\\ \nabla_\mu^*\psi(x) &= \frac{1}{a}\Big[\psi(x) - U^\dagger(x, x-a\hat{\mu})\psi(x-a\hat{\mu})\Big]\,,\end{aligned} \tag{3}$$

where we denote the SU(3) link variables by $U_{x,\mu}$. We shall set $a \equiv 1$ in the following for convenience. Discretising the theory is by far not a unique procedure. Instead of Wilson's original formulation one may equally well chose the Wilson twisted mass formulation and the corresponding Dirac operator [3]

$$D_{\mathrm{tm}} = (D_W[U] + m_0)\ 1_f + i\mu_q\gamma_5\tau^3 \tag{4}$$

for a mass degenerate doublet of quarks. We denote by m_0 the bare (Wilson) quark mass, μ_q is the bare twisted mass parameter, τ^i the i-th Pauli matrix and 1_f the unit matrix acting in flavour space (see appendix A for our convention). In the framework of Wilson twisted mass QCD only flavour doublets of quarks can be simulated, however, the two quarks do not need to be degenerate in mass. The corresponding mass non-degenerate flavour doublet reads [4]

$$D_h(\bar{\mu}, \bar{\epsilon}) = D_{\mathrm{W}}\ 1_f + i\bar{\mu}\gamma_5\tau^3 - \bar{\epsilon}\tau^1\,. \tag{5}$$

It has the property

$$D_h^\dagger = \tau^1\gamma_5 D_h \gamma_5 \tau^1\,.$$

Note that this notation is not unique. Equivalently – as used in Ref. [5] – one may write

$$D'_h(\mu_\sigma, \mu_\delta) = D_{\mathrm{W}} \cdot 1_f + i\gamma_5\mu_\sigma\tau^1 + \mu_\delta\tau^3\,, \tag{6}$$

which is related to D_h by $D'_h = (1 + i\tau^2)D_h(1 - i\tau^2)/2$ and $(\mu_\sigma, \mu_\delta) \to (\bar{\mu}, -\bar{\epsilon})$.

2 Installation and Usage

The software ships with a GNU autoconf environment and a configure script, which will generate GNU Makefiles to build the programmes. It is supported and recommended to configure and build the executables in a separate build directory. This also allows to have several builds with different options from the same source code directory.

2.1 Prerequisites

In order to compile the programmes the `LAPACK` [6] library (fortran version) needs to be installed. In addition it must be known which linker options are needed to link against `LAPACK`, e.g. `-Lpath-to-lapack -llapack -lblas`. Also a the latest version (tested is version 1.2.3) of `C-LIME` [7] must be available, which is used as a packaging scheme to read and write gauge configurations and propagators to files.

2.2 Configuring the hmc package

In order to get a simple configuration of the hmc package it is enough to just type

```
path-to-src-code/configure   --with-lime=<path-to-lime> \
     --with-lapack=<linker-flags> CC=<mycc> \
     F77=<myf77> CFLAGS=<c-compiler flags>
```

in the build directory. If `CC, F77` and `CFLGAS` are not specified, `configure` will guess them.

The code was successfully compiled and run at least on the following platforms: i686 and compatible, x64 and compatible, IBM Regatta systems, IBM Blue Gene/L, IBM Blue Gene/P, SGI Altix and SGI PC clusters, powerpc clusters.

The configure script accepts certain options to influence the building procedure. One can get an overview over all supported options with `configure --help`. There are `enable|disable` options switching on and off optional features and `with|without` switches usually related to optional packages. In the following we describe the most important of them (check `configure --help` for the defaults and more options):

- `--enable-mpi`:
  This option switches on the support for MPI. On certain platforms it automatically chooses the correct parallel compiler or searches for a command `mpicc` in the search path.
- `--enable-p4`:
  Enable the use of special Pentium4 instruction set and cache management.
- `--enable-opteron`:
  Enable the use of special opteron instruction set and cache management.
- `--enable-sse2`:
  Enable the use of SSE2 instruction set. This is a huge improvement on Pentium4 and equivalent systems.
- `--enable-sse3`:
  Enable the use of SSE3 instruction set. This will give another 20% of speedup when compared to only SSE2. However, only a few processors are capable of SSE3 so far.
- `--enable-gaugecopy`:
  See section 3.4 for details on this option. It will increase the memory requirement of the code.
- `--enable-halfspinor`:
  If this option is enabled the Dirac operator using half spinor fields is used. See sub-section 3.4 for details. If this feature is switched on, also the gauge copy feature is switched on automatically.
- `--with-mpidimension=n`:
  This option has only effect if the preceding one is switched on. The number of parallel directions can be specified. 1,2,3 and 4 dimensional parallelisation is supported.
- `--with-lapack="<linker flags>"`:
  the code requires lapack to be linked. All linker flags necessary to do so must be specified here. Note, that `LIBS="..."` works similar.
- `--with-limedir=<dir>`:
  Tells configure where to find the lime package, which is required for the build of the HMC. It is used for the ILDG file format.

The configure script will guess at the very beginning on which platform the build is done. In case this fails or a cross compilation must be performed please use the option `--host=HOST`. For instance in order to compile for the BG/P one needs to specify `--host=ppc-ibm-bprts --build=ppc64-ibm-linux`.

For certain architectures like the Blue Gene systems there are `README.arch` files in the top source directory with example configure calls.

2.3 Building and Installing

After successfully configuring the package the code can be build by simply typing `make` in the build directory. This will compile the standard executables. Typing `make install` will copy these executables into the install directory. The default install directory is `$HOME/bin`, which can be influenced e.g. with the `--prefix` option to `configure`.

2.4 Input parameter for main program

The main programs are called `hmc_tm` for the HMC algorithm and `invert` for even odd preconditioned inversion. They can be called with

- `-f filename`:
  where `filename` is the name of the input file to be used. The default name is `hmc.input` for `hmc_tm` and `invert.input` for `invert`.
- `-o name`:
  `name` will be used as name for several output files. This files differ by their suffix. Default for `name` is `output`.
- `-v` :
  makes the code a bit more verbose. Unrelated to input parameter `DebugLevel`.
- `-?|-h`:
  This will produce help output and exit then.

There are several input parameters read from an input file. The parser is contained in the file `gwc/src/bin/read_input.l`. The file `read_input.l` is converted to `read_input.c` using `flex` and defines the following function:

Definition:
```
int read_input(char * conf_file)
```

`conf_file` in string with input file name

The functions returns 0, if no error occurs, 2, if the input file could not be opened. If no input file could be opened or if there is no value given in the input file for a paramter, default values are used. All default values can be found in the file `gwc/src/bin/default_input_values.h`. The syntax is mostly `keyword = value` and `keyword` must be at the beginning of the line. Comments starting with `#` and empty lines are allowed. The order of the lines is not importand as long as every keyword appears only once. If it appears more than once, the last appearance becomes valid. The parser is case-insensitive.

In the following a list of the currently supported general input paramters:

1. `T`:
   The global time extension of the lattice. Default is 4.
2. `L`:
   The global spatial extension of the lattice. Default is 4.
3. `LX`:
   The global spatial x-extension of the lattice. Default is 4.
4. `LY`:
   The global spatial y-extension of the lattice. Default is 4.
5. `LZ`:
   The global spatial z-extension of the lattice. Default is 4.

6. `NrXProcs`:
   The number of processors in x-direction in case of two dimensional parallelisation. This has no effect in case of one dimensional parallelisation. In case of two dimensional parallelisation it must be properly set. The number of processors in time direction is automatically computed.

7. `NrYProcs, NrZProcs`:
   See `NrXProcs`.

8. `seed`:
   The seed for the random number generator. Default value is 123456.

9. `kappa`:
   The κ value. Default is 0.12. For the `hmc_tm` application, this must be set to the physical value! It can have different values in the single monomials, but here we need the target value.

10. `csw`:
    The value of the clover coefficient c_{sw}. Must be larger than zero to have effect. For the `hmc_tm` application, this must be set to the physical value! It can have different values in the single monomials, but here we need the target value. If set to larger than zero it will automatically trigger an additional monomial in the even/odd case for the trace log of the clover term. Default behaviour is no clover term.

11. `2KappaMu`:
    Twisted mass parameter (the physical one) for twisted mass action. This is for internal reasons $2\kappa\mu$. For the `hmc_tm` application, this must be set to the physical value! It can have different values in the single monomials, but here we need the target value.

12. `2KappaMuBar`:
    The average mass of the heavy doublet multiplied with 2κ. For the `hmc_tm` application, this must be set to the physical value! It can have different values in the single monomials, but here we need the target value.

13. `2KappaEpsBar`:
    The splitting mass multiplied with 2κ. For the `hmc_tm` application, this must be set to the physical value! It can have different values in the single monomials, but here we need the target value.

14. `Measurements`:
    Number of measurements in units of trajectories to be done. Default value is 3. For the `invert` programme this counts the number of gauge configurations to invert on. (See `Nsave` for the increment in the gauge index!)

15. `Nsave`:
    For `hmc_tm`: save every n-th trajectory the configuration to disk. For the `invert` programme it means that every n-th configuration is measured. This was formerly called `Nskip`.

    For `invert`: if more than one measurement is performed (see `Measurements` parameter), the gauge index is incremented by `Nsave` for each new measurement.

16. `InitialStoreCounter`:
    Start with value to label measurements. Default is 0. Can be also set to `readin` which causes to let the code check for a file `.nstore_counter` and reads the initial value from this file. If it is not existing, the counter will be set to 0.

17. `GaugeConfigInputFile`:
    Name of input file for the gauge field. Default is `conf`

18. `ThetaT|X|Y|Z=x`:
    This sets the boundary condition angle for the fermion fields in t, x, y or z direction to $\theta_t = \pi x$. Default value is zero. A value of 1 would mean antiperiodic boundary conditions for the fermion fields.

19. `DebugLevel`:
    If set to a value larger than 0 this causes verbose output:

    - `DebugLevel = 1`: forces, iteration counts and flops are printed out.
    - `DebugLevel = 2`: every iteration step is printed. Chronological Solver gives details about which routines are called, the same for the monomials. polynomial gets more verbose.
    - `DebugLevel > 2`: all available normal output.
    - `DebugLevel > 3`: all debug output. Involves extra computations, so the code will be (significantly) slower

20. `UseSloppyPrecision`:
    Use a reduced precision Dirac operator in the MC part of the HMC. Possible values are yes and no, the latter being the default. This could be possibly used in the invert code along the lines of `hep-lat/0609023` in the future.

21. `DisableIOChecks`:
    Defaults to no, if set to yes, this will disable several checks performed on gauge configuration input files, such size verification or SciDAC checksum matching. It will also disable the readback performed with Lemon IO.

22. `GaugeConfigRead|WritePrecision`:
    Read/Write gauge configurations in single (32) or double (64) precision. Default is 64.

23. `UseEvenOdd`:
    Whether or not to use even/odd preconditioning in the invert executable.

24. `OMPNumThreads`:
    Number of OpenMP threads to use per process when compiled with OpenMP support. On some architectures, the `OMP_NUM_THREADS` environment variable needs to be set to the same value for correct operation. The default is 1.

The following input parameters are `invert` specific:

1. `Indices=n-m`:
   Compute only components n to m of the quark propagator. n, m must be in $[0, 99]$. If the start index is not zero the data will be appended to the propagator file, unless `SplittedPropagator` is chosen. The program does not take care of the order, the data is just appended!

2. `UseRelativePrecision`:
   Possible values `yes, no`. Indicates whether relative precision is used in the inversions for the force and the acceptance computation. Default is no.

3. `GMRESMParameter`:
   Krylov subspace size m in GMRES(m) and such like iterative solvers. Not yet working!

4. `GMRESDRNrEv`:
   Number of eigenvalues to be deflated in GMRES-DR iterative solver. Not yet working!

5. `ReadSource`:
   If set to yes, then the source vector is read from a file.

6. `SourceTimeSlice`:
   The time slice of the source to be read. At the moment used only for the automatic construction of filenames. The filename will then be constructed as `basefilename.nstore.ts.ind`
   `SourceTimeSlice` can be also set to `detect` in order to let the code determine the appropriate timeslice value. (this might be slow, though, but it is unavoidable if `invert` should run more than one gauge in a single run and the timeslice value changes on a gauge basis.)

   It has only effect, if every source is in a separate file (i.e. SourceInfo.splitted is set, which is the default).

7. `SourceFilename` and `PropagatorFilename`:
   This sets the basefilename for sources and propagators respectively. The default is `source` for both.

8. `NoSamples`:
   in case of stochastic source the number of samples.

9. `SourceType`:
   lets you chose the source type: `Volume, Point, TimeSlice, PionTimeSlice, GenPionTimeSlice` are possible here.

10. `ComputeEVs`:
    compute eigenvalues and vectors before inversion in invert. Values can be no, yes and readin. In the latter case the eigenvalues and vectors are only read from disk, if possible. In case of yes it is also tried to read them from disk, but they are also recomputed, to a possibly higher precision.

11. `NoEigenvalues`:
    number of eigenvalues to compute.

12. `EigenvaluePrecision`:
    precision for eigenvalues.

13. `ComputeModeNumber`:
    compute the topological susceptibility using the spectral projectors method. Values can be yes or no.

14. `ComputeModeNumber`:
    compute the average number of eigenmodes of the massive hermitian operator $D_{tm}^{\dagger} D_{tm} + m^2$ with eigenvalues $\alpha \leq M^2$. The value can be yes and no.

15. `MStarSq`:
    value of the parameter M_*^2 necessary in order to compute the mode number or the topological susceptibility using the method of the spectral projectors.

16. `NoSourcesZ2`:
    number of Z2 stochastic sources for the spectral projectors method.

17. `SourceLocation`:
    integer indicating the location of the source. The location is computed as `SourceLocation = z+L*y+L*L*x+L*L*L*t`.

18. `UseStoutSmearing`:
    Whether or not to stout smear the configuration before inversion.

19. `StoutRho` and `StoutNoIterations`:
    Stout smearing parameter.

20. `WritePropagatorFormat` or `PropagatorType`:
    The type in which to store the propagator. There are

    - `DiracFermion_Sink`
    - `DiracFermion_Source_Sink_Pairs`
    - `DiracFermion_ScalarSource_TwelveSink`
    - `DiracFermion_ScalarSource_FourSink`

    available. However, only the first two are implemented so far.

21. `ComputeReweightingFactor`:
    If enabled reweighting factors will be computed corresponding to monomials that must be specified in the input file as well.

22. `NoReweightingSamples`:
    Number of random samples used per gauge configuration to estimate the reweighting factor. The default is 10.

The following input parameters are `hmc_tm` specific:

1. `ThermalisationSweeps`:
   As long as the number of trajectories is smaller than this number the acceptance test will be discarded. This might help to faster equilibrate the system.

2. `Startcondition`:
   The starting condition for a run. Possible values are `hot, cold, restart, continue`. Default is `cold`. Restart uses the seed to reset the random number generator. In case of `continue` the programme uses the file `.nstore_counter` to get the information about from where to read the gauge and the random number status. If this file does not exist (its written in the course of the HMC) then the input parameter described here are used instead.

3. `ReversibilityCheck`:
   If set to `yes` the program will perform a check of reversibility violation in the integrator by integrating back in time. If not yet existing, the program creates a file `return_check.data` in which it stores the reversibility violation as the difference in the Hamiltonian, the difference in the gauge fields and the relative difference in the Hamiltonian.

4. `ReversibilityCheckIntervall`:
   Here one can specify the intervall in terms of trajectories the program should check the reversibility violation.

Following the CHROMA notation we call every part in the action a monomial. A monomial is added to the action in the input file in the following way:

```
BeginMonomial TYPE
  Option = value
EndMonomial
```

`TYPE` can be one of the following

- `DET`: pseudo fermion representation of the (mass degenerate)

$$\det(Q^2(\kappa) + \mu^2)$$

- `CLOVERDET`: pseudo fermion representation of the (mass degenerate)

$$\det(Q^2_{\mathrm{sw}}(\kappa, c_{\mathrm{sw}}))$$

  for the clover operator without twisted mass term. This monomial is only available with even/odd preconditioning right now. It automatically adds another monomial for the Tr ln part of the clover term.

- `DETRATIO`: pseudo fermion representation of

$$\det(Q^2(\kappa) + \mu^2)/\det(Q^2(\kappa_2) + \mu_2^2)$$

- `GAUGE`:

$$\frac{\beta}{3}\sum_x \left(c_0 \sum_{\substack{\mu,\nu=1 \\ 1\le\mu<\nu}}^{4} \{1 - \mathrm{Re}\,\mathrm{Tr}(U^{1\times 1}_{x,\mu,\nu})\} + c_1 \sum_{\substack{\mu,\nu=1 \\ \mu\neq\nu}}^{4} \{1 - \mathrm{Re}\,\mathrm{Tr}(U^{1\times 2}_{x,\mu,\nu})\} \right),$$

- `NDPOLY`: polynomial representation of the (possibly non-degenerate) Wilson twisted mass doublet

$$[\det(\hat{Q}_h(\bar{\epsilon}, \bar{\mu})^2)]^{1/2} \approx \det(\mathcal{R}^{-1})$$

- `NDRAT`: rational representation of the (possibly non-degenerate) Wilson twisted mass doublet

$$[\det(\hat{Q}_h(\bar{\epsilon}, \bar{\mu})^2)]^{1/2}$$

  with an approximation

$$\mathcal{R}(Q^2_{nd}) \;=\; \prod_{i=1}^{N} \frac{Q^2_{nd} + a_{2i}}{\hat{Q}^2_h + a_{2i-1}} \approx \quad \frac{1}{\sqrt{\hat{Q}^2_h}}$$

- `NDRATCOR`: correction monomial for approximation errors in the rational approximation used in `NDRAT`
$$\det\left(\hat{Q}_h\mathcal{R}\right) .$$
- `NDCLOVERRAT, NDCLOVERRATCOR`: clover versions of `NDRAT` and `NDRATCOR`, respectively.
- `NDCLOVER`: polynomial representation of the (possibly non-degenerate) clover twisted mass doublet
$$[\det(Q_{nd}(\bar{\epsilon},\bar{\mu})^2), c_{\rm sw}]^{1/2}$$
- `POLY`: polynomial approximation ($P_n(x) \approx \frac{1}{x}$) of the mass degenerate determinant
$$\left[\det(P_n(Q^2(\kappa)+\mu^2))\right]^{-1}$$
- `POLYDETRATIO`: pseudo fermion representation of (for PHMC + mass precondition)
$$\left[\det(P_n(Q^2(\kappa)+\mu^2))det(Q^2(\kappa_2)+\mu_2^2)\right]^{-1}$$

Each of them has different options:

- `DET, CLOVERDET`:
  - `2KappaMu`
- `CLOVERDET`:
  - `csw`
- `DET, CLOVERDET`:
  - `Kappa`
  - `Timescale`: the timescale on which to integrate this monomial. Counting starts from zero up to the total number of timescales minus 1.
  - `CSGHistory`: the maximal number of vectors to store for the chronolical predictor (for CG and BiCGstab), default 0.
  - `CSGHistory2`: the maximal number of vectors to store for the second chronolical predictor (for BiCGstab only), default 0.
  - `ForcePrecision`: the solver precision used in the force computation
  - `AcceptancePrecision`: the solver precision used in the acceptance and heatbath
  - `MaxSolverIterations`: default is 5000
  - `Solver`: the solver to be used, either CG or BiCGstab. Default is CG.
  - `Name`: a name to be assigned to the monomial. The default is `DET`
- `DETRATIO`: the same as for `DET`, but in addition:
  - `2KappaMu2`
  - `Kappa2`

  - `Name`: a name to be assigned to the monomial. The default is `DETRATIO`

- `GAUGE`:
  - `Timescale`: the timescale on which to integrate this monomial. Counting starts from zero up to the total number of timescales minus 1.
  - `Name`: a name to be assigned to the monomial. The default is `GAUGE`.
  - `beta`:
    The invers coupling β. Default value is 5.2.
  - `Type`: can be one of `Wilson, tlsym, Iwasaki, DBW2, user`. For type user you can specify also the two following options. Default is `user` here.
  - `UseRectangleStaples`: can be yes or no, indicating whether to use also the rectangle staples. No corresponds to pure Wilson plaquette. Default is no. Is effective only for `type = user`.
  - `RectangleCoefficient`: the value of the parameter c_1. The coefficient c_0 is computed from $c_0 = 1 - 8c_1$. Is effective only for `type = user`.

  There is maximally one instance allowed of this type.

- `NDPOLY`: switches on the PHMC part for the non-degenerate heavy doublet and lets you specify the timescale on which to integrate this and the parameters.
  - `2KappaMubar`: $2\kappa\bar{\mu}$ the heavy twisted mass
  - `2KappaEpsbar`: $2\kappa\bar{\epsilon}$ the heavy splitting
  - `Kappa`: the κ value
  - `Timescale`: the timescale on which to integrate this monomial. Counting starts from zero up to the total number of timescales minus 1.
  - `Name`: a name to be assigned to the monomial. The default is `NDPOLY`
  - `ComputeEVFreq`: If you want to calculate the eigenvalues every n'th trajectory then set this parameter to n if you want no eigenvalues set this to 0 during thermalization you should set this to 1 or 2 to follow the evolution of smallest and largest eigenvalue to adjust the approximation interval of the polynomial
  - `ComputeOnlyEVs`: Computes only once at the very beginning of the run the eigenvalues of the heavy split operator and exits.
  - `StildeMin`: lower bound for the approximation interval of the polynomial
  - `StildeMax`: upper bound for the approximation interval of the polynomial
  - `DegreeOfMDPolynomial`: degree of the less precise polynomial P. Must be identical to the degree used to compute the roots.
  - `LocNormConst`: Constant (local normalisation constant) which is multiplied to each monomial (of the polynomial P_n).
  - `RootsFile`: File name specifying a file containing the $n =$ `Degree` roots of the Polynomial
  - `PrecisionPtilde`: Precision of the more precise polynomial $\tilde{P}$ used in the heat-bath and the acceptance step of the PHMC.
  - `PrecisionHfinal`:

So far, there is maximally one instance allowed for this type. This might change in the future.

- `NDRAT`: like `NDPOLY`, but with a rational approximation.
  - `2KappaMubar`: $2\kappa\bar{\mu}$ the heavy twisted mass
  - `2KappaEpsbar`: $2\kappa\bar{\epsilon}$ the heavy splitting
  - `Kappa`: the κ value
  - `DegreeOfRational`: the order N of the rational approximation
  - `StildeMin`: lower bound for the approximation interval of the rational approximation
  - `StildeMax`: upper bound for the approximation interval of the rational approximation
  - `Cmin`: it is possible to use only pairs of coefficients in the range from $[c_a, c_b]$ in order to introduce an frequency splitting. `Cmin` corresponds to $0 < c_a < N$, where N is the order of the rational approximation. The ordering of the partial fractions in the rational approximation is such that

    $$\mu_0 > \mu_1 > ... > \mu_{N-1}\,,$$

    and hence $c_a = N - 1$ and $c_b = N - 1$ would generate a rational with only the smallest and, therefore, most expensive shift (which one would typically integrate on a coarse timescale). $c_a = 0$ and $c_b = k < N$ would correspond to a rational with the $k + 1$ largest shifts.
  - `Cmax`: $c_b \geq c_a$, see `Cmin`.
  - `ComputeOnlyEVs`: Computes only once at the very beginning of the run the eigenvalues of the heavy split operator and exits.
  - `ForcePrecision`: the CGMMS solver precision used in the force computation
  - `AcceptancePrecision`: the CGMMS solver precision used in the acceptance and heatbath
  - `MaxSolverIterations`: maximal number of CGMMS solver iterations, default is 5000.

  It is important to realise that if the splitting is used, then every partial fraction *must appear once and only once*. Otherwise, the algorithm will not describe the desired physics! Consequently, also the different `NDRAT` monomials from the same rational approximation used for frequency splitting have to have identical order.

- `NDRATCOR`: correction monomial for approximation errors in the rational approximation for the heavy doublet. This monomial has no derivative part and it is only used in the heatbath and acceptance steps.
  - `2KappaMubar`: $2\kappa\bar{\mu}$ the heavy twisted mass
  - `2KappaEpsbar`: $2\kappa\bar{\epsilon}$ the heavy splitting
  - `Kappa`: the κ value
  - `DegreeOfRational`: the order N of the rational approximation. *The order must match the order of the corresponding (splitted) `NDRAT` monomial(s).*

- `StildeMin`: lower bound for the approximation interval of the rational approximation
- `StildeMax`: upper bound for the approximation interval of the rational approximation
- `ComputeOnlyEVs`: Computes only once at the very beginning of the run the eigenvalues of the heavy split operator and exits.
- `ForcePrecision`: the CGMMS solver precision used in the force computation
- `AcceptancePrecision`: the CGMMS solver precision used in the acceptance and heatbath
- `MaxSolverIterations`: maximal number of CGMMS solver iterations, default is 5000.

- `NDCLOVERRAT, NDCLOVERRATCOR`: The same as `NDRAT, NDRATCOR`, but with the additional parameter `CSW` and only for `NDCLOVERRAT`
  - `AddTrLog =yes|no`: adds a clover trlog monomial with the parameters of this monomial. `no` is default. One needs only one trlog monomial per non-degenerate doublet, so one needs to take care in case of frequency splitting of the rational approximation to have this set to `yes` only once.

- `POLY, POLYDETRATIO`:
  - `Degree`: Degree of the Polynomial.
  - `Lmin`: Lower bound of approximation interval.
  - `Lmax`: Upper bound of approximation interval.
  - `LocNormConst`: Constant (local normalisation constant) which is multiplied to each monomial (of the polynomial P_n).
  - `RootsFile`: File name specifying a file containing the $n =$ `Degree` roots of the Polynomial
  - `+ Parameters from DET & DETRATIO monomial`

  There can be arbitrary many POLY monomials. But take into account that there will be allocated $n/2$ number of spinor fields for EACH poly monomial. (Maybe in the future we should think about to share these fields with all POLY/NDPOLY monomials as there are used only for the computation of the force and have to be updated before each successive calculation of the force.)
  This monomial needs a valid `RootsFile` and `LocNormConst` parameter. Both can be obtained from the `oox` program in the `util/oox` subdirectory of the hmc code. It can be invoked by the command:
  `$ oox -d <degree> -e <epsilon>`
  `<epsilon>` is to be replaced by the ratio `Lmin/Lmax`.

2.4.1 The Integrator

The Integrator can be specified similar to the monomials:

```
BeginIntegrator
  Option = value
EndIntegrator
```

with the following options available:

- `Tau`: total trajectory length.
- `NumberOfTimescales`: total number of timescales.
- `MonitorForces`: setting this to `yes` enables the computation of the forces per monomial at the beginning of each trajectory.
- `IntegrationStepsN = M` where `N` is the timescale (as integer value, counting starts from zero and goes up to the number of timescales minus 1) and `M` is the number of integration steps on that timescale. Note, that the integrators or defined recursively.
- `LambdaN = F` where `N` is the timescale and `F` is a floating point number specifying the λ value to be used on this timescale in case of the second order minimal norm integrator (2MN, 2MNPOSITION). The default value is 0.19. Note, that $\lambda = 1/6$ is the Sexton-Weingarte scheme.
- `TypeN = TYPE`: set the type of integrator to be used on timescale `N`. The following types available: `2MN, 2MNPOSITION, LEAPFROG`

  The position versions are not compatible with the velocity versions, thus they must not be used together.

A timescale must not be empty. Currently the maximal number of timescales is 10 and there cannot be more than 10 monomials per timescale. But there can be more than one monomial per timescale.

2.4.2 Chosing the Operator for Inversions

```
BeginOperator TYPE
  Option = value
EndOperator
```

`TYPE` can be one of the following

- `WILSON`: simple Wilson Dirac operator, with options:
  - `UseEvenOdd`
- `TMWILSON`: Wilson Twisted Mass Dirac operator, with options:
  - `2KappaMu`
  - `UseEvenOdd`
- `CLOVER`: Clover Twisted Mass Dirac operator, with options:
  - `2KappaMu`
  - `UseEvenOdd`

  - CSW

- DBTMWILSON: two flavour mass non-degenerate Wilson Twisted Mass Dirac operator:
  - 2KappaMubar
  - 2KappaEpsbar
- DBCLOVER: two flavour mass non-degenerate Clover Twisted Mass Dirac operator:
  - CSW
  - 2KappaMubar
  - 2KappaEpsbar
- OVERLAP: overlap operator:
  - m
  - s
  - DegreeOfPolynomial
  - NoKernerlEigenvalues
  - KernelEigenvaluePrecision

All of them provide the following options available:

- kappa:
- Solver:
  Sets the solver to be used. Possible values are among others CG, BiCGstab, CGS, GMRES, PCG and CGMMS.
- MaxSolverIterations:
- PropagatorPrecision:
- SolverPrecision:

The CGMMS solver can be used to invert the operator for multiple masses at the same time. To this end a list of masses needs to be provided either as a comma-separated list or as the filename of a text file which lists one mass per line. The masses must be provided in the format $2\kappa\mu_n$. The normal mass specified for the operator is used as μ_0. The masses must be ordered such that $\mu_0 < \mu_1 < ... < \mu_n$.:

- ExtraMasses = 0.12, 0.14, 0.17, 0.21, 0.30
- ExtraMasses = extra_masses.input

2.4.3 Online Measurements

A number of measurements can be performed online while the hmc is running.

```
BeginMeasurement TYPE
  Option = value
EndMeasurement
```

where `TYPE` can be currently one of the following:

- `CORRELATORS`:
  - `MaxSolverIterations`

  this is for zero temperature, so the stochastic source is at fixed t. In addition it needs an operator defined in the input file, otherwise it will do nothing. (see input keywords for `invert` above)

- `PIONNORM`:
  - `MaxSolverIterations`

  this is for finite temperature, the stochastic source is at fixed z.

- `POLYAKOVLOOP`:
  - `Directions` can be either 0 for time- or 3 for z-direction.

The frequency of measuring all of these can be adjusted with the Option `Frequency`.

2.4.4 Example Input File

The following is a typical HMC input file:

```
L=8
T=16
Measurements = 1
Startcondition = hot
2KappaMu = 0.03
kappa = 0.090
2KappaMubar = 1.
2KappaEpsbar = 0.2

#This is a comment

PhmcRecEVInterval = 1
Nsave = 50
ThetaT = 1.
InitialStoreCounter = readin
UseEvenOdd = yes
ReversibilityCheck = no
ReversibilityCheckIntervall = 1
```

```
DebugLevel = 3

BeginMeasurement CORRELATORS
  MaxSolverIterations = 1000
  Frequency = 1
EndMeasurement

BeginMonomial GAUGE
  beta = 3.30
  Timescale = 0
  Type = tlsym
EndMonomial

BeginMonomial DET
  Timescale = 1
  2KappaMu = 0.
  kappa = 0.125
  AcceptancePrecision =  1.e-20
  ForcePrecision = 1.e-12
  Name = det
  solver   = cg
  CSGHistory = 10
  CSGHistory2 = 10
EndMonomial

BeginMonomial DETRATIO
  Timescale = 2
  2KappaMu = 0.03
  2KappaMu2 = 0.1
  kappa = 0.125
  kappa2 = 0.125
  maxiter = 20000
  AcceptancePrecision =  1.e-20
  ForcePrecision = 1.e-12
  Name = detrat
  solver = cg
EndMonomial

# this is a NDPOLY monomial
# but commented out
#BeginMonomial NDPOLY
#  Timescale = 1
#EndMonomial

BeginIntegrator
  Type0 = 2MN
  Type1 = 2MN
  Type2 = 2MN
  IntegrationSteps0 = 1
```

```
  IntegrationSteps1 = 2
  IntegrationSteps2 = 3
  tau = 1.
  Lambda0 = 0.19
  NumberOfTimescales = 3
EndIntegrator

# for the CORRELATORS online measurement
BeginOperator TMWILSON
  2kappaMu = 0.177
  kappa = 0.177
  UseEvenOdd = yes
  Solver = CG
  SolverPrecision = 1e-14
  MaxSolverIterations = 1000
EndOperator
```

There are realistic small volume sample input files in the sub-directory `sample-input`, which also represent test runs for the code. For the inverter a typical file would look like

```
L=4
T=4
DebugLevel = 2
InitialStoreCounter = 1
Indices = 0-7
ReadSource = no
Measurements = 1
ThetaT = 1.
UseEvenOdd = no
UseRelativePrecision = yes
SplittedPropagator = yes
PropagatorType = DiracFermion_Source_Sink_Pairs
UseStoutSmearing = no
StoutRho = 0.15
StoutNoIterations = 10
UseSloppyPrecision = yes

# both operators will be inverted for
BeginOperator TMWILSON
  Solver = CG
  2KappaMu = 0.177
  kappa = 0.177
  SolverPrecision = 1.e-15
  UseEvenOdd = yes
EndOperator

BeginOperator DBTMWILSON
  2KappaMubar = 0.177
  2KappaEpsbar = 0.190
```

```
  kappa = 0.177
EndOperator


# and for reweighting possibly
BeginMonomial DETRATIO
  Timescale = 2
  2KappaMu = 0.03
  2KappaMu2 = 0.0305
  kappa = 0.15
  kappa2 = 0.15
  maxiter = 20000
  AcceptancePrecision =  1.e-20
  Name = detrat
  solver = cg
EndMonomial
```

2.4.5 Reread functionality

If you store a file with name `hmc.reread` in the working directory of a running HMC, the program will read in this file after the next finished trajectory. Then it will change the parameters accordingly without the need of restarting the program.

One cannot change from gauge action without rectangle part to gauge action with rectangle part. If one wants to change μ-, ϵ^2- or N_i-parameter one has to give allways all of them. Otherwise the internal matching does not work and the program will do nonsense.

The file will be deleted automatically, if it was used. A message will be posted to standard output and to the file `history_hmc_tm` to let you identify the exact point where the parameters changed.

2.5 Output files

`output.data`

The file `output.data` contains lines for each performed trajectory. Each line has entries with the following meaning:

1. Plaquette value.
2. ΔH
3. $\exp(-\Delta H)$
4. number of pseudo fermion monomials times two integers. The first is the number of CG or BiCGstab solveriterations used in the acceptance and heatbath steps, the second is the number of CG (BiCGstab) iterations used for the force computation.
5. Acceptance (0 is rejected, 1 is accepted).

6. Time in seconds needed for this trajectory. In case of non MPI this is zero, because not measured.
7. Value of the rectangle part in the gauge action, if used.

Every new run will append its numbers to an already existing file.

`output.para`

This file contains the parameters used in this run. Old files will be overwritten.

`history_hmc_tm`

This file provides a mapping between the configuration number and its plaquette and Poliakov loop values. Moreover the simulation parameters are stored there and in case of a reread the time point can be found there.

`return_check.data`

Contains the reversibility violation measurements, if they are performed.

`conf.save`

This file is written after each trajectory, if no regular configuration is saved. It contains the most recent gauge configuration and the status of the random number generator for a restart of the programme.

`onlinemeas.N`

Contains the online measurement for trajectory `N` if this feature is switched on.

2.6 Programme `gen_sources`

The programme `gen_sources` provides an interface to generate stochastic sources for several different situations. It is able to generate those for the nucleon case (which should not be used, because point sources are optimal), for mesons in general and for the special case of the pion only.

The programme offers command line options as follows:

- `-h|?` a help.
- `-L` the spatical lattice size
- `-T` the temporal lattice size
- `-o` the base filename of the sources (default is `source`)
- `-n` the configuration number (default is 0)
- `-s` the sample number (default is 0)

- `-t` the value of the start timeslice (default 0)
- `-S` the spatial spacing/dilution (default 1)
- `-P` the temporal spacing/dilution (default T)
- `-N` produce nucleon sources (default meson sources)
- `-p` plain output filename (see below)
- `-O` the special pion only case
- `-E` extended sources for pion three point functions. Together with `-O`
- `-d` write source in double precision (default single)
- `-a` write all sources in one file rather than 12 (pion only is one file anyhow)

The output filename is generated like `base.sampleno.gaugeno.tsno.00 -11`, unless `-p` is chosen, which would correspond to `base.00-11`.

The special pion only case corresponds to a single timeslice source without any dilution in spin or colour or space.

3 Implementation

The general strategy of the tmLQCD package is to provide programs for the main applications used in lattice QCD with Wilson twisted mass fermions. The code and the algorithms are designed to be general enough such as to compile and run efficiently on any modern computer architecture. This is achieved code-wise by using standard C as programming language and for parallelisation the message passing interface (MPI) standard version 1.1.

Performance improvements are achieved by providing dedicated code for certain widely used architectures, like PC's or the Blue Gene family. Dedicated code is mainly available for the kernel routine – the application of the Dirac operator, which will be discussed in detail in section 3.4, and for the communication routines.

The tmLQCD package provides three main applications. The first is an implementation of the (P)HMC algorithm, the second and the third are executables to invert the Wilson twisted mass Dirac operator (4) and the non-degenerate Wilson twisted mass Dirac operator (5), respectively. All three do have a wide range of run-time options, which can be influenced using an input file. The syntax of the input file is explained in the documentation which ships with the source code. The relevant input parameters will be mentioned in the following where appropriate, to ease usage.

We shall firstly discuss the general layout of the three aforementioned applications, followed by a general discussion of the parallelisation strategy used in all three of them.

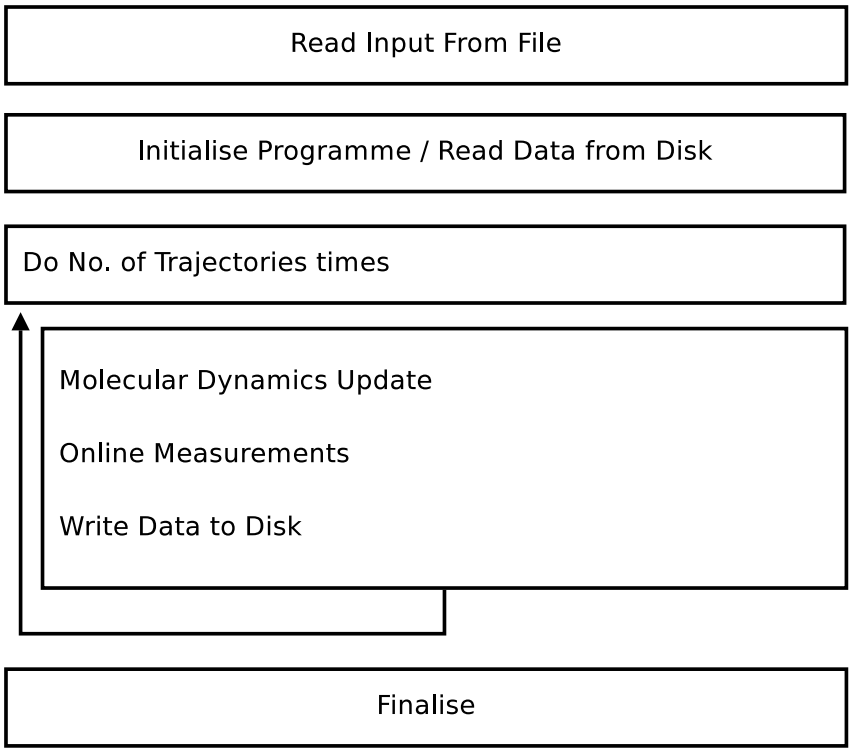

Figure 1: Flowchart for the `hmc_tm` executable

3.1 `hmc_tm`

In figure 1 the programme flow of the `hmc_tm` executable is depicted. In the first block the input file is parsed and parameters are set accordingly. Then the required memory is allocated and, depending on the input parameters, data is read from disk in order to continue a previous run.

The main part of this application is the molecular dynamics update. For a number of trajectories, which must be specified in the input file, first a heat-bath is performed, then the integration according to the equations of motion using the integrator as specified in the input file, and finally the acceptance step.

After each trajectory certain online measurements are performed, such as measuring the plaquette value. Other online measurements are optional, like measuring the pseudo scalar correlation function.

3.1.1 command line arguments

The programme offers command line options as follows:

- `-h|?` prints a help message and exits.
- `-f` input file name. The default is `hmc.input`
- `-o` the prefix of the output filenames. The default is `output`. The code will generate or append to two files, `output.data` and `output.para`.

3.1.2 Input / Output

The parameters of each run are read from an input file with default name `hmc.input`. If it is missing all parameters will be set to their default values. Any parameter not set in the input file will also be set to its default value.

During the run the `hmc_tm` program will generate two output files, one called per default `output.data`, the other one `output.para`. Into the latter important parameters will be written at the beginning of the run.

The file `output.data` has several columns with the following meanings

1. Plaquette value.
2. ΔH
3. $\exp(-\Delta H)$
4. number of pseudo fermion monomials times two integers. The first of the two is the sum of solver iterations needed in the acceptance and heatbath steps, the second is the sum of iterations needed for the force computation of the whole trajectory.
5. Acceptance (0 or 1).
6. Time in seconds needed for this trajectory.
7. Value of the rectangle part in the gauge action, if used.

Every new run will append its numbers to an already existing file.

In addition, the program will create a file `history_hmc_tm`. This file provides a mapping between the configuration number and its plaquette and Polyakov loop values. Moreover the simulation parameters are stored there and in case of a reread the time point can be found there.

After every trajectory the program will save the current configuration in the file `conf.save`.

3.2 `invert` and `invert_doublet`

The two applications `invert` and `invert_doublet` are very similar. The main difference is that in `invert` the one flavour Wilson twisted mass Dirac operator is inverted, whereas in `invert_doublet` the non-degenerate doublet is inverted.

The main part of the two executables is depicted in figure 2. Each measurement corresponds to one gauge configuration that is read from disk into memory. For each of these gauge configurations a number of inversions will be performed.

The sources can be either generated or read in from disk. In the former case the programme can currently generate point sources at random location in space time. In the latter case the name of the source file can be specified in the input file.

The relevant Dirac operator is then inverted on each source and the result is stored on disk. The inversion can be performed with a number of inversion algorithms, such as conjugate gradient (CG), BiCGstab, and others [8]. And optionally even/odd preconditioning as described previously can be used.

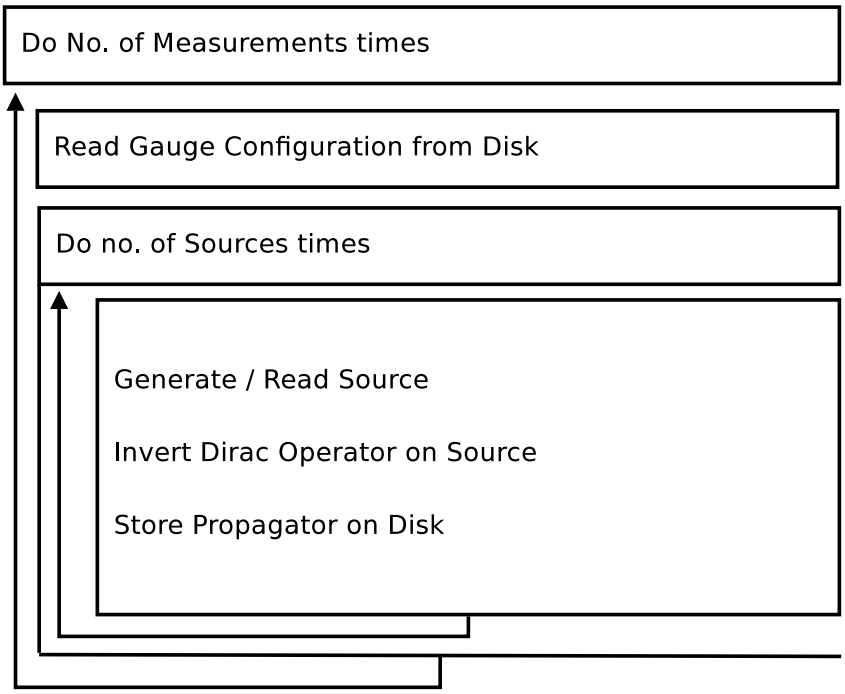

Figure 2: Flowchart for the main part of the `invert` and `invert_doublet` executables.

3.2.1 command line arguments

The two programmes offer command line options as follows:

- `-h|?` prints a help message and exits.
- `-f` input file name. The default is `hmc.input`
- `-o` the prefix of the output filenames. The default is `output`. The code will generate or append to one file called `output.para`.

3.2.2 Output

The program will create a file called `output.data` with information about the parameters of the run. Of course, also the propagators are stored on disc. The corresponding file names can be influenced via input parameters. The file format is discussed in some detail in sub-section 4.

One particularity of the `invert_doublet` program is that the propagators written to disk correspond to the two flavour Dirac operator of eq. (6), i.e.

$$D'_h(\mu_\sigma, \mu_\delta) = D_{\mathrm{W}} \cdot 1_f + i\mu_\sigma \tau^1 + \gamma_5 \mu_\delta \tau^3 \,,$$

essentially for compatibility reasons. For the two flavour components written the first is the would be *strange* component and the second one the would be *charm* one.

3.3 Parallelisation

The whole lattice can be parallelised in up to 4 space-time directions. It is controlled with configure switches, see section 2.2. The Message Passing Interface (MPI, standard version

1.1) is used to implement the parallelisation. So for compiling the parallel executables a working MPI implementation is needed.

Depending on the number of parallelised space-time directions the t-direction, the t- and x-direction, the t-, x- and y-direction or the t-, x- and y- and z-direction are parallelised.

The number of processors per space direction must be specified at run time, i.e. in the input file. The relevant parameters are `NrXProcs`, `NrYProcs` and `NrZProcs`. The number of processors in time direction is determined by the program automatically. Note that the extension in any direction must divide by the number of processors in this direction.

In case of even/odd preconditioning further constraints have to be fulfilled: the local number of lattice sites must be even and the local L_z must be even. Moreover, the local product $L_t \times L_x \times L_y$ must be even in case of even/odd preconditioning.

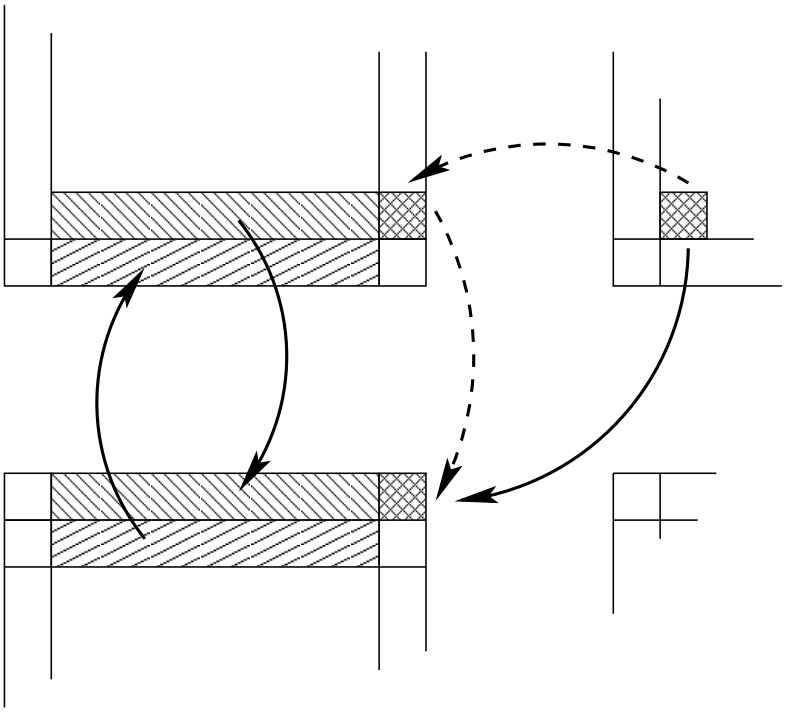

Figure 3: Boundary exchange in a two dimensional parallel setup. One can see that the internal boundary is send while the external one is received. The corners need a two step procedure.

The communication is organised using boundary buffer, as sketched in figure 3. The MPI setup is contained in the file `mpi_init.c`. The corresponding function must be called at the beginning of a main program just after the parameters are read in, also in case of a serial run. In this function also the various `MPI_Datatypes` are constructed needed for the exchange of the boundary fields. The routines performing the communication for the various data types are located in files starting with `xchange_`.

The communication is implemented using different types of MPI functions. One implementation uses the `MPI_Sendrecv` function to communicate the data. A second one uses non-blocking MPI functions and a third one persistent MPI calls. See the MPI standard for details [9]. On machines with network capable of sending in several directions in parallel the non-blocking version is the most efficient one. The relevant configure

switches are `--with-nonblockingmpi` and `--with-persistentmpi`, the latter of which is only available for the Dirac operator with halfspinor fields, see section 3.4.

3.4 Dirac Operator

The Dirac operator is the kernel routine of any lattice QCD application, because its inverse is needed for the HMC update procedure and also for computing correlation functions. The inversion is usually performed by means of iterative solvers, like the conjugate gradient algorithm, and hence the repeated application of the Dirac operator to a spinor field is needed. Thus the optimisation of this routine deserves special attention.

At some space-time point x the application of a Wilson type Dirac operator is mainly given by

$$\begin{aligned}\phi(x) =&(m_0 + 4r + i\mu_q\gamma_5)\psi(x) \\ &- \frac{1}{2}\sum_{\mu=1}^{4}\Big[U_{x,\mu}(r+\gamma_\mu)\psi(x+a\hat{\mu}) + U^{\dagger}_{x-a\hat{\mu},\mu}(r-\gamma_\mu)\psi(x-a\hat{\mu})\Big]\end{aligned} \tag{7}$$

where r is the Wilson parameter, which we set to one in the following. The most computer time consuming part is the next-neighbour interaction part.

For this part it is useful to observe that

$$(1 \pm \gamma_\mu)\psi$$

has only two independent spinor components, the other two follow trivially. So only two of the components need to be computed, then to be multiplied with the corresponding gauge field U, and then the other two components are to be reconstructed.

The operation in eq. (7) must be performed for each space-time point x. If the loop over x is performed such that all elements of ϕ are accessed sequentially (one output stream), it is clear that the elements in ψ and U cannot be accessed sequentially as well. This non-sequential access may lead to serious performance degradations due to too many cache misses, because modern processing units have only a very limited number of input streams available.

While the ψ field is usually different from one to the next application of the Dirac operator, the gauge field stays often the same for a large number of applications. This is for instance so in iterative solvers, where the Dirac operator is applied $\mathcal{O}(1000)$ times with fixed gauge fields. Therefore it is useful to construct a double copy of the original gauge field sorted such that the elements are accessed exactly in the order needed in the Dirac operator. For the price of additional memory, with this simple change one can obtain large performance improvements, depending on the architecture. The double copy must be updated whenever the gauge field change. This feature is available in the code at configure time, the relevant switch is `--with-gaugecopy`.

Above we were assuming that we run sequentially through the resulting spinor field ϕ. Another possibility is to run sequentially through the source spinor field ψ. Moreover, one could split up the operation (7) as follows, introducing intermediate result vectors $\varphi^{\pm}$ with only two spinor components per lattice site[1]. Concentrating on the hopping part

[1] We thank Peter Boyle for useful discussions on this point.

only, we would have

$$\begin{aligned}\varphi^+(x,\mu) &= P^{4\to 2}_{+\mu}\ U_{x,\mu}(r+\gamma_\mu)\psi(x)\\ \varphi^-(x,\mu) &= P^{4\to 2}_{-\mu}\ (r-\gamma_\mu)\psi(x)\ .\end{aligned} \tag{8}$$

From $\varphi^\pm$ we can then reconstruct the resulting spinor field as

$$\begin{aligned}\phi(x) =& \sum_\mu P^{2\to 4}_{+\mu}\varphi^+(x+a\hat{\mu},\mu)\\ &+\sum_\mu P^{2\to 4}_{-\mu}U^\dagger_{x-a\hat{\mu},\mu}\varphi^-(x-a\hat{\mu},\mu)\end{aligned} \tag{9}$$

Here we denote with $P^{4\to 2}_{\pm\mu}$ the projection to the two independent spinor components for $1\pm\gamma_\mu$ and with $P^{2\to 4}_{\pm\mu}$ the corresponding reconstruction. The half spinor fields $\varphi^\pm$ can be interlaced in memory such that $\psi(x)$ as well as $\varphi^\pm(x)$ are always accessed sequentially in memory. The same is possible for the gauge fields, as explained above. So only for ϕ we cannot avoid strided access. So far we have only introduced extra fields $\varphi^\pm$, which need to be loaded and stored from and to main memory, and divided the Dirac operator into two steps (8) and (9) which are very balanced with regard to memory bandwidth and floating point operations.

The advantage of this implementation of the Dirac operator comes in the parallel case. In step (8) we need only elements of $\psi(x)$, which are locally available on each node. So this step can be performed without any communication. In between step (8) and (9) one then needs to communicate part of $\varphi^\pm$, however only half the amount is needed compared to a communication of ψ. After the second step there is then no further communication needed. Hence, one can reduce the amount of data to be send by a factor of two.

There is yet another performance improvement possible with this form of the Dirac operator, this time for the price of precision. One can store the intermediate fields $\varphi^\pm$ with reduced precision, e.g. in single precision when the regular spinor fields are in double precision. This will lead to a result with reduced precision, however, in a situation where this is not important, as for instance in the MD update procedure, it reduces the data to be communicated by another factor of two. And the required memory bandwidth is reduced as well. This version of the hopping matrix (currently it is only implemented for the hopping matrix) is available at configure time with the switch `--enable-halfspinor`.

The reduced precision version (sloppy precision) is available through the input parameter `UseSloppyPrecision`. It will be used in the MD update where appropriate. Moreover, it is implemented in the CG iterative solver following the ideas outlined in Ref. [10] for the overlap operator.

The various implementations of the Dirac operator can be found in the file `D_psi.c` and – as needed for even/odd preconditioning – the hopping matrix in the file `Hopping_Matrix.c`. There are many different versions of these two routines available, each optimised for a particular architecture, e.g. for the Blue Gene/P double hummer processor or the streaming SIMD extensions of modern PC processors (SSE2 and SSE3), see also Ref. [11]. Martin Lüscher has made available his standard C and SSE/SSE2 Dirac operator [12] under the GNU General Public License, which are partly included into the tmLQCD package.

3.4.1 Blue Gene Version

The IBM PowerPC 450d processor used on the Blue Gene architecture provides a dual FPU, which supports a set of SIMD operations working on 32 special registers useful for

Algorithm 1 $\varphi^+ = \kappa U P_{+0}^{4\to 2}(1+\gamma_0)\psi$

```
1: // load components of ψ into registers
2: _bgl_load_rs0((*s).s0);
3: _bgl_load_rs1((*s).s1);
4: _bgl_load_rs2((*s).s2);
5: _bgl_load_rs3((*s).s3);
6: // prefetch gauge field for next direction (1 + γ1)
7: _prefetch_su3(U+1);
8: // do now first P_{+0}^{4→2}(1 + γ0)ψ
9: _bgl_vector_add_rs2_to_rs0_reg0();
10: _bgl_vector_add_rs3_to_rs1_reg1();
11: //now multiply both components at once with gauge field U and κ
12: _bgl_su3_multiply_double((*U));
13: _bgl_vector_cmplx_mul_double(ka0);
14: // store the result
15: _bgl_store_reg0_up((*phi[ix]).s0);
16: _bgl_store_reg1_up((*phi[ix]).s1);
```

lattice QCD. These operations can be accessed using build in functions of the IBM XLC compiler. The file `bgl.h` contains all macros relevant for the Blue Gene version of the hopping matrix and the Dirac operator.

A small fraction of half spinor version (see above) is given in algorithm 1, which represents the operation $\varphi^+ = \kappa U P_{+0}^{4\to 2}(1+\gamma_0)\psi$. After loading the components of ψ into the special registers and prefetching the gauge field for the next direction (in this case $1+\gamma_1$), $P_{+0}^{4\to 2}(1+\gamma_0)\psi$ is performed. It is then important to load the gauge field U only once from memory to registers and multiply both spinor components in parallel.

Finally the result is multiplied with κ (which inherits also a phase factor due to the way we implement the boundary conditions, see next sub-section) and stored in memory.

3.4.2 Boundary Conditions

As discussed previously, we allow for arbitrary phase factors in the boundary conditions of the fermion fields. This is conveniently implemented in the Dirac operator as a phase factor in the hopping term

$$\sum_{\mu}\left[e^{i\theta_\mu \pi/L_\mu}\ U_{x,\mu}(r+\gamma_\mu)\psi(x+a\hat{\mu}) + e^{-i\theta_\mu \pi/L_\mu}\ U^{\dagger}_{x-a\hat{\mu},\mu}(r-\gamma_\mu)\psi(x-a\hat{\mu})\right].$$

The relevant input parameters are `ThetaT`, `ThetaX`, `ThetaY`, `ThetaZ`.

3.5 The HMC Update

We assume in the following that the action to be simulated can be written as

$$S = S_{\mathrm{G}} + \sum_{i=1}^{N_{\mathrm{monomials}}} S_{\mathrm{PF}_i},$$

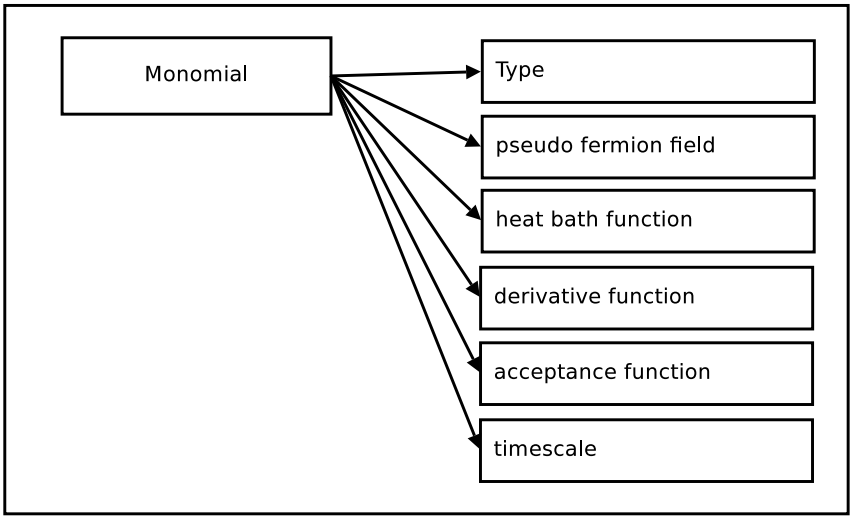

Figure 4: Data type monomial and its components

and we call – following the CHROMA notation [13] – each term in this sum a *monomial*. We require that there is exactly one gauge monomial S_{G} (which we identify with S_0 in the following) and an arbitrary number of pseudo fermion monomials S_{PF_i}.

As a data type every monomial must known how to compute its contribution to the initial Hamiltonian $\mathcal{H}$ at the beginning of each trajectory in the heat-bath step. Then it must know how to compute the derivative with respect to the gauge fields for given gauge field and pseudo fermion field needed for the MD update. And finally there must be a function to compute its contribution to the final Hamiltonian $\mathcal{H}'$ as used in the acceptance step.

In addition for each monomial it needs to be known on which timescale it should be integrated. The corresponding data type is sketched in figure 4. The general definitions for this data type can be found in the file `monomial.c`.

There are several sorts of monomials implemented:

- `DET`: pseudo fermion representation of the (mass degenerate) simple determinant

$$\det(Q^2(\kappa) + \mu^2)$$

- `DETRATIO`: pseudo fermion representation of the determinant ratio

$$\det(Q^2(\kappa) + \mu^2)/\det(Q^2(\kappa_2) + \mu_2^2)$$

- `NDPOLY`: polynomial representation of the (possibly non-degenerate) doublet

$$[\det(Q_{nd}(\bar{\epsilon}, \bar{\mu})^2)]^{1/2}\,.$$

- `GAUGE`:

$$\frac{\beta}{3}\sum_x \left(c_0 \sum_{\substack{\mu,\nu=1 \\ 1\le\mu<\nu}}^{4} \{1 - \mathrm{Re}\,\mathrm{Tr}(U^{1\times 1}_{x,\mu,\nu})\} + c_1 \sum_{\substack{\mu,\nu=1 \\ \mu\neq\nu}}^{4} \{1 - \mathrm{Re}\,\mathrm{Tr}(U^{1\times 2}_{x,\mu,\nu})\} \right),$$

Algorithm 2 integrate

Require: $0 < n_{\text{ts}} \leq N_{\text{ts}}$, $\tau > 0$

```
Δτ = τ/noSteps[n_ts]
for i = 0 to noSteps[n_ts] do
   if n_ts == 1 then
      updateGauge(Δτ)
   else
      integrate(n_ts − 1, Δτ)
   end if
   updateMomenta(Δτ, monomialList[n_ts])
end for
```

The parameter c_1 can be set in the input file and $c_0 = 1 - 8c_1$. Note that $c_1 = 0$ corresponds to the Wilson plaquette gauge action.

The corresponding specific functions are defined in the files `det_monomial.c`, `detratio_monomial.c`, `ndpoly_monomial.c` and `gauge_monomial.c`. Additional monomials can easily be implemented by providing the corresponding functions as discussed above.

The integration scheme is implemented recursively, as exemplified in algorithm 2 for the leap-frog integration scheme (where we skipped half steps for simlicity). The updateMomenta function simply calls the derivative functions of all monomials that are integrated on timescale n_{ts} and updates the momenta P according to the time step $\Delta\tau$.

The recursive scheme for the integration can easily be extended to more involved integration schemes. The details can be found in the file `integrator.c`. We have implemented the leap-frog and the second order minimal norm [14] integrations schemes. They are named in the input file as `LEAPFROG` and `2MN`, respectively. These two can be mixed on different timescales. In addition we have implemented a position version of the second order minimal norm integration scheme, denoted by `2MNPOSITION` in the input file. The latter must not be mixed with the former two.

The MD update is summarised in algorithm 3. It computes the initial and final Hamiltonians and calls in between the integration function with the total number of timescales N_{ts} and the total trajectory length τ.

3.5.1 Reduced Precision in the MD Update

As shortly discussed previously, as long as the integration in the MD udpate is reversible and area preserving there is large freedom in choosing the integration scheme, but also the operator: it is not necessary to use the Dirac operator here, it can be any approximation to it. This is only useful if the acceptance rate is not strongly affected by such an approximation.

The code provides two possibilities to adapt the precision of the Dirac operator used in the MD update: the first is to reduce the precision in the inversions needed for the force computation. This causes reduced iteration numbers needed for the integration of one trajectory. The relevant input parameter is `ForcePrecision` available for each monomial. The precision needed in the acceptance and/or heatbath step can be adjusted separately using `AcceptancePrecision`. It is advisable to have the acceptance precision always close to machine precision.

Algorithm 3 MD update

1: $\mathcal{H} = \mathcal{H}' = 0$
2: **for** $i = 0$ to $N_{\text{monomials}}$ **do**
3: $\quad \mathcal{H}$ += monomial[i]→heat-bath-function
4: **end for**
5: integrate(N_{ts}, τ)
6: **for** $i = 0$ to $N_{\text{monomials}}$ **do**
7: $\quad \mathcal{H}'$ += monomial[i]→acceptance-function
8: **end for**
9: accept with probability $\min\{1, \exp(-\Delta\mathcal{H})\}$

The second possibility for influencing the Dirac operator is given by the reduced precision Dirac operator described in sub-section 3.4, which is switched on with the `UseSloppyPrecision` input parameter. The two possibilities can also be used in parallel.

Note that one should always test for reversibility violations as explained in sub-section 3.6.

3.5.2 Chronological Solver

The idea of the chronological solver method (or similar methods [15]) is to optimize the initial guess for the solution used in the solver. To this end the history of N_{CSG} last solutions of the equation $M^2\chi = \phi$ is saved and then a linear combination of the fields χ_i with coefficients c_i is used as an initial guess for the next inversion. M stands for the operator to be inverted and has to be replaced by the different ratios of operators used in this paper.

The coefficients c_i are determined by solving

$$\sum_i \chi_j^\dagger M^2 \chi_i c_i = \chi_j^\dagger \phi \tag{10}$$

with respect to the coefficients c_i. This is equivalent to minimising the functional that is minimised by the CG inverter itself.

The downside of this method is that the reversibility violations increase significantly by one or two orders of magnitude in the Hamiltonian when the CSG is switched on and all other parameters are kept fixed. Therefore one has to adjust the residues in the solvers, which increases the number of matrix vector multiplications again. Our experience is that the methods described in the previous sub-section are more effective in particular in the context of multiple time scale integration, because the CSG is most effective for small values of $\Delta\tau$.

The input parameters is the `CSGHistory` parameter available for the relevant monomials. Setting it to zero means no chronological solver, otherwise this parameter specifies the number of last solutions N_{CSG} to be saved.

3.6 Online Measurements

The HMC program includes the possibility to perform a certain number of measurements after every trajectory *online*, whether or not the configuration is stored on disk.

Some of those are performed per default, namely all that are written to the output file `output.data`:

1. the plaquette expectation value, defined as:

$$\langle P \rangle = \frac{1}{6V} \sum_{\mu,\nu=1 \;\; 1 \leq \mu < \nu}^{4} \mathrm{Re}\,\mathrm{Tr}(U^{1\times 1}_{x,\mu,\nu})\,,$$

   where V is the global lattice volume.

2. the rectangle expectation value, defined as:

$$\langle R \rangle = \frac{1}{12V} \sum_{\mu,\nu=1 \;\; \mu \neq \nu}^{4} \mathrm{Re}\,\mathrm{Tr}(U^{1\times 2}_{x,\mu,\nu})$$

3. $\Delta\mathcal{H} = \mathcal{H}' - \mathcal{H}$ and $\exp(-\Delta\mathcal{H})$.

See the overview section for details about the `output.data` file. These observables all come with no extra computational cost.

Optionally, other online measurements can be performed, which – however – need in general extra inversions of the Dirac operator. First of all the computation of certain correlation functions is implemented. They need *one* extra inversion of the Dirac operator, as discussed in Ref. [16], using the one-end-trick. Define a stochastic source ξ as follows

$$\lim_{R\to\infty} [\xi_i^* \xi_j] = \delta_{ij}, \quad \lim_{R\to\infty} [\xi_i \xi_j] = 0\,. \tag{11}$$

Here R labels the number of samples and i all other degrees of freedom. Then

$$[\phi_i^{r*} \phi_j^r]_R = M^{-1*}_{ik} \cdot M^{-1}_{jk} + \text{noise}\,, \tag{12}$$

if ϕ was computed from

$$\phi_j^r = M^{-1}_{jk} \xi_k^r\,.$$

Having in mind the γ_5-hermiticity property of the Wilson and Wilson twisted mass Dirac propagator $G_{u,d}$, i.e.

$$G_u(x,y) = \gamma_5 G_d(y,x)^\dagger \gamma_5$$

it is clear that eq. (12) can be used to evaluate

$$C_\pi(t) = \langle \mathrm{Tr}[G_u(0,t)\gamma_5 G_d(t,0)\gamma_5] \rangle = \langle \mathrm{Tr}[G_u(0,t) G_u(0,t)^\dagger] \rangle$$

with only one inversion. But, even if the one gamma structure at the source is fixed to be γ_5 due to the γ_5-hermiticity trick, we are still free to insert any γ-structure Γ at the source, i.e. we can evaluate any correlation function of the form

$$C_{P\Gamma}(t) = \langle \mathrm{Tr}[G_u(0,t)\gamma_5 G_d(t,0)\Gamma] \rangle = \langle \mathrm{Tr}[G_u(0,t) G_u(0,t)^\dagger \gamma_5 \Gamma] \rangle\,.$$

Useful combinations of correlation functions are $\langle PP \rangle$, $\langle PA \rangle$ and $\langle PV \rangle$, with

$$P^\alpha = \bar\chi \gamma_5 \frac{\tau^\alpha}{2} \chi\,, \quad V^\alpha_\mu = \bar\chi \gamma_\mu \frac{\tau^\alpha}{2} \chi\,, \quad A^\alpha_\mu = \bar\chi \gamma_5 \gamma_\mu \frac{\tau^\alpha}{2} \chi$$

From $\langle PP \rangle$ one can extract the pseudo scalar mass, and – in the twisted mass case – the pseudo scalar decay constant. $\langle PA \rangle$ can be used together with $\langle PP \rangle$ to extract the so called PCAC quark mass and $\langle PV \rangle$ to measure the renormalisation constant Z_{V}. For details we refer the reader to Ref. [16].

These online measurements are controlled with the two following input parameters: `PerformOnlineMeasurements` to switch them on or off and to specify the frequency `OnlineMeasurementsFreq`. The three correlation functions are saved in files named `onlinemeas.n`, where `n` is the trajectory number. Every file contains five columns, specifying the type, the operator type and the Euclidean time t. The last two columns are the values of the correlation function itself, $C(t)$ and $C(-t)$, respectively. The type is equal to 1, 2 or 6 for the $\langle PP \rangle$, the $\langle PA \rangle$ and the $\langle PV \rangle$ correlation functions. The operator type is for online measurements always equal to 1 for local source and sink (no smearing of any kind), and the time runs from 0 to $T/2$. Hence, $C(-t) = C(T-t)$. $C(-0)$ and $C(-T/2)$ are set to zero for convenience.

In addition to correlation functions also the minimal and the maximal eigenvalues of the $(\gamma_5 D)^2$ can be measured.

An online measurement not related to physics, but related to the algorithm are checks of reversibility violations. The HMC algorithm is exact, if and only if the integration scheme is reversible. On a computer with finite precision this is only guaranteed up to machine precision. These violations can be estimated by integrating one trajectory forward and then backward in Monte Carlo time. The difference $\delta\Delta\mathcal{H}$ among the original Hamiltonian $\mathcal{H}$ and the final one $\mathcal{H}''$ after integrating back can serve as one measure for those violations, another one is provided by the difference among the original gauge field U and the final one U''

$$\delta\Delta U = \frac{1}{12V} \sum_{x,\mu} \sum_{i,j} (U_{x,\mu} - U''_{x,\mu})^2_{i,j}$$

where we indicate with the $\delta\Delta$ that this is obtained after integrating a trajectory forward and backward in time. The results for $\delta\Delta\mathcal{H}$ and $\delta\Delta U$ are stored in the file `return_check.data`. The relevant input parameters are `ReversibilityCheck` and `ReversibilityChec`

3.7 Iterative Solver and Eigensolver

There are several iterative solvers implemented in the tmLQCD package for solving

$$D\ \chi = \phi$$

for χ. The minimal residual (MR), the conjugate gradient (CG), the conjugate gradient squared (CGS), the generalised minimal residual (GMRES), the generalised conjugate residual and the stabilised bi-conjugate gradient (BiCGstab). For details regarding these algorithms we refer to Refs. [8, 17].

For the `hmc_tm` executable only the CG and the BiCGstab solvers are available, while all the others can be used in the `invert` executables. Most of them are both available with and without even/odd preconditioning. For a performance comparison we refer to Ref. [18, 10].

The stopping criterion is implemented in two ways: the first is an absolute stopping criterion, i.e. the solver is stopped when the squared norm of the residual vector (depending on the solver this might be the iterated residual or the real residual) fulfills

$$\|r\|^2 < \epsilon^2 \,.$$

The second is relative to the source vector, i.e.

$$\frac{\|r\|^2}{\|\phi\|^2} < \epsilon^2 \,.$$

The value of ϵ^2 and the choice of relative or absolute precision can be influenced via input parameters.

The reduced precision Dirac operator, as discussed in sub-section 3.4, is available for the CG solver. In the CG solver the full precision Dirac operator is only required at the beginning of the CG search, because the relative size of the contribution to the resulting vector decreases with the number of iterations. Thus, as soon as a certain precision is achieved in the CG algorithm we can switch to the reduced precision Dirac operator without spoiling the precision of the final result. We switch to the lower precision operator at a precision of $\sqrt{\epsilon}$ in the CG search, when aiming for a final precision of $\epsilon < 1$.

The eigensolver used to compute the eigenvalues (and vectors) of $(\gamma_5 D)^2$ is the so called Jacobi-Davidson method [19, 20]. For a discussion for the application of this algorithm to lattice QCD we refer again to Ref. [18, 10].

All solver related files can be found in the sub-directory `solver`. Note that there are a few more solvers implemented which are, however, in an experimental status.

3.8 Stout Smearing

Smearing techniques have become an important tool to reduce ultraviolet fluctuations in the gauge fields. One of those techniques, coming with the advantage of being usable in the MD update, is usually called stout smearing [21].

The $(n+1)^{\text{th}}$ level of stout smeared gauge links is obtained iteratively from the n^{th} level by

$$U_\mu^{(n+1)}(x) \;=\; e^{i\,Q_\mu^{(n)}(x)}\, U_\mu^{(n)}(x).$$

We refer to the unsmeared ("thin") gauge field as $U_\mu \equiv U_\mu^{(0)}$. The SU(3) matrices Q_μ are defined via the staples C_μ:

$$\begin{aligned}
Q_\mu^{(n)}(x) &= \frac{i}{2}\Big[U_\mu^{(n)}(x)C_\mu^{(n)\dagger}(x) - \text{h.c.}\Big] \;-\; \frac{i}{6}\,\text{Tr}\,\Big[U_\mu^{(n)}(x)C_\mu^{(n)\dagger}(x) - \text{h.c.}\Big]\,, \\
C_\mu^{(n)} &= \sum_{\nu\neq\mu} \rho_{\mu\nu}\,\Big(U_\nu^{(n)}(x)U_\mu^{(n)}(x+\hat{\nu})U_\nu^{(n)\dagger}(x+\hat{\mu}) \\
&\qquad + U_\nu^{(n)\dagger}(x-\hat{\nu})U_\mu^{(n)}(x-\hat{\nu})U_\nu^{(n)}(x-\hat{\nu}+\hat{\mu})\Big)\,,
\end{aligned}$$

where in general $\rho_{\mu\nu}$ is the smearing matrix. In the tmLQCD package we have only implemented isotropic 4-dimensional smearing, i.e., $\rho_{\mu\nu} = \rho$.

Currently stout smearing is only implemented for the `invert` executables. I.e. the gauge field can be stout smeared at the beginning of an inversion. The input parameters are `UseStoutSmearing`, `StoutRho` and `StoutNoIterations`.

	TR_0	TR_1	TR_2
input-file	sample-hmc0.input	sample-hmc2.input	sample-hmc3.input
$L^3 \times T$	$4^3 \times 4$	$4^3 \times 4$	$4^3 \times 4$
S_G	Wilson	TlSym	Iwasaki
β	6.0	3.3	1.95
κ	0.177	0.17	0.163260
$2\kappa\mu_q$	0.177	0.01	0.002740961
$2\kappa\bar{\mu}$	–	0.1105	–
$2\kappa\bar{\epsilon}$	–	0.0935	–
$\langle P \rangle$	0.62457(7)	0.53347(17)	0.5951(2)
$\langle R \rangle$	–	0.30393(22)	0.3637(3)

Table 1:

3.9 Random Number Generator

The random number generator used in the code is the one proposed by Martin Lüscher and usually known under the name RANLUX [22]. A single and double precision implementation was made available by the author under the GNU General Public License and can be downloaded [23]. For convenience it is also included in the tmLQCD package.

The source code ships with a number of sample input files. They are located in the `sample-input` sub-directory. They are small volume $V = 4^4$ test runs designated to measure for instance the average plaquette values.

Such a testrun can be performed for instance on a scalar machine by typing

```
./hmc_tm -f sample-hmc0.input .
```

Depending on the environment you are running in, you may need to adjust the input parameters to match the maximal run-time and so on. The expected average plaquette values are quoted in table 1 and also in the sample input files.

3.10 Benchmark Executable

Another useful test executable is a benchmark code. It can be build by typing `make benchmark` and it will, when run, measure the performance of the Dirac operator. It can be run in the serial or parallel case. It reads its input from a file `benchmark.input` and the relevant input parameters are the following:

```
L = 4
T = 4
NrXProcs = 2
NrYProcs = 2
NrZProcs = 2
UseEvenOdd = yes
UseSloppyPrecision = no
```

In case of even/odd preconditioning the performance of the hopping matrix is evaluated, in case of no even/odd the performance of the Dirac operator. The important part of the output of the code is as follows

```
[...]

 (1429 Mflops [64 bit arithmetic])

communication switched off
 (2592 Mflops [64 bit arithmetic])

The size of the package is 36864 Byte
The bandwidth is 662.91 + 662.91   MB/sec
```

The bandwidth is not measured directly but computed from the performance difference among with and without communication and the package size. In case of a serial run the output is obviously reduced.

4 File Formats and IO

4.1 Fermion Field File Formats

We note at the beginning, that we do not use a different format for source or sink fermion fields. They are both stored using the same lime records. The meta-data stored in the same lime-packed file is supposed to clarify all other things.

4.1.1 Propagators

Here we mainly concentrate on storing propagators (sink). The file can contain only sources, or both, source and sink. We (plan to) support four different formats

1. (arbitrary number of) sink, no sources
2. (arbitrary number of) source/sink pairs
3. one source, 12 sink
4. one source, 4 sink

This is very similar to the formats in use in parts of the US community. However, they use XML as a markup language, which we don't (yet) use.

We adopt the SCIDAC chechsum for gauge and propagator files.

Every source and sink has to be in a seperate lime record. The order in one file for the four formats mentioned above is supposed to be

1. sink, no sources: -
2. source/sink pairs: first source, then sink
3. one source, 12 sink: first source, then 12 sinks
4. one source, 4 sink: first source, then 4 sinks

All fermion field files must have a record indicating the type. The record itself is of type `propagator-type` and the record has a single entry (ascii string) which can contain one of

- `DiracFermion_Sink`
- `DiracFermion_Source_Sink_Pairs`
- `DiracFermion_ScalarSource_TwelveSink`
- `DiracFermion_ScalarSource_FourSink`

Those strings are also used in the input files of the hmc code for the input parameter `PropagatorType`. The binary data corresponding to one Dirac fermion field (source or sink) is then stored with at least two (three) records. The first is of type
`etmc-propagator-format`
and should contain the following information:

```
<?xml version="1.0" encoding="UTF-8"?>
<etmcFormat>
  <field>diracFermion</field>
  <precision>32</precision>
  <flavours>1</flavours>
  <lx>4</lx>
  <ly>4</ly>
  <lz>4</lz>
  <lt>4</lt>
</etmcFormat>
```

The `flavours` entry must be set to `1` for a one flavour propagator (flavour diagonal case) and to `2` for a two flavour propagator (flavour non-diagonal 2-flavour operator). In the former case there follows one record of type `scidac-binary-data`, which is identical to the SCIDAC format, containing the fermion field. In the latter case there follow two of such records, the first of which is the upper flavour. To be precise, lets call the two flavours s and c. Then we always store the s component first and then the c component. Any number of other records can be added for convenience.

The first two types are by now supported. In the future the other two might follow.

The indices in the binary data `scidac-binary-data` are in the following order:

$$t, z, y, x, s, c\,,$$

where t is the slowest and colour the fastest running index. The binary data is stored big endian and either in single or in double precision, depending on the `precision` parameter in the `etmc-propagator-format` record.

The γ-matrix convention is the one of the HMC code (see subsection A) and the operator is normalised to

$$D = \frac{1}{2}[\gamma_\mu(\nabla_\mu + \nabla^*_\mu) - a\nabla^*_\mu\nabla_\mu] + m_0 + i\mu\gamma_5\tau^3\,.$$

For the non-degenerate case with the two flavour operator the following operator is inverted:

$$D_{\rm nd} = \frac{1}{2}[\gamma_\mu(\nabla_\mu + \nabla^*_\mu) - a\nabla^*_\mu\nabla_\mu] + m_0 + i\bar{\mu}\gamma_5\tau_1 + \bar{\epsilon}\tau_3$$

4.1.2 Source Fields

Source fields are, as mentioned before, stored with the same binary data format. There are again several types of source files possible:

- `DiracFermion_Source`
- `DiracFermion_ScalarSource`
- `DiracFermion_FourScalarSource`
- `DiracFermion_TwelveScalarSource`

This type is stored in a record called `source-type` in the lime file. There might be several sources stored within the same file. We add a format reacord `etmc-source-format` looking like

```
<?xml version="1.0" encoding="UTF-8"?>
<etmcFormat>
  <field>diracFermion</field>
  <precision>32</precision>
  <flavours>1</flavours>
  <lx>4</lx>
  <ly>4</ly>
  <lz>4</lz>
  <lt>4</lt>
  <spin>4</spin>
  <colour>3</colour>
</etmcFormat>
```

with obvious meaning for every `scidac-binary-data` record within the lime packed file. This format record also allows to store a subset of the whole field, e.g. a timeslize.

5 Interfaces to external QCD libraries

5.1 QUDA: A library for QCD on GPUs

The QUDA [24, 25, 26] interface is complementary to tmLQCD's own CUDA kernels for computations on the GPU by Florian Burger. So far it is exclusively used for inversions.

5.1.1 Design goals of the interface

The QUDA interface has been designed with the following goals in mind, sorted by priority:

1. *Safety.* Naturally, highest priority is given to the correctness of the output of the interface. This is trivially achieved by always checking the final residual on the CPU with the default tmLQCD routines.

2. *Ease of use.* Within the operator declarations of the input file (between `BeginOperator` and `EndOperator`) a simple flag `UseQudaInverter` is introduced which, when set to `yes`, will let QUDA perform the inversion of that operator. The operators `TMWILSON, WILSON, DBTMWILSON` and `CLOVER` are supported.[2]

3. *Minimality.* Minimal changes in the form of `#ifdef QUDA` precompiler directives to the tmLQCD code base. The main bulk of the interface lies in a single separate file `quda_interface.c` (with corresponding header file). In the file `operators.c`, the QUDA library is initialized when an operator is initialized which has set `UseQudaInverter = yes`. There, the actual call to the inverter is conditionally replaced with a call to the QUDA interface.

4. *Performance.* The higher priority of the previous items results in small performance detriments. In particular:

   - tmLQCD's θ-boundary conditions are not compatible with QUDA's 8 and 12 parameter reconstruction of the gauge fields (as of QUDA-0.7.0). Therefore reconstruction/compression is deactivated by default, although it may be activated via the input file, see below.
   - The gaugefield is transferred each time to the GPU before the inversion starts in order to ensure not to miss any modifications of the gaugefield.

5.1.2 Installation

If not already installed, you have to install QUDA first. Download the most recent version from `http://lattice.github.io/quda/`. Note that QUDA version $\geq$ 0.7.0 is required (chiral gamma basis).

QUDA can be installed without any dependencies, consider, e.g., the following minimal configuration:

```
cmake \
  -DQUDA_DIRAC_STAGGERED=OFF \
  -DQUDA_DIRAC_DOMAIN_WALL=OFF \
  -DQUDA_DIRAC_WILSON=ON \
  -DQUDA_DIRAC_CLOVER=ON \
  -DQUDA_DIRAC_TWISTED_MASS=ON \
  -DQUDA_DIRAC_TWISTED_CLOVER=ON \
  -DQUDA_DIRAC_NDEG_TWISTED_MASS=ON \
  -DQUDA_DYNAMIC_CLOVER=ON \
  -DQUDA_MPI=ON \
  -DQUDA_INTERFACE_MILC=OFF \
  -DQUDA_INTERFACE_QDP=ON \
  -DQUDA_MULTIGRID=ON \
  -DQUDA_GPU_ARCH=sm_37 \
  ${path_to_quda}
```

where `$CUDADIR` and `$MPI_PATH` have to be set appropriately. `$QUDADIR` is your choice for the installation directory of QUDA. Note that for Wilson clover quarks, you should set

[2] `DBCLOVER` is supported by the interface but not by QUDA as of version 0.7.0.

-DQUDA_DYNAMIC_CLOVER=OFF, whereas the opposite is strictly necessary for twisted mass clover quarks, which means that you will require two QUDA and tmLQCD builds for the time being if you intend to work with both actions. Note also that if you want to use QUDA in a scalar build of tmLQCD, you should remove the lines --enable-multi-gpu and --with-mpi=$MPI_PATH in the configuration (and probably you want to replace the MPI compilers). In order to profit from QUDA's autotuning functionality, set the environment variable QUDA_RESOURCE_PATH to a directory of your choice. Every time that you update your QUDA installation or change some of the many QUDA environment variables, the files in the directory will have to be deleted or a new directory chosen. It is convenient to base the directory dynamically on the head git commit of your QUDA source tree as well as the value of the QUDA_ENABLE_GDR environment variable. There may be other environment variables which make one set of auto-tuning results incompatible with another.

Once QUDA is installed, a minimal configuration of tmLQCD could look like, e.g.,

```
./configure CC=mpicc \
--prefix=$TMLQCDDIR \
--with-limedir=$LIMEDIR \
--with-lapack=<linker-flags> \
--enable-mpi \
--with-mpidimension=4 \
CXX=mpiCC \
--with-qudadir=$QUDADIR \
--with-cudadir=${CUDADIR}/lib
```

Note that a C++ compiler is required for linking against the QUDA library, therefore set CXX appropriately. $QUDADIR is where you installed QUDA in the previous step and $CUDADIR is required again for linking.

5.1.3 Usage

Any main program that reads and handles the operator declaration from an input file can easily be set up to use the QUDA inverter by setting the UseExternalInverter flag to quda. For example, in the input file for the invert executable, add the flag to the operator declaration as

```
BeginOperator TMWILSON
  2kappaMu = 0.05
  kappa = 0.177
  UseEvenOdd = yes
  Solver = CG
  SolverPrecision = 1e-14
  MaxSolverIterations = 1000
  UseExternalInverter = quda
EndOperator
```

and the operator of interest will be inverted using QUDA. The initialization of QUDA is done automatically within the operator initialization, the QUDA library should be finalized by a call to _endQuda() just before finalizing MPI. When you use the QUDA interface for work that is being published, don't forget to cite [24, 25, 26].

5.1.4 General settings

Some properties of the QUDA interface can be configured via the `ExternalInverter` section.

```
BeginExternalInverter QUDA
  FermionBC = [theta, pbc, apbc]
EndExternalInverter
```

The option `FermionBC` shown above forces twisted (`theta`), periodic (`pbc`) or antiperiodic (`apbc`) temporal quark field boundary conditions. This setting exists because at the time of writing (2017.12.28), there seems to be a bug or incompatibility in QUDA which causes (anti-)periodic boundary conditions with gauge compression to produce incorrect propagators.

5.1.5 QUDA-MG interface

The interface has support for the QUDA Multigrid (MG) solver implementation and allows a number of parameters to be adjusted in order to tune the MG setup. The defaults for these parameters follow the recommendations of `https://github.com/lattice/quda/wiki/Multigrid-Solver`, which also provides useful hints for further tuning. Although some of the parameters can be set on a per-level basis, the interface currently only exposes a single setting for all levels, where appropriate. The K-cycle is used by default and there is currently no user-exposed option for changing this.

The MG-preconditioned GCR solver is selected as follows:

```
BeginOperator TMWILSON
  2kappaMu = 0.05
  kappa = 0.177
  UseEvenOdd = yes
  Solver = mg
  SolverPrecision = 1e-18
  MaxSolverIterations = 200
  UseExternalInverter = quda
  UseSloppyPrecision = single
EndOperator
```

The MG setup can be tuned using the following parameters in the `BeginExternalInverter QUDA` section:

- `MGNumberOfLevels`: number of levels to be used in the MG, 3 is usually ideal but 2 can be similarly efficient depending on the quark mass (positive integer, default 3)
- `MGSetupSolver`: solver used for generating null vectors. `CG` or `BiCGstab` (default `CG`). Usage of `BiCGstab` may be recommended for Wilson or clover Wilson quarks.
- `MGSetupSolverTolerance`: relative target residual (unsquared!) during setup phase. (positive float, default $1 \cdot 10^{-6}$)
- `MGSetupMaxSolverIterations`: maximum number of iterations during setup phase. (positive integer, default 1000)

- `MGCoarseSolverTolerance`: unsquared relative target residual on the coarse grids. (positive float, default 0.25)
- `MGNumberOfVectors`: number of null vectors to compute on a per-level basis. (possible values $[24, 32]$, default 24)
- `MGCoarseMaxSolveriterations`: maximum number of iterations on coarse grids. (positive integer, default 75)
- `MGEnableSizeThreeBlocks`: By default, QUDA has limited support for size 3 aggregates. If set to *yes*, the automatic blocking algorithm will attempt to use them for lattice extents divisible by 3 when the local lattice extent at a given level is smaller than 16 aggregate sites. This requires you to instantiate the necessary block sizes in QUDA (see comments below). (boolean *yes* or *no*, default *no*)
- `MGBlockSizes[X,Y,Z,T]`: aggregate sizes on each level. When these are set for a given lattice dimension, the automatic blocking algorithm for that dimension is overridden and the specified blockings are forced. When the required aggregate sizes are not instantiated in QUDA, the setup phase will fail with an informative error message. (comma-separated list of integers, for a three level solver, for example, this needs to be specified for the first and second level)
- `MGSmootherTolerance`: unsquared relative target residual of the smoother on all levels. (positive float, default 0.25)
- `MGSmootherPreIterations`: number of smoothing steps before coarse grid correction. (zero or positive integer, default 0)
- `MGSmootherPostIterations`: number of smoothing steps after prolongation. (zero or positive integer, default 4)
- `MGOverUnderRelaxationFactor`: Over- or under-relaxation factor. (positive float, default 0.85)
- `MGCoarseMuFactor`: Scaling factor for twisted mass on a per-level basis, accelerates convergence and reduces condition numer of coarse grid. From experience it seems that it's reasonable to set this > 1.0 only on the coarsest level, but it might also help on intermediate levels. If running with twisted mass, this should always be set and tuned for maximum efficiency. (positive float, usually > 1.0, default 8.0 from the second level upwards).
- `MGRunVerify`: Check GPU coarse operators against CPU coarse operators and verify Galerkin projectors during setup phase. This is usually fast enough to always be performed, although sometimes it seems to fail even though the setup works fine. (*yes* or *no*, default *yes*)

If no blocking is specified manually, the aggregation parameters are set automatically as follows:

- A default block size of 4 is attempted if the MPI-partitioned fine or aggregate lattice extent is larger or equal to 16 lattice sites.

- If the number of aggregate lattice sites in a given direction is even and smaller than 16, a block size of 2 is used.
- The option `MGEnableSizeThreeBlocks` can be set to `yes`. Then, for levels coarser than the fine grid, extents smaller than 16 and divisible by 3, a block size of 3 will be used. This will almost certainly require the addition of instantiations of block sizes to QUDA in the restrictor and transfer operator. (`lib/restrictor.cu` and `lib/transfer_util.cu`)
- In all other cases, aggregation is disabled for this direction and level. This includes, for instance, extents divisible by primes other than 2 or 3.

Note that at the time of writing (2017.12.28), only double-single mixed-precision is supported for the MG-preconditioned GCR solver and the solve will abort if a double-half precision solve is attempted.

A typical MG setup might look like this for twisted mass clover quarks:

```
BeginExternalInverter QUDA
  MGNumberOfLevels = 3
  MGSetupSolver = cg
  MGSetupSolverTolerance = 1e-6
  MGSetupMaxSolverIterations = 1000
  MGCoarseSolverTolerance = 0.25
  MGCoarseSolverIterations = 75
  MGSmootherTolerance = 0.25
  MGSmootherPreIterations = 2
  MGSmootherPostIterations = 4
  MGOverUnderRelaxationFactor = 0.85
  MGCoarseMuFactor = 1.0, 1.0, 12.0
  MGNumberOfVectors = 24, 24, 32
  MGRunVerify = yes
  MGEnableSizeThreeBlocks = no
EndExternalInverter
```

Alternatively, a blocking can be specified manually:

```
BeginExternalInverter QUDA
  MGNumberOfLevels = 3
  MGBlockSizesX = 4, 3
  MGBlockSizesY = 4, 3
  MGBlockSizesZ = 6, 4
  MGBlockSizesT = 6, 4
  MGSetupSolver = cg
  MGSetupSolverTolerance = 1e-6
  MGSetupMaxSolverIterations = 1000
  MGCoarseSolverTolerance = 0.25
  MGCoarseSolverIterations = 75
  MGSmootherTolerance = 0.25
  MGSmootherPreIterations = 2
```

```
  MGSmootherPostIterations = 4
  MGOverUnderRelaxationFactor = 0.85
  MGCoarseMuFactor = 1.0, 1.0, 12.0
  MGRunVerify = yes
  MGEnableSizeThreeBlocks = no
EndExternalInverter
```

5.1.6 More advanced settings

To achieve higher performance you may choose single (default) or even half precision as sloppy precision for the inner solver of the mixed precision inverter with reliable updates. After `BeginOperator` and before `EndOperator` set `UseSloppyPrecision = double|single|half`. The MG-preconditioned GCR solver only works in double-single mixed precision, but the null vectors are stored in half precision as recommended by Kate Clark.

To activate compression of the gauge fields (in order to save bandwidth and thus to achieve higher performance), set `UseCompression = 8|12|18` within `BeginOperator` and `EndOperator`. The default is 18 which corresponds to no compression. Note that if you use compression, trivial (anti)periodic boundary conditions will be applied to the gauge fields, instead of the default θ-boundary conditions. As a consequence, the residual check on tmLQCD side will fail. Moreover, compression is not applicable when using general θ-boundary conditions in the spatial directions. If trying to do so, compression will be de-activated automatically and the user gets informed via the standard output. The `FermionBC` setting can be used to force particular temporal boundary conditions to be applied to the gauge field in the Dirac operator.

5.1.7 Functionality

The QUDA interface can currently be used to invert `TMWILSON, WILSON, DBTMWILSON` and `CLOVER` within a 4D multi-GPU (MPI) parallel environment with CG, BICGSTAB or MG-preconditioned GCR. QUDA uses even-odd preconditioning, if wanted (`UseEvenOdd = yes`), and the interface is set up to use a mixed precision solver by default. For more details on the QUDA settings check the function `_initQuda()` in `quda_interface.c`.

5.2 DDalphaAMG: A library for multigrid preconditioning on LQCD

DD-αAMG [27] is an Adaptive Aggregation-based Domain Decomposition Multigrid method for Lattice QCD. A library named DDalphaAMG is publicly available[3] and it contains the full method with additional development tools. DD-αAMG has been successfully extended to $N_f = 2$ twisted mass fermions in [28].

5.2.1 Installation

Download the Twisted Mass version of the DDalphaAMG library at

`https://github.com/sbacchio/DDalphaAMG.`

[3] `https://github.com/DDalphaAMG/DDalphaAMG`

The Makefile should be ready for being compiled in a Intel environment. You may want to change the environment or just set some variables; you can do it editing the first lines of the Makefile:

```
CC = mpiicc

# --- CFLAGS -----------------------------------------
CFLAGS_gnu = -std=gnu99 -Wall -pedantic -fopenmp -O3 -ffast-math -msse4.2
CFLAGS_intel = -std=gnu99 -Wall -pedantic -qopenmp -O3  -xHOST
CFLAGS = $(CFLAGS_intel)
```

The library can be installed with

```
make -j library LIMEDIR="/your/lime/installation/dir"
```

and tmLQCD can be configured and compiled by using

```
autoreconf -f
./configure YOUR_OPTIONS --with-DDalphaAMG="/path/to/DDalphaAMG/dir"
make -j
```

5.2.2 Usage

For calling the solver with a standard setting of parameters, it is just necessary to use `DDalphaAMG` as a solver:

```
BeginOperator TMWILSON
  2kappaMu = 0.05
  kappa = 0.177
  Solver = DDalphaAMG
  SolverPrecision = 1e-14
  MaxSolverIterations = 100
EndOperator
```

More options are available and explained in the next section. At the first call of the solver, a setup phase will be run and then the same setup will be used for all the inversions with the same configuration. Be aware that the change of configuration at the moment is supported just for HMC simulations for which specific parameters are defined.

5.2.3 More advanced settings

For tuning purpose, several parameters of DDalphaAMG can be set inside the section `DDalphaAMG` and here after the complete list of implemented parameters:

```
BeginDDalphaAMG
  MGOMPNumThreads = 1
  MGBlockX = 4
  MGBlockY = 4
  MGBlockZ = 4
  MGBlockT = 4
  MGNumberOfVectors = 24
```

```
  MGNumberOfLevels = 3
  MGCoarseMuFactor = 5
  MGSetupIter = 5
  MGCoarseSetupIter = 3
  MGSetup2KappaMu = 0.001
  MGMixedPrecision = yes
  MGdtauUpdate = 0.05
  MGrhoUpdate = 0.0
  MGUpdateSetupIter = 1
EndDDalphaAMG
```

Not all the parameters have to be use and for all of them a standard value is defined. Here a brief explanation:

`MGOMPNumThreads`: the DDalphaAMG library does not take advantages on exploiting hyper-threading; while most of the applications of tmLQCD do. For this reason the `OMPNumThreads` for DDalphaAMG has been separated by the standard one. If this parameter is not used, the value of `OMPNumThreads` is used.

`MGBlock?`: [4] block size in the directions X,Y,Z,T. The values have to divide the local size of the lattice and by default an optimal value is used.

`MGNumberOfVectors`: [4] number of vectors used in the fine level. This parameter require some tuning.

`MGNumberOfLevels`: number of levels for the multigrid method. Can take values from 1 (no multigrid) to 4. A value of 3 is suggested.

`MGCoarseMuFactor`: [4] multiplicative factor for the twisted mass term μ on the coarsest level. A good performance is achieved with a value between 3 and 6.

`MGSetupIter, MGCoarseSetupIter`: number of setup iterations in the fine and coarse grid respectively. For the fine grid a value between 3 and 5 is suggested. For the coarse grid 2, 3 iterations should be enough.

`MGSetup2KappaMu`: out of the physical point, the solver could have advantages on running the setup with a lower mu, closer to the physical point.

`MGMixedPrecision`: using the mixed precision solver, a speed-up of 20% can be achieved. One has to be careful that the mixed precision solver do not restart more than once and that the restarted relative residual (in double precision) is not order of magnitude higher than the one in single precision, see Section 5.2.4. In that case the mixed precision solver is not suggested.

`MGdtauUpdate`: for HMC, $d\tau$ interval after that the setup is updated. If 0, it will be updated every time the configuration is changed.

`MGrhoUpdate`: for HMC, rho value of the monomial at which the setup have to be updated. It can be combined with `MGdtauUpdate` or used standalone.

[4] for a better understanding of these parameters we strongly suggest the reading of the numerical results presented in [28]

`MGUpdateSetupIter:` for HMC, number of setup iterations to do on the fine level when the setup has to be updated.

`MGNumberOfShifts:` for MG in multi-shift systems, number of shifted linear systems, N, to be solved by DDalphaAMG. MG will solve the N smaller shifts.

`MGMMSMass:` for MG in multi-shift systems, alternative to the previous. MG will solve all the mass-shifts smaller than the given value.

5.2.4 Output analysis

Running tmLQCD programs with the option `-v`, the full output of DDalphaAMG is shown. Here some hints on the informations given. Just before the setup, the full set of parameters is printed, with an output similar to the following:

```
+-------------------------------------------------------+
| 3-level method                                        |
| postsmoothing K-cycle                                 |
| FGMRES + red-black multiplicative Schwarz             |
|          restart length: 10                           |
|                      m0: -0.430229                    |
|                     csw: +1.740000                    |
|                      mu: +0.001200                    |
+-------------------------------------------------------+
|   preconditioner cycles: 1                            |
|            inner solver: minimal residual iteration   |
|               precision: single                       |
+---------------------- depth  0 ----------------------+
|          global lattice: 96  48  48  48               |
|           local lattice: 16  8   8   24               |
|           block lattice: 4   4   4   4                |
|        post smooth iter: 2                            |
|     smoother inner iter: 4                            |
|              setup iter: 3                            |
|            test vectors: 24                           |
+---------------------- depth  1 ----------------------+
|          global lattice: 24  12  12  12               |
|           local lattice: 4   2   2   6                |
|           block lattice: 2   2   2   2                |
|        post smooth iter: 2                            |
|     smoother inner iter: 4                            |
|              setup iter: 3                            |
|            test vectors: 28                           |
+---------------------- depth  2 ----------------------+
|          global lattice: 12  6   6   6                |
|           local lattice: 2   1   1   3                |
|           block lattice: 1   1   1   1                |
|      coarge grid solver: odd even GMRES               |
|              iterations: 25                           |
|                  cycles: 40                           |
|               tolerance: 5e-02                        |
|                      mu: +0.012000                    |
```

```
+-----------------------------------------------------------+
|          K-cycle length: 5                                |
|        K-cycle restarts: 2                                |
|       K-cycle tolerance: 1e-01                            |
+-----------------------------------------------------------+
```

You may want to check that all the parameters agree to what expected and a good set of parameters is presented in [28].

5.2.5 Warnings and error messages

5.3 QPhiX: Optimised kernels and solvers for Intel Processors

The QPhiX [29] interface provides a library of MPI- and OpenMP-parallel linear operators and solvers for Wilson-type lattice fermions as well as a code-generator for the kernels employed by these operators. QPhiX has been extended to include all the operators relevant for tmLQCD, including the non-degenerate operator with and without the clover term.

5.3.1 Installation

If not already installed, you have to install QPhiX first. At the time of writing, the version with support for all twisted mass operators is in branch

- `devel` branch of `https://github.com/JeffersonLab/qphix`.

It depends on QMP (`https://github.com/usqcd-software/qmp`), which is built and installed through the usual `configure, make, make install` mechanism.

QPhiX is built using CMake and requires the availability of python 3, as well as the jinja2 library (`https://jinja.pocoo.org`). The latter can easily be installed via the pip package installer:

```
pip install --user jinja
```

QPhiX AVX2 Compilation: In order to compile QPhiX using GCC on an AVX2 machine, CMake is called in this way:

```
CXX=mpicxx \
CXXFLAGS="-mavx2 -mtune=core-avx2 -march=core-avx2 -std=c++11 -O3 -fopenmp" \
cmake -Disa=avx2  \
      -DQMP_DIR=${QMP_INSTALL_DIR} \
      -Dparallel_arch=parscalar \
      -Dhost_cxx=g++ \
      -Dhost_cxxflags="-std=c++11 -O3" \
```

```
      -Dtwisted_mass=TRUE \
      -Dtm_clover=TRUE \
      -Dclover=TRUE \
      -Dtesting=FALSE  \
      -DCMAKE_INSTALL_PREFIX=${QPHIX_INSTALL_DIR} ${QPHIX_SRC_DIR}
```

where `QMP_INSTALL_DIR`, `QPHIX_INSTALL_DIR` and `QPHIX_SRC_DIR` should be replaced with the QMP installation directory, the target installation directory for QPhiX and the QPhiX source directory respectively.

In the command above:

- `-Dtesting=FALSE` disables the building of all tests, which would additionally require QDP++ to be available
- `-Dhost_cxx` and `-Dhost_cxxflags` define the compiler used for building the code generator executables. This can be any compiler and `g++` works just fine for this purpose.+

QPhiX AVX512 Compilation: On a KNL-based machine like Marconi-KNL instead, the Intel compiler and Intel MPI library should be used:

```
CXX=mpiicpc \
CXXFLAGS="-xKNL -std=c++11 -O3 -qopenmp" \
CFLAGS="-xKNL -O3 -std=c99 -qopenmp" \
cmake -Disa=avx512  \
      -DQMP_DIR==${QMP_INSTALL_DIR} \
      -Dparallel_arch=parscalar \
      -Dhost_cxx=g++ \
      -Dhost_cxxflags="-std=c++11 -O3" \
      -Dtwisted_mass=TRUE \
      -Dtm_clover=TRUE \
      -Dclover=TRUE \
      -Dtesting=FALSE  \
      -DCMAKE_INSTALL_PREFIX=${QPHIX_INSTALL_DIR} ${QPHIX_SRC_DIR}
```

Note that for Skylake, the correct code for targetting vectorisation is `SKYLAKE-AVX512`.

tmLQCD AVX512 Compilation: Once QPhiX is built and installed, tmLQCD can be configured as follows on a KNL AVX512 machine, for example:

```
$ cd ${TMLQCD_SRC_DIR}
$ autoconf
$ cd ${TMLQCD_BUILD_DIR}
```

```
$ ${TMLQCD_SRC_DIR}/configure \
  --host=x86_64-linux-gnu \
  --with-limedir=${LIME_INSTALL_DIR} \
  --with-lemondir=${LEMON_INSTALL_DIR} \
  --with-mpidimension=4 --enable-omp --enable-mpi \
  --disable-sse2 --disable-sse3 \
  --with-lapack="-Wl,--start-group ${MKLROOT}/lib/intel64/libmkl_intel_lp64.a
                 ${MKLROOT}/lib/intel64/libmkl_core.a
                 ${MKLROOT}/lib/intel64/libmkl_intel_thread.a
                 -Wl,--end-group -lpthread -lm -ldl" \
  --disable-halfspinor --enable-gaugecopy \
  --enable-alignment=64 \
  --enable-qphix-soalen=4 \
  --with-qphixdir=${QPHIX_INSTALL_DIR} \
  --with-qmpdir=${QMP_INSTALL_DIR} \
  CC=mpiicc CXX=mpiicpc F77=ifort \
  CFLAGS="-O3 -std=c99 -qopenmp -xKNL" \
  CXXFLAGS="-O3 -std=c++11 -qopenmp -xKNL" \
  LDFLAGS="-qopenmp"
```

IMPORTANT: On AVX512 machines, for some reason, the half-spinor tmLQCD operators do not work. This is likely related to MPI and alignment, but we were unable to resolve it at the time of writing. As a result, `--disable-halfspinor` is passed when building on these architectures.

`--enable-qphix-soalen=4` sets the QPhiX *structure of array* (SoA) length, which defines the size of the innermost direction in the blocked data structures in QPhiX. *Half* the *local* lattice extent in X direction, $L_x/2$, has to be divisible by this number. Setting this equal to the double-precision SIMD length on a given architecture means that a full double-precision SIMD vector can be loaded in a single instruction, while values below the SIMD vector length will result in multiple load and store instructions, while all computation are always carried out on full vectors.

For now, the same SoA length is used for all supported arithmetic precisions as this facilitates thinking about possible parallelisation strategies. On AVX512 machines, a setting this to 8 is optimal whereas 4 is recommended for AVX2.

Note that compiling for KNL requires cross-compilation (if not on a KNL build node), but it seems to be sufficient to specify `--host=x86_64-linux-gnu` for all test programs to compile correctly during the configuration stage.

The QPhiX interface can be combined with DDαAMG without problems, but building together with the QUDA interface is only possible using GCC or clang, since QUDA is not compatible with the Intel compiler. On the QPhiX side, this will result in a potentially significant reduction of performance.

5.3.2 Usage

QPhiX global parameters: The blocking and threading parameters for QPhiX are passed by adding the following section to the tmLQCD input file:

```
BeginExternalInverter QPHIX
  # physical cores per MPI task
  NCores = 34
  # block sizes (see qphix papers for details)
  By = 8
  Bz = 8
  # split the processing of time slices into this many
  # independent blocks
  MinCt = 1
  # (hyper-)thread geometry per core
  # ompnumthreads = NCores * Sy * Sz
  # if only a single thread per core is launched
  # these should both be left as '1'
  Sy = 1
  Sz = 2
  # paddings in XY and XYZ blocks
  PadXY = 1
  PadXYZ = 0
EndExternalInverter
```

- `NCores`: number of physical cores per MPI task. On KNL, it might even make sense to specify twice the number of physical cores since each core contains two vector processing units (VPUs). Another possiblity would be to specify the number of tiles per MPI tasks and consider cores and VPUs throuh `Sz` and `Sy` below. The only case that has been tested for performance is to set this equal to the number of physical cores per MPI task.

- `By, Bz`: the QPhiX data structures are organised into blocks which can be efficiently loaded into CPU caches. `By` and `Bz` define the size of these blocks in the Y and Z lattice dimensions. The local lattice extent in the given dimension should be divisible by the respective block extent. Generally, 4 or 8 are good values and the larger of the two may be preferable.

- `MinCt`: Processing of time slices is split into MinCt blocks. This is useful for dual-socket systems when running with a single MPI task per node. In this case, this should be set to 2 which will allow the kernels to run in a NUMA-friendly fashion. The local T dimension must be divisible by this number. On KNL, this should be set to 1. Note that in all cases tested so far, running with 2 MPI tasks per node on dual-socket systems was superior.

- `Sy, Sz`: thread blocking parameters. When multiple threads share resources (this is the case for cores and hyperthreads on KNL, for example), these parameters make it possible to consider this in the volume-traversal loops implemented in QPhiX. On KNL, the only setting which has been tested for performance is to set this equal to 2, given that `NCores` has been set to the number of physical cores. `Sz` then splits the local Z direction among two hyperthreads.

- `PadXY(Z)`: Adds padding to the QPhiX data structures which may result in higer overall performance. Only value tested on KNL is `PadXY=1` and `PadXYZ=0`.

IMPORTANT: The global setting `OmpNumThreads` should be set to `NCores * Sy * Sz`, otherwise the QPhiX interface will abort execution.

QPhiX operator / monomial parameters: QPhiX solvers are available in operators for inversions and monomials for performing HMC with the same parameters. For a clover determinant, using QPhiX solvers instead of tmLQCD-native ones would be achieved as follows:

```
BeginMonomial CLOVERDET
  Timescale = 1
  kappa = 0.1394267
  2KappaMu = 0.00069713350
  CSW = 1.69
  rho = 0.238419657
  MaxSolverIterations = 5000
  AcceptancePrecision =  1.e-21
  ForcePrecision = 1.e-16
  Name = cloverdetlight
  Solver = mixedcg
  UseExternalInverter = qphix
  UseCompression = 12
  UseSloppyPrecision = single
EndMonomial
```

- `Solver`: specify the solver type (see below for the solvers supported by the QPhiX interface).
- `UseExternalInverter`: the external inverter `qphix` should be used for this monomial.
- `UseCompression`: gauge compression should be used (`12`). This improves performance by increasing the flop/byte ratio. Twisted boundary conditions are fully supported in all directions.
- `UseSloppyPrecision`: for a solver using just a single arithmetic precision (like basic `cg` or `bicgstab`), this sets the arithmetic precision employed. For a mixed-precision solver such as `mixedcg`, this sets the arithmetic precision of the inner solver.

Supported solvers: The QPhiX interface provides support for the solvers:

- `cg`
- `mixedcg`
- `bicgstab`

- `mixedbicgstab`
- `cgmms` (single-flavour rational *monomials* only)
- `cgmmsnd` (two-flavour non-degenerate rational *monomials* only)

Note that as usual, `bicgstab` and `mixedbicgstab` do not converge for twisted mass fermions at maximal twist.

Note also that if the solver is any of `cg`, `bicgstab`, `cgmms` or `cgmmsnd` and `UseSloppyPrecision = single` is set, the selected solver will run in single precision arithmetic. Only `mixedcg` and `mixedbicgstab` are mixed precision solvers which use the sloppy precision as the precision of the inner solver.

5.3.3 Notes about QPhiX performance

- **MPI Task and Thread pinning:** QPhiX performs best when MPI tasks are pinned to the resources assigned to them and when threads are bound to individual cores or hyperthreads. This is conventiently achieved for Intel MPI by taking control of resource pinning from the job scheduler, setting `I_MPI_PIN=1` and `I_MPI_PIN_DOMAIN=N`, where `N` should be set to a pinning domain appropriate for the chosen parallelisation. In order to then distribute application threads in an optimal fashion across the cores that have been assigned to a given MPI task in this way, setting `KMP_AFFINITY="balanced,granularity=fine"` is recommended.
  - Generally, the size of the pinning domain is the number of hyperthreads per core supported by the CPU in question, times the number of cores that a given MPI task should run on. If hyperthreading is disabled on the machine in question, it is simply the number of cores that each MPI task should be allocated.
- **Halo packing overheads:** In QPhiX, communication in the Y and especially in the X dimension incurs halo packing overheads. These are usually greater than the gain from having a more balanced surface to bulk ratio. It is thus recommended to do as little MPI parallelisation as possible in the X dimension and similarly limit it in the Y dimension, although the latter is less performance-critical.
- **OmniPath networking performance:** On machines based on Intel Knight's Landing or Skylake processors with OmniPath networks, best single node performance is generally reached with a single MPI task per node (KNL) or a single MPI task per socket (Skylake). However, until computing centres have implemented the recommendations of Ref. [30], more than one or two MPI tasks per node are required to saturate network bandwidth on these machines. Generally, 4 to 8 MPI tasks per node seem to work well.

References

[1] C.-N. Yang and R. L. Mills, Phys. Rev. **96**, 191 (1954).

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

[3] **ALPHA** Collaboration, R. Frezzotti, P. A. Grassi, S. Sint and P. Weisz, JHEP **08**, 058 (2001), `hep-lat/0101001`.

[4] R. Frezzotti and G. C. Rossi, Nucl. Phys. Proc. Suppl. **128**, 193 (2004), `hep-lat/0311008`.

[5] T. Chiarappa *et al.*, Eur. Phys. J. **C50**, 373 (2007), `arXiv:hep-lat/0606011`.

[6] `http://www.netlib.org/lapack/`.

[7] USQCD, `http://usqcd.jlab.org/usqcd-docs/c-lime/`.

[8] Y. Saad, *Iterative Methods for sparse linear systems*, 2nd ed. (SIAM, 2003).

[9] `http://www-unix.mcs.anl.gov/mpi/`.

[10] T. Chiarappa *et al.*, Comput. Sci. Disc. **01**, 015001 (2008), `arXiv:hep-lat/0609023`.

[11] M. Lüscher, Nucl. Phys. Proc. Suppl. **106**, 21 (2002), `arXiv:hep-lat/0110007`.

[12] M. Lüscher, `http://luscher.web.cern.ch/luscher/QCDpbm/`.

[13] **SciDAC** Collaboration, R. G. Edwards and B. Joo, Nucl. Phys. Proc. Suppl. **140**, 832 (2005), `hep-lat/0409003`.

[14] T. Takaishi and P. de Forcrand, Phys. Rev. **E73**, 036706 (2006), `arXiv:hep-lat/0505020`.

[15] R. C. Brower, A. R. Levi and K. Orginos, Nucl. Phys. Proc. Suppl. **42**, 855 (1995), `hep-lat/9412004`.

[16] **ETM** Collaboration, P. Boucaud *et al.*, Comput. Phys. Commun. **179**, 695 (2008), `arXiv:0803.0224 [hep-lat]`.

[17] A. Meister, *Numerik linearer Gleichungssysteme* (vieweg, 1999).

[18] T. Chiarappa *et al.*, Nucl. Phys. Proc. Suppl. **140**, 853 (2005), `arXiv:hep-lat/0409107`.

[19] G. L. G. Sleijpen and H. A. V. der Vorst, SIAM Journal on Matrix Analysis and Applications **17**, 401 (1996).

[20] R. Geus, *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*, PhD thesis, Swiss Federal Institute Of Technology Zürich, 2002.

[21] C. Morningstar and M. J. Peardon, Phys. Rev. **D69**, 054501 (2004), `arXiv:hep-lat/0311018`.

[22] M. Lüscher, Comput. Phys. Commun. **79**, 100 (1994), `arXiv:hep-lat/9309020`.

[23] M. Lüscher, http://luscher.web.cern.ch/luscher/ranlux/.

[24] M. Clark, R. Babich, K. Barros, R. Brower and C. Rebbi, Comput.Phys.Commun. **181**, 1517 (2010), arXiv:0911.3191 [hep-lat].

[25] R. Babich *et al.*, arXiv:1109.2935 [hep-lat].

[26] A. Strelchenko, C. Alexandrou, G. Koutsou and A. V. Aviles-Casco, PoS **LATTICE2013**, 415 (2014), arXiv:1311.4462 [hep-lat].

[27] A. Frommer, K. Kahl, S. Krieg, B. Leder and M. Rottmann, SIAM J. Sci. Comput. **36**, A1581 (2014), arXiv:1303.1377 [hep-lat].

[28] C. Alexandrou *et al.*, arXiv:1610.02370 [hep-lat].

[29] B. Joó *et al.*, *Lattice QCD on Intel® Xeon PhiTM Coprocessors* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013), pp. 40–54.

[30] P. Boyle *et al.*, arXiv:1711.04883 [cs.DC].

[31] http://www.ginac.de/CLN/.

[32] R. Frezzotti and K. Jansen, Phys. Lett. **B402**, 328 (1997), hep-lat/9702016.

[33] T. A. DeGrand and P. Rossi, Comput. Phys. Commun. **60**, 211 (1990).

[34] K. Jansen and C. Liu, Comput. Phys. Commun. **99**, 221 (1997), hep-lat/9603008.

[35] M. Hasenbusch and K. Jansen, Nucl.Phys. **B659**, 299 (2003), arXiv:hep-lat/0211042 [hep-lat].

[36] M. A. Clark and A. D. Kennedy, Phys. Rev. Lett. **98**, 051601 (2007), arXiv:hep-lat/0608015.

[37] M. Luscher, arXiv:1002.4232 [hep-lat].

[38] M. Luscher and S. Schaefer, Comput.Phys.Commun. **184**, 519 (2013), arXiv:1206.2809 [hep-lat].

A γ and Pauli Matrices

In the following we specify our conventions for γ- and Pauli-matrices.

A.1 γ-matrices

We use the following convention for the Dirac γ-matrices:

$$\gamma_0 = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}, \quad \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & +i & 0 & 0 \\ +i & 0 & 0 & 0 \end{pmatrix},$$
$$\gamma_2 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & +1 & 0 \\ 0 & +1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}, \quad \gamma_3 = \begin{pmatrix} 0 & 0 & -i & 0 \\ 0 & 0 & 0 & +i \\ +i & 0 & 0 & 0 \\ 0 & -i & 0 & 0 \end{pmatrix}.$$

In this representation γ_5 is diagonal and reads

$$\gamma_5 = \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

A.2 Pauli-matrices

For the Pauli-matrices acting in flavour space we use the following convention:

$$1_f = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \tau^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \tau^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \tau^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

B Initialising the PHMC

The function $1/\sqrt{s}$ in the interval $[\epsilon, 1]$ can be approximated using polynomials or rational functions of different sorts. In the tmLQCD package we use Chebysheff polynomials, which are easy to construct. They can be constructed as to provide a desired overall precision in the interval $[\epsilon, 1]$.

The roots of the polynomial $P_{n,\epsilon}$ are needed for the evaluation of the force. Even though the roots come in complex conjugate pairs, for our case the roots cannot be computed analytically, hence we need to determine them numerically. Such an evaluation requires usually high precision. This is why these roots need to be determined *before* a PHMC run using an external program, i.e. they cannot be computed at the beginning of a run in the `hmc_tm` program.

Such an external program ships with the tmLQCD code, which is located in the `util/laguere` directory[5]. It is based on Laguerre's method and uses the Class Library for Numbers (CLN) [31], which provides arbitrary precision data types. In order to compute roots the CLN library must be available, which is free software.

[5]We thank Istvan Montvay for providing us with his code.

Taking for granted that the CLN library is available, the procedure for computing the roots is as follows: assuming the non-degenerate Dirac operator has eigenvalues in the interval $[\tilde{s}_{\min}, \tilde{s}_{\max}]$, i.e. $\epsilon = \tilde{s}_{\min}/\tilde{s}_{\max}$, and the polynomial degree is n. Edit the file `chebyRoot.H` and set the variable `EPSILON` to the value of ϵ. Moreover, set the variable `MAXPOW` to the degree n. Adapt the `Makefile` to your local installation and compile the code by typing `make`. After running the `ChebyRoot` program successfully, you should find two files in the directory

1. `Square_root_BR_roots.dat`:
   which contains the roots of the polynomial in bit-reverse order [32].

2. `normierungLocal.dat`:
   which contains a normalisation constant.

Copy these two files into the directory where you run the code and adjust the input parameters to match *exactly* the values used for the root computation. I.e. the input parameters `StildeMin`, `StildeMax` and `DegreeOfMDPolynomial` must be set appropriately in the `NDPOLY` monomial. The maximal degree $\tilde{n}_{\max}$ for $\tilde{P}$ can be influenced using `MaxPtildeDegree`.

The minimal and maximal eigenvalue of the non-degenerate flavour doublet can be computed as an online measurement. The frequency can be specified in the `NDPOLY` monomial with the input parameter `ComputeEVFreq` and they are written to the file called `phmc.data`. Note that this is not a cheap operation in terms of computer time. However, if the approximation interval of the polynomial is chosen wrongly the algorithm performance might deteriorate drastically, in particular if the upper bound is set wrongly. It is therefore advisable to introduce some security measure in particular in the value of $\tilde{s}_{\max}$.

While the degree of the MD polynomial can be adjusted in the input file, the degree of $\tilde{P}$ used in the heatbath and acceptance steps is computed at the beginning of the run depending on the precision specified in the input file. The procedure is as follows: Compute the first $\tilde{n}_{\max}$ coefficients d_i of the polynomial. Then determine the degree $\tilde{n}$ of $\tilde{P}$ such that

$$\sum_{i=n}^{n_{\max}} d_i < \epsilon$$

where ϵ is set using the input parameter `PrecisionPtilde`.

C Even/Odd Preconditioning

C.1 HMC Update

In this section we describe how even/odd [33, 34] preconditioning can be used in the HMC algorithm in presence of a twisted mass term. Even/odd preconditioning is implemented in the tmLQCD package in the HMC algorithm as well as in the inversion of the Dirac operator, and can be used optionally.

We start with the lattice fermion action in the hopping parameter representation in the χ-basis written as

$$\begin{aligned} S[\chi,\bar{\chi},U] = \sum_x \Big\{ & \bar{\chi}(x)[1+2i\kappa\mu\gamma_5\tau^3]\chi(x) \\ & - \kappa\bar{\chi}(x)\sum_{\mu=1}^{4}\Big[U(x,\mu)(r-\gamma_\mu)\chi(x+a\hat{\mu}) \\ & + U^\dagger(x-a\hat{\mu},\mu)(r+\gamma_\mu)\chi(x-a\hat{\mu})\Big]\Big\} \\ \equiv & \sum_{x,y}\bar{\chi}(x)M_{xy}\chi(y)\ . \end{aligned} \tag{13}$$

For convenience we define $\tilde{\mu} = 2\kappa\mu$. Using the matrix M one can define the hermitian (two flavour) operator:

$$Q \equiv \gamma_5 M = \begin{pmatrix} Q_+ & \\ & Q_- \end{pmatrix} \tag{14}$$

where the sub-matrices $Q_\pm$ can be factorised as follows (Schur decomposition):

$$\begin{aligned} Q^\pm &= \gamma_5 \begin{pmatrix} 1\pm i\tilde{\mu}\gamma_5 & M_{eo} \\ M_{oe} & 1\pm i\tilde{\mu}\gamma_5 \end{pmatrix} = \gamma_5 \begin{pmatrix} M_{ee}^\pm & M_{eo} \\ M_{oe} & M_{oo}^\pm \end{pmatrix} \\ &= \begin{pmatrix} \gamma_5 M_{ee}^\pm & 0 \\ \gamma_5 M_{oe} & 1 \end{pmatrix} \begin{pmatrix} 1 & (M_{ee}^\pm)^{-1}M_{eo} \\ 0 & \gamma_5(M_{oo}^\pm - M_{oe}(M_{ee}^\pm)^{-1}M_{eo}) \end{pmatrix}\ . \end{aligned} \tag{15}$$

Note that $(M_{ee}^\pm)^{-1}$ can be computed to be

$$(1\pm i\tilde{\mu}\gamma_5)^{-1} = \frac{1\mp i\tilde{\mu}\gamma_5}{1+\tilde{\mu}^2}. \tag{16}$$

Using $\det(Q) = \det(Q_+)\det(Q_-)$ the following relation can be derived

$$\begin{aligned} \det(Q_\pm) &\propto \det(\hat{Q}_\pm) \\ \hat{Q}_\pm &= \gamma_5(M_{oo}^\pm - M_{oe}(M_{ee}^\pm)^{-1}M_{eo})\,, \end{aligned} \tag{17}$$

where $\hat{Q}_\pm$ is only defined on the odd sites of the lattice. In the HMC algorithm the determinant is stochastically estimated using pseudo fermion field ϕ_o: Now we write the determinant with pseudo fermion fields:

$$\begin{aligned} \det(\hat{Q}_+\hat{Q}_-) &= \int \mathcal{D}\phi_o\,\mathcal{D}\phi_o^\dagger\ \exp(-S_{\rm PF}) \\ S_{\rm PF} &\equiv\ \phi_o^\dagger\ \left(\hat{Q}_+\hat{Q}_-\right)^{-1}\phi_o\,, \end{aligned} \tag{18}$$

where the fields ϕ_o are defined only on the odd sites of the lattice. In order to compute the force corresponding to the effective action $S_{\rm PF}$ we need the variation of $S_{\rm PF}$ with respect to the gauge fields (using $\delta(A^{-1}) = -A^{-1}\delta A A^{-1}$):

$$\begin{aligned} \delta S_{\rm PF} &= -[\phi_o^\dagger(\hat{Q}_+\hat{Q}_-)^{-1}\delta\hat{Q}_+(\hat{Q}_+)^{-1}\phi_o + \phi_o^\dagger(\hat{Q}_-)^{-1}\delta\hat{Q}_-(\hat{Q}_+\hat{Q}_-)^{-1}\phi_o] \\ &= -[X_o^\dagger\delta\hat{Q}_+Y_o + Y_o^\dagger\delta\hat{Q}_-X_o] \end{aligned} \tag{19}$$

with X_o and Y_o defined on the odd sides as

$$X_o = (\hat{Q}_+\hat{Q}_-)^{-1}\phi_o, \quad Y_o = (\hat{Q}_+)^{-1}\phi_o = \hat{Q_-}X_o \ , \tag{20}$$

where $(\hat{Q}_\pm)^\dagger = \hat{Q}^\mp$ has been used. The variation of $\hat{Q}_\pm$ reads

$$\delta\hat{Q}_\pm = \gamma_5 \left(-\delta M_{oe}(M_{ee}^\pm)^{-1}M_{eo} - M_{oe}(M_{ee}^\pm)^{-1}\delta M_{eo}\right), \tag{21}$$

and one finds

$$\begin{aligned} \delta S_{\mathrm{PF}} &= -(X^\dagger \delta Q_+ Y + Y^\dagger \delta Q_- X) \\ &= -(X^\dagger \delta Q_+ Y + (X^\dagger \delta Q_+ Y)^\dagger) \end{aligned} \tag{22}$$

where X and Y are now defined over the full lattice as

$$X = \begin{pmatrix} -(M_{ee}^-)^{-1}M_{eo}X_o \\ X_o \end{pmatrix}, \quad Y = \begin{pmatrix} -(M_{ee}^+)^{-1}M_{eo}Y_o \\ Y_o \end{pmatrix}. \tag{23}$$

In addition $\delta Q_+ = \delta Q_- = \delta Q, M_{eo}^\dagger = \gamma_5 M_{oe}\gamma_5$ and $M_{oe}^\dagger = \gamma_5 M_{eo}\gamma_5$ has been used. Since the bosonic part is quadratic in the ϕ_o fields, the ϕ_o are generated at the beginning of each molecular dynamics trajectory with

$$\phi_o = \hat{Q}_+ R, \tag{24}$$

where R is a random spinor field taken from a Gaussian distribution with norm one.

C.1.1 Symmetric even/odd Preconditioning

One may write instead of eq. (15) the following symmetrical factorisation of $Q_\pm$:

$$Q_\pm = \gamma_5 \begin{pmatrix} M_{ee}^\pm & 0 \\ M_{oe} & M_{oo}^\pm \end{pmatrix} \begin{pmatrix} 1 & (M_{ee}^\pm)^{-1}M_{eo} \\ 0 & (1 - (M_{oo}^\pm)^{-1}M_{oe}(M_{ee}^\pm)^{-1}M_{eo}) \end{pmatrix} . \tag{25}$$

Where we can now re-define

$$\hat{Q}_\pm = \gamma_5(1 - (M_{oo}^\pm)^{-1}M_{oe}(M_{ee}^\pm)^{-1}M_{eo}) \tag{26}$$

With this re-definition the procedure is analogous to what we discussed previously. Only the vectors X and Y need to be modified to

$$\begin{aligned} X &= \begin{pmatrix} -(M_{ee}^-)^{-1}M_{eo}(M_{oo}^-)^{-1}X_o \\ X_o \end{pmatrix}, \\ Y &= \begin{pmatrix} -(M_{ee}^+)^{-1}M_{eo}(M_{oo}^+)^{-1}Y_o \\ Y_o \end{pmatrix}. \end{aligned} \tag{27}$$

Note that the variation of the action is still given by

$$\delta S_{\mathrm{PF}} = -\,\mathrm{Re}(X^\dagger \delta Q_+ Y) \ . \tag{28}$$

C.1.2 Mass non-degenerate flavour doublet

Even/odd preconditioning can also be implemented for the mass non-degenerate flavour doublet Dirac operator D_h eq. (5). Denoting

$$Q^h = \gamma_5 D_h$$

the even/odd decomposition is as follows

$$\begin{aligned} Q^h &= \begin{pmatrix} (\gamma_5 + i\bar{\mu}\tau^3 - \bar{\epsilon}\gamma_5\tau^1) & Q^h_{eo} \\ Q^h_{oe} & (\gamma_5 + i\bar{\mu}\tau^3 - \bar{\epsilon}\gamma_5\tau^1) \end{pmatrix} \\ &= \begin{pmatrix} Q^h_{ee} & 0 \\ Q^h_{oe} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & (Q^h_{ee})^{-1} Q_{eo} \\ 0 & \hat{Q}^h_{oo} \end{pmatrix} \end{aligned} \tag{29}$$

where $\hat{Q}^h_{oo}$ is given in flavour space by

$$\hat{Q}^h_{oo} = \gamma_5 \begin{pmatrix} 1 + i\bar{\mu}\gamma_5 - \frac{M_{oe}(1-i\bar{\mu}\gamma_5)M_{eo}}{1+\bar{\mu}^2-\bar{\epsilon}^2} & -\bar{\epsilon}\left(1 + \frac{M_{oe}M_{eo}}{1+\bar{\mu}^2-\bar{\epsilon}^2}\right) \\ -\bar{\epsilon}\left(1 + \frac{M_{oe}M_{eo}}{1+\bar{\mu}^2-\bar{\epsilon}^2}\right) & 1 - i\bar{\mu}\gamma_5 - \frac{M_{oe}(1+i\bar{\mu}\gamma_5)M_{eo}}{1+\bar{\mu}^2-\bar{\epsilon}^2} \end{pmatrix}$$

with the previous definitions of M_{eo} etc. The implementation for the HMC is very similar to the mass degenerate case. $\hat{Q}^h$ has again a hermitian conjugate given by

$$(\hat{Q}^h)^\dagger = \tau^1 \ \hat{Q}^h \ \tau^1$$

C.1.3 Combining Clover and Twisted mass term

We start again with the lattice fermion action in the hopping parameter representation in the χ-basis now including the clover term written as

$$\begin{aligned} S[\chi, \bar{\chi}, U] = \sum_x \Bigg\{ & \bar{\chi}(x)[1 + 2\kappa c_{SW} T + 2i\kappa\mu\gamma_5\tau^3]\chi(x) \\ & - \kappa\bar{\chi}(x) \sum_{\mu=1}^{4} \Big[U(x,\mu)(r - \gamma_\mu)\chi(x + a\hat{\mu}) \\ & + U^\dagger(x - a\hat{\mu}, \mu)(r + \gamma_\mu)\chi(x - a\hat{\mu}) \Big] \Bigg\} \\ \equiv & \sum_{x,y} \bar{\chi}(x) M_{xy} \chi(y) \,, \end{aligned} \tag{30}$$

with the clover term T. For convenience we define $\tilde{\mu} \equiv 2\kappa\mu$ and $\tilde{c}_{SW} = 2\kappa c_{SW}$. Using the matrix M one can define the (two flavour) operator:

$$Q \equiv \gamma_5 M = \begin{pmatrix} Q_+ & \\ & Q_- \end{pmatrix} \tag{31}$$

where the sub-matrices $Q_\pm$ can be factorised as follows (Schur decomposition):

$$\begin{aligned} Q^\pm &= \gamma_5 \begin{pmatrix} 1 + T_{ee} \pm i\tilde{\mu}\gamma_5 & M_{eo} \\ M_{oe} & 1 + T_{oo} \pm i\tilde{\mu}\gamma_5 \end{pmatrix} = \gamma_5 \begin{pmatrix} M^\pm_{ee} & M_{eo} \\ M_{oe} & M^\pm_{oo} \end{pmatrix} \\ &= \begin{pmatrix} \gamma_5 M^\pm_{ee} & 0 \\ \gamma_5 M_{oe} & 1 \end{pmatrix} \begin{pmatrix} 1 & (M^\pm_{ee})^{-1} M_{eo} \\ 0 & \gamma_5(M^\pm_{oo} - M_{oe}(M^\pm_{ee})^{-1} M_{eo}) \end{pmatrix} . \end{aligned} \tag{32}$$

Note that $(M_{ee}^{\pm})^{-1}$ cannot be computed as easily as in the case of Twisted mass fermions without clover term. Using $\det(Q) = \det(Q_+)\det(Q_-)$ the following relation can be derived

$$\begin{aligned} \det(Q_\pm) &\propto \det(1 + T_{ee} \pm i\tilde{\mu}\gamma_5)\det(\hat{Q}_\pm) \\ \hat{Q}_\pm &= \gamma_5((1 + T_{oo} \pm i\tilde{\mu}\gamma_5) - M_{oe}(1 + T_{ee} \pm i\tilde{\mu}\gamma_5)^{-1}M_{eo}) \,, \end{aligned} \tag{33}$$

where $\hat{Q}_\pm$ is only defined on the odd sites of the lattice. In the HMC algorithm the second determinant is stochastically estimated using pseudo fermion fields ϕ_o: now we write the determinant with pseudo fermion fields:

$$\begin{aligned} \det(\hat{Q}_+\hat{Q}_-) &= \int \mathcal{D}\phi_o \, \mathcal{D}\phi_o^\dagger \, \exp(-S_{\mathrm{PF}}) \\ S_{\mathrm{PF}} &\equiv \; \phi_o^\dagger \, \left(\hat{Q}_+\hat{Q}_-\right)^{-1} \phi_o \,, \end{aligned} \tag{34}$$

where the fields ϕ_o are defined only on the odd sites of the lattice. From the first factor in the Schur decomposition a second term needs to be taken into account in the effective action for the fermion determinant, this reads

$$\begin{aligned} S_{\mathrm{det}} &= -\log[\det(1 + T_{ee} + i\tilde{\mu}\gamma_5) \;\cdot\; \det(1 + T_{ee} - i\tilde{\mu}\gamma_5)] \\ &= -\,\mathrm{Tr}[\log(1 + T_{ee} + i\tilde{\mu}\gamma_5) + \log(1 + T_{ee} - i\tilde{\mu}\gamma_5)] \,. \end{aligned} \tag{35}$$

Note that for $\tilde{\mu} = 0$, $\det(1 + T_{ee})$ is real. For $\tilde{\mu} \neq 0$ however, $\det(1 + T_{ee} + i\tilde{\mu}\gamma_5)$ is the complex conjugate of $\det(1 + T_{ee} - i\tilde{\mu}\gamma_5)$ as the product of the two must be real. The latter can be seen from

$$\begin{aligned} &(1 + T_{ee} + i\tilde{\mu}\gamma_5) \;\cdot\; (1 + T_{ee} - i\tilde{\mu}\gamma_5) = \\ &(1 + T_{ee})^2 + \tilde{\mu}^2 \,. \end{aligned}$$

In order to compute the force corresponding to the effective action S_{PF} we need the variation of S_{PF} with respect to the gauge fields (using $\delta(A^{-1}) = -A^{-1}\delta A A^{-1}$):

$$\begin{aligned} \delta S_{\mathrm{PF}} &= -[\phi_o^\dagger(\hat{Q}_+\hat{Q}_-)^{-1}\delta\hat{Q}_+(\hat{Q}_+)^{-1}\phi_o + \phi_o^\dagger(\hat{Q}_-)^{-1}\delta\hat{Q}_-(\hat{Q}_+\hat{Q}_-)^{-1}\phi_o] \\ &= -[X_o^\dagger\delta\hat{Q}_+Y_o + Y_o^\dagger\delta\hat{Q}_-X_o] \end{aligned} \tag{36}$$

with X_o and Y_o defined on the odd sides as

$$X_o = (\hat{Q}_+\hat{Q}_-)^{-1}\phi_o, \quad Y_o = (\hat{Q}_+)^{-1}\phi_o = \hat{Q_-}X_o \;, \tag{37}$$

where $(\hat{Q}_\pm)^\dagger = \hat{Q}^\mp$ has been used. The variation of $\hat{Q}_\pm$ reads

$$\begin{aligned} \delta\hat{Q}_\pm = \gamma_5 \big(&\delta T_{oo} - \delta M_{oe}(M_{ee}^\pm)^{-1}M_{eo} - M_{oe}(M_{ee}^\pm)^{-1}\delta M_{eo} \\ &+ M_{oe}(M_{ee}^\pm)^{-1}\delta T_{ee}(M_{ee}^\pm)^{-1}M_{eo}\big) \,, \end{aligned} \tag{38}$$

and one finds

$$\begin{aligned} \delta S_{\mathrm{PF}} &= -(X^\dagger\delta Q_+Y + Y^\dagger\delta Q_-X) \\ &= -(X^\dagger\delta Q_+Y + (X^\dagger\delta Q_+Y)^\dagger) \end{aligned} \tag{39}$$

where X and Y are now defined over the full lattice as

$$X = \begin{pmatrix} -(M_{ee}^-)^{-1}M_{eo}X_o \\ X_o \end{pmatrix}, \quad Y = \begin{pmatrix} -(M_{ee}^+)^{-1}M_{eo}Y_o \\ Y_o \end{pmatrix}. \tag{40}$$

In addition $\delta Q_+ = \delta Q_- = \delta Q, M_{eo}^\dagger = \gamma_5 M_{oe} \gamma_5$ and $M_{oe}^\dagger = \gamma_5 M_{eo} \gamma_5$ has been used. δQ is now the original

$$\delta Q = \gamma_5 \begin{pmatrix} \delta T_{ee} & \delta M_{eo} \\ \delta M_{oe} & \delta T_{oo} \end{pmatrix}$$

defined over the full lattice. Since the bosonic part is quadratic in the ϕ_o fields, the ϕ_o are generated at the beginning of each molecular dynamics trajectory with

$$\phi_o = \hat{Q}_+ R, \tag{41}$$

where R is a random spinor field taken from a Gaussian distribution with norm one.

The additional bit in the action S_{det} needs to be treated separately. The variation of this part is

$$\delta S_{\mathrm{det}} = -\operatorname{Tr}\left\{ \left[(1 + i\tilde{\mu}\gamma_5 + T_{ee})^{-1} + (1 - i\tilde{\mu}\gamma_5 + T_{ee})^{-1} \right] \delta T_{ee} \right\} \ . \tag{42}$$

The main difference in between pure Twisted mass fermions and Twisted mass fermions plus clover term is that the matrices M_{ee} and M_{oo} need to be inverted numerically. A stable numerical method for this task needs to be devised.

For the implementation it is useful to compute the term

$$1 + T_{a\alpha,b\beta} = 1 + \frac{i}{2} c_{\mathrm{sw}} \kappa \sigma_{\mu\nu}^{\alpha\beta} F_{\mu\nu}^{ab}(x) \tag{43}$$

once for all x. This is implemented in `clover_leaf.c` in the routine `sw_term`. The twisted mass term is not included in this routine, as this would require double the storage for plus and minus μ, respectively. It is easier to add the twisted mass term in later on.

The term in eq. (43) corresponds to a 12×12 matrix in colour and spin which reduces to two complex 6×6 matrices per site because it is block-diagonal in spin (one matrix for the two upper spin components, one for the two lower ones). For each 6×6 matrix the off-diagonal 3×3 matrices are just hermitian conjugate to each other since $1 + T$ is hermitian. We therefore get away with storing two times three 3×3 complex matrices. These are stored in the array `sw[VOLUME][3][2]` of type `su3`. Here, `sw[x][0][0]` is the upper diagonal 3×3 matrix, `sw[x][1][0]` the upper off-diagnoal 3×3 matrix and `sw[x][2][0]` the lower diagonal matrix. The lower off-diagonal matrix would be the hermitian conjugate of `sw[x][1][0]`. The second 6×6 matrix is stored following the same conventions.

For computing S_{det}, we take into account the structure of the 24×24 flavour, spin and colour matrix:

$$M_{ee}(x) = \begin{pmatrix} A(x) + i\tilde{\mu} & 0 & 0 & 0 \\ 0 & B(x) - i\tilde{\mu} & 0 & 0 \\ 0 & 0 & A(x) - i\tilde{\mu} & 0 \\ 0 & 0 & 0 & B(x) + i\tilde{\mu} \end{pmatrix} , \tag{44}$$

where A and B are the 6×6 matrices mentioned above and are individually hermitian.

The implementation `sw_trace` in `clover_det.c` populates a temporary 6×6 array from the `sw` array and adds $+i\mu$ to the diagonal. Using $\det(\gamma_5) = 1$, the contribution to the effective action is then:

$$\begin{aligned} \log\det(M_{ee}) &= \log\left(|\det(A + i\tilde{\mu})|^2 \cdot |\det(B + i\tilde{\mu})|^2 \right) \\ &= \log\left(|\det(A + i\tilde{\mu})|^2 \right) + \log\left(|\det(B + i\tilde{\mu})|^2 \right) \ , \end{aligned} \tag{45}$$

where the summands are computed individually in a loop.

When it comes to computing the inverse of $1 \pm i\mu\gamma_5 + T_{ee}$, the dependence on the sign of μ is unavoidable. However, it is only needed for even (odd) sites, so we can use an array `sw_inv[VOLUME][4][2]` of type `su3` to store e.g. $+\mu$ at even and $-\mu$ at odd sites.

For evaluating the force for $S_{\rm det}$ in the function `sw_deriv` we have to compute

$$\mathrm{Tr}_{\rm dirac}[\; i\sigma_{\mu\nu}(1 + T_{ee}(x) \pm i\tilde{\mu}\gamma_5)^{-1} \;], \tag{46}$$

with $\sigma_{\mu\nu} = i\gamma_\mu\gamma_\nu \; \forall \mu \neq \nu$. The matrix $(1 + T_{ee}(x) \pm i\tilde{\mu}\gamma_5)^{-1}$ has the general structure

$$T_{\rm det} = \begin{pmatrix} u_0 & u_1 & 0 & 0 \\ u_3 & u_2 & 0 & 0 \\ 0 & 0 & l_0 & l_1 \\ 0 & 0 & l_3 & l_2 \end{pmatrix} .$$

Evaluating eq. (46) with matrix $T_{\rm det}$ for $\mu \neq \nu$ leads to the following terms

$$\begin{array}{cc} \mu\nu & \\ 01 & -i((l_1 - u_1) + (l_3 - u_3)) \\ 02 & (l_1 - u_1) - (l_3 - u_3) \\ 03 & i((l_2 - u_2) - (l_0 - u_0)) \\ 12 & i((l_2 + u_2) - (l_0 - u_0)) \\ 13 & (l_3 + u_3) - (l_1 + u_1) \\ 23 & -i(l_3 + u_3 + l_1 + u_1)\,. \end{array}$$

The force for $S_{\rm PF}$ can be computed in exactly the same way, even if in this case the matrix $T_{\rm PF}$ is a full matrix stemming from

$$\mathrm{Tr}_{\rm dirac}[\; i\sigma_{\mu\nu}(\gamma_5 Y(x) \otimes X^\dagger(x) + \gamma_5 X(x) \otimes Y^\dagger(x)) \;] \equiv \mathrm{Tr}_{\rm dirac}[\; i\sigma_{\mu\nu}\; T_{\rm PF} \;]\,. \tag{47}$$

$T_{\rm PF}$ is computed in the function `sw_spinor`. After multiplying with $\sigma_{\mu\nu}$ only the upper left and lower right blocks survive and the structure stays identical to the case discussed for $T_{\rm det}$. So in both cases, in order to compute the trace, we have to compute first in the functions `sw_spinor` and `sw_deriv` only

$$m_i = l_i - u_i\,, \quad p_i = l_i + u_i \quad i = 0, ..., 3\,. \tag{48}$$

The m_i and p_i are then passed on to the function `sw_all` which combines them to the correct insertion matrices, whereafter the traceless antihermitian part of it is computed. Finally, δT_{ee} is computed and combined with the insertion matrices.

C.1.4 Combining Clover and Nondegenerate Twisted mass term

Now we have

$$\hat{Q}^h_{oo} = \gamma_5(M^h_{oo} - (M^h_{oe}\; (M^h_{ee})^{-1}\; M^h_{eo})\,,$$

with

$$M^h_{oo|ee} = 1 + T_{oo|ee} + i\bar{\mu}\gamma_5\tau^3 - \bar{\epsilon}\tau^1\,. \tag{49}$$

The clover part $1 + T_{ee}$ is identical to the one in the $N_f = 2$ flavour case and stored in the array `sw`.

Because $1 + T_{ee}$ is hermitian, we can invert M^h_{ee} by

$$(1 + T_{ee} + i\bar{\mu}\gamma_5\tau^3 - \bar{\epsilon}\tau^1)^{-1} = \frac{(1 + T_{ee} - i\bar{\mu}\gamma_5\tau^3 + \bar{\epsilon}\tau^1)}{(1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2}\,. \tag{50}$$

In practice we compute $((1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2)^{-1}$ and store the result in the first `VOLUME/2` elements of the array `sw_inv`. Wherever the clover terms needs to be applied we then multiply with $((1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2)^{-1}$ and then with the nominator in eq. (50). One could save computing time here for the price of using more memory by storing the full inverse. Actually, it would be only slightly more than in the two flavour case: in addition we would only have to store $\bar{\epsilon}((1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2)^{-1}$. This would also allow to re-use a lot of the $N_f = 2$ flavour implementation.

The determinant we have to compute is

$$\det(Q^h) = \det[\gamma_5(1 + T_{ee} + i\bar{\mu}\gamma_5\tau^3 - \bar{\epsilon}\tau^1)]\ \det[\hat{Q}^h_{oo}].$$

Again, the first factor can be computed as S_{det}, for which we take into account the structure of the 24×24 flavour, spin and colour matrix:

$$M^h_{ee}(x) = \begin{pmatrix} A(x) + i\bar{\mu} & 0 & -\bar{\epsilon} & 0 \\ 0 & B(x) - i\bar{\mu} & 0 & -\bar{\epsilon} \\ -\bar{\epsilon} & 0 & A(x) - i\bar{\mu} & 0 \\ 0 & -\bar{\epsilon} & 0 & B(x) + i\bar{\mu} \end{pmatrix}, \tag{51}$$

where A and B are the 6×6 matrices mentioned in sub-section C.1.3 and are individually hermitian.

The determinant of the 24×24 matrix can be simplified by writing it as follows in 12×12 blocks in flavour:

$$\begin{aligned} \det(M^h_{ee}) &= \det \begin{pmatrix} K & D \\ D & K^\dagger \end{pmatrix} = \det \left[\begin{pmatrix} K & D - KD^{-1}K^\dagger \\ D & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & D^{-1}K^\dagger \\ 0 & 1 \end{pmatrix} \right] \\ &= -\det(D) \cdot \det(D - KD^{-1}K^\dagger) \\ &= \det(KK^\dagger - D^2) \\ &= \det(A^2 + \bar{\mu}^2 - \bar{\epsilon}^2) \cdot \det(B^2 + \bar{\mu}^2 - \bar{\epsilon}^2)\,, \end{aligned}$$

where the sign in the second line comes from the first term and in the third line the proportionality of D to the identity matrix was used.

The implementation `sw_trace_nd` in `clover_det.c` populates a temporary 6×6 array from the `sw` array, squares it and adds $\bar{\mu}^2 - \bar{\epsilon}^2$ to the diagonal. Using $\det(\gamma_5) = 1$, the contribution to the effective action is then:

$$\log\det(M_{ee}) = \log\left(\det(A^2 + \bar{\mu}^2 - \bar{\epsilon}^2) \cdot \det(B^2 + \bar{\mu}^2 - \bar{\epsilon}^2)\right). \tag{52}$$

For the variation of this term we have to compute now

$$\mathrm{Tr}_{\mathrm{dirac,flavour}}[\ i\sigma_{\mu\nu}(1 + T_{ee}(x) + i\bar{\mu}\gamma_5\tau^3 - \bar{\epsilon}\tau^1)^{-1}\ ]\,, \tag{53}$$

which is equal to

$$\mathrm{Tr}_{\mathrm{dirac,flavour}}\left[\ i\sigma_{\mu\nu}\frac{(1 + T_{ee} - i\bar{\mu}\gamma_5\tau^3 + \bar{\epsilon}\tau^1)}{(1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2}\ \right]. \tag{54}$$

The trace in flavour simplifies the computation to

$$\mathrm{Tr}_{\mathrm{dirac}}\left[\ i\sigma_{\mu\nu}\frac{2(1 + T_{ee})}{(1 + T_{ee})^2 + \bar{\mu}^2 - \bar{\epsilon}^2}\ \right]. \tag{55}$$

This can be treated analogously to the degenerate case described above.

C.2 Inversion

In addition to even/odd preconditioning in the HMC algorithm as described above, it can also be used to speed up the inversion of the fermion matrix.

Due to the factorization (15) the full fermion matrix can be inverted by inverting the two matrices appearing in the factorization

$$\begin{pmatrix} M_{ee}^{\pm} & M_{eo} \\ M_{oe} & M_{oo}^{\pm} \end{pmatrix}^{-1} = \begin{pmatrix} 1 & (M_{ee}^{\pm})^{-1} M_{eo} \\ 0 & (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo}) \end{pmatrix}^{-1} \begin{pmatrix} M_{ee}^{\pm} & 0 \\ M_{oe} & 1 \end{pmatrix}^{-1} .$$

The two factors can be simplified as follows:

$$\begin{pmatrix} M_{ee}^{\pm} & 0 \\ M_{oe} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} (M_{ee}^{\pm})^{-1} & 0 \\ -M_{oe}(M_{ee}^{\pm})^{-1} & 1 \end{pmatrix}$$

and

$$\begin{aligned} &\begin{pmatrix} 1 & (M_{ee}^{\pm})^{-1} M_{eo} \\ 0 & (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo}) \end{pmatrix}^{-1} \\ &= \begin{pmatrix} 1 & -(M_{ee}^{\pm})^{-1} M_{eo} (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo})^{-1} \\ 0 & (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo})^{-1} \end{pmatrix} . \end{aligned}$$

The complete inversion is now performed in two separate steps: First we compute for a given source field $\phi = (\phi_e, \phi_o)$ an intermediate result $\varphi = (\varphi_e, \varphi_o)$ by:

$$\begin{pmatrix} \varphi_e \\ \varphi_o \end{pmatrix} = \begin{pmatrix} M_{ee}^{\pm} & 0 \\ M_{oe} & 1 \end{pmatrix}^{-1} \begin{pmatrix} \phi_e \\ \phi_o \end{pmatrix} = \begin{pmatrix} (M_{ee}^{\pm})^{-1} \phi_e \\ -M_{oe}(M_{ee}^{\pm})^{-1} \phi_e + \phi_o \end{pmatrix} .$$

This step requires only the application of M_{oe} and $(M_{ee}^{\pm})^{-1}$, the latter of which is given by Eq (16). The final solution $\psi = (\psi_e, \psi_o)$ can then be computed with

$$\begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix} = \begin{pmatrix} 1 & (M_{ee}^{\pm})^{-1} M_{eo} \\ 0 & (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo}) \end{pmatrix}^{-1} \begin{pmatrix} \varphi_e \\ \varphi_o \end{pmatrix} = \begin{pmatrix} \varphi_e - (M_{ee}^{\pm})^{-1} M_{eo} \psi_o \\ \psi_o \end{pmatrix} ,$$

where we defined

$$\psi_o = (M_{oo}^{\pm} - M_{oe}(M_{ee}^{\pm})^{-1} M_{eo})^{-1} \varphi_o .$$

Therefore the only inversion that has to be performed numerically is the one to generate ψ_o from φ_o and this inversion involves only an operator that is better conditioned than the original fermion operator.

Even/odd preconditioning can also be used for the mass non-degenerate Dirac operator D_h eq. (5). The corresponding equations follow immediately from the previous discussion and the definition from eq. (29).

C.2.1 Inverting M on ϕ_o

In case inverting the full matrix M is much faster than inverting the even/odd preconditioned matrix – as might be the case with deflation, one may use for symmetric even/odd preconditioning

$$(\hat{M}^{\pm})^{-1} \phi_o \;=\; P_{l \to o} \; (M_{\pm})^{-1} \; P_{o \to l} \; M_{oo}^{\pm} \; \phi_o \tag{56}$$

Where $P_{l\to o}$ projects the odd sides of a full spinor and $P_{o\to l}$ reverses this by filling up with zeros. $M_\pm$ is here just $\gamma_5 Q_\pm$. For asymmetric even/odd preconditioning the formula reads

$$(\hat{M}^\pm)^{-1}\phi_o \;=\; P_{l\to o}\;(M_\pm)^{-1}\;P_{o\to l}\;\phi_o\,. \tag{57}$$

It is based on the observation that

$$M^{-1} = \begin{pmatrix} A_{ee} & A_{eo} \\ A_{oe} & A_{oo} \end{pmatrix}$$

with (skipping the $\pm$ index for brevity)

$$\begin{aligned} A_{ee} &= (1 - M_{ee}^{-1}M_{eo}M_{oo}^{-1}M_{oe})^{-1}\;M_{ee}^{-1} \\ A_{eo} &= -M_{ee}^{-1}\;M_{eo}\;A_{oo} \\ A_{oe} &= -M_{oo}^{-1}\;M_{oe}\;A_{ee} \\ A_{oo} &= (1 - M_{oo}^{-1}M_{oe}M_{ee}^{-1}M_{eo})^{-1}\;M_{oo}^{-1} \end{aligned}$$

In practice The projectors $P_{l\to o}$ and $P_{o\to l}$ are trivially implemented by inverting the full matrix on a spinor with all even sites set to zero and the odd sites to ϕ_o.

Using this allows one to use one the one hand the speeding up due to even/odd preconditioning in the HMC, and on the other hand the speeding up due to a deflated solver.

C.3 Hasenbusch trick for dynamical tmQCD

We shall now discuss the the trick presented in [35] (mass preconditioning) for dynamical twisted mass lattice QCD. Let $\hat{Q}_\pm$ and $\hat{W}_\pm$ be two matrices as defined in (17) with two parameters μ_1 and μ_2, respectively. The idea is to choose μ_2 bigger than μ_1. With this we can write

$$\det[\hat{Q}_+\hat{Q}_-] = \det[\hat{W}_+\hat{W}_-]\cdot\det[\hat{W}_+^{-1}\hat{Q}_+\hat{Q}_-\hat{W}_-^{-1}]. \tag{58}$$

The first term on the right hand side of (58) can be handled as described in the previous section. The second term needs some further investigation: we again write the determinant as an integral over pseudo fermion fields:

$$\begin{aligned} \det[\hat{W}_+^{-1}\hat{Q}_+\hat{Q}_-\hat{W}_-^{-1}] &\propto \int D[\phi_o]D[\phi_o^\dagger]\exp(-\phi_o^\dagger(\hat{W}_+^{-1}\hat{Q}_+\hat{Q}_-\hat{W}_-^{-1})^{-1}\phi_o) \\ &= \int D[\phi_o]D[\phi_o^\dagger]\exp(-\phi_o^\dagger\hat{W}_-\hat{Q}_-^{-1}\hat{Q}_+^{-1}\hat{W}_+\phi_o) \\ &= \int D[\phi_o]D[\phi_o^\dagger]\exp(-S_{F_2}) \end{aligned} \tag{59}$$

The variation of S_{F_2}, needed for the HMC, then reads as follows:

$$\begin{aligned} \delta S_{F_2} = {}& \phi_o^\dagger[\delta\hat{W}_-(\hat{Q}_+\hat{Q}_-)^{-1}\hat{W}_+ + \hat{W}_-(\hat{Q}_+\hat{Q}_-)^{-1}\delta\hat{W}_+]\phi_o \\ & - \phi_o^\dagger[\hat{W}_-\hat{Q}_-^{-1}\delta\hat{Q}_-(\hat{Q}_+\hat{Q}_-)^{-1}\hat{W}_+ + \hat{W}_-(\hat{Q}_+\hat{Q}_-)^{-1}\delta\hat{Q}_+\hat{Q}_+^{-1}\hat{W}_+]\phi_o \end{aligned} \tag{60}$$

If we define now

$$X_W = (\hat{Q}_+\hat{Q}_-)^{-1}\hat{W}_+\phi_o\,,\quad Y_W = \hat{Q}_+^{-1}\hat{W}_+\phi_o = \hat{Q}_-X_W\,, \tag{61}$$

we can rewrite (60):

$$\begin{aligned}\delta S_{F_2} = \phi_o^\dagger \delta\hat{W}_- X_W + X_W^\dagger \delta\hat{W}_+ \phi_o \\ - Y_W^\dagger \delta\hat{Q}_- X_W - X_W^\dagger \delta\hat{Q}_+ Y_W \,.\end{aligned} \tag{62}$$

Recalling the variation of $\hat{Q}_\pm$ (and of $\hat{W}_\pm$):

$$\begin{aligned}\delta\hat{Q}_\pm &= \gamma_5 \left(-\delta M_{oe}(1 \pm i\mu_1\gamma_5)^{-1} M_{eo} - M_{oe}(1 \pm i\mu_1\gamma_5)^{-1}\delta M_{eo}\right) \,, \\ \delta\hat{W}_\pm &= \gamma_5 \left(-\delta M_{oe}(1 \pm i\mu_2\gamma_5)^{-1} M_{eo} - M_{oe}(1 \pm i\mu_2\gamma_5)^{-1}\delta M_{eo}\right) \,,\end{aligned} \tag{63}$$

we find:

$$\begin{aligned}\delta S_{F_2} &= Y_2^\dagger \delta Q X_2 + X_2^\dagger \delta Q Y_2 - X_1^\dagger \delta Q Y_1 - Y_1^\dagger \delta Q X_1 \\ &= 2\,\mathrm{Re}\left[Y_2^\dagger \delta Q X_2 - Y_1^\dagger \delta Q X_1\right] \,,\end{aligned} \tag{64}$$

where the fields $X_{1,2}$, $Y_{1,2}$ and the matrix δQ are now defined over the full lattice as follows:

$$\begin{aligned}Y_1 = \begin{pmatrix} -(1 + i\mu_1\gamma_5)^{-1} M_{eo} Y_W \\ Y_W \end{pmatrix} \,, \quad Y_2 = \begin{pmatrix} -(1 + i\mu_2\gamma_5)^{-1} M_{eo}\phi_o \\ \phi_o \end{pmatrix} \,, \\ X_{1,2} = \begin{pmatrix} -(1 - i\mu_{1,2}\gamma_5)^{-1} M_{eo} X_W \\ X_W \end{pmatrix} \,, \quad \delta Q = \gamma_5 \begin{pmatrix} 0 & \delta M_{eo} \\ \delta M_{oe} & 0 \end{pmatrix} \,.\end{aligned} \tag{65}$$

The bosonic part is again quadratic in the fields ϕ_o and can be therefore generated at the beginning of each molecular dynamics trajectory with:

$$\phi_o = \hat{W}_+^{-1} \hat{Q}_+ R \tag{66}$$

where R is again a random spinor field taken from a Gaussian distribution with norm one.

This can again be used also with symmetrical even/odd preconditioning by re-defining $Y_{1,2}$ and $X_{1,2}$

$$\begin{aligned}Y_1 &= \begin{pmatrix} -(1 + i\mu_1\gamma_5)^{-1} M_{eo} (1 + i\mu_1\gamma_5)^{-1} Y_W \\ Y_W \end{pmatrix} \\ Y_2 &= \begin{pmatrix} -(1 + i\mu_2\gamma_5)^{-1} M_{eo} (1 + i\mu_2\gamma_5)^{-1} \phi_o \\ \phi_o \end{pmatrix} \,, \\ X_{1,2} &= \begin{pmatrix} -(1 - i\mu_{1,2}\gamma_5)^{-1} M_{eo} (1 - i\mu_{1,2}\gamma_5)^{-1} X_W \\ X_W \end{pmatrix} \,.\end{aligned} \tag{67}$$

C.3.1 Hasenbusch-Trick and Twisted-Clover

In order to avoid to recompute $(1 \pm i\mu\gamma_5 + T_{ee}(x))^{-1}$ too often, the following version of mass preconditioning – which is close to the original paper [35] – might be best suited for Twisted-Clover: define

$$\hat{W}_\pm = \hat{Q}_\pm \pm i\rho = \gamma_5(\hat{M}_\pm \pm i\rho\gamma_5) \tag{68}$$

with a real mass-shift ρ. Here $\hat{Q}_\pm$ is now the even/odd preconditioned clover operator eq. (33)

$$\hat{Q}_\pm = \gamma_5((1 + T_{oo} \pm i\tilde{\mu}\gamma_5) - M_{oe}(1 + T_{ee} \pm i\tilde{\mu}\gamma_5)^{-1} M_{eo}) \,.$$

Then we have $\hat{W}_+ = \hat{W}_-^\dagger$ and $\delta\hat{Q}_\pm = \delta\hat{W}_\pm$. The latter is given by eq. (38)

$$\begin{aligned}\delta\hat{Q}_\pm = \gamma_5 \big(&\delta T_{oo} - \delta M_{oe}(M_{ee}^\pm)^{-1}M_{eo} - M_{oe}(M_{ee}^\pm)^{-1}\delta M_{eo} \\ &+ M_{oe}(M_{ee}^\pm)^{-1}\delta T_{ee}(M_{ee}^\pm)^{-1}M_{eo}\big)\,,\end{aligned}$$

which is in particular independent of ρ. The pseudo-fermion action of a determinant ratio is then given by

$$S_{\rm PF} = \phi^\dagger \hat{W}_-(\hat{Q}_+\hat{Q}_-)^{-1}\hat{W}_+\phi$$

and the variation of $S_{\rm PF}$ is again given by eq. (60). We also define again X_W and Y_W as in eq. (61)

$$X_W = (\hat{Q}_+\hat{Q}_-)^{-1}\hat{W}_+\phi_o\,, \quad Y_W = \hat{Q}_+^{-1}\hat{W}_+\phi_o = \hat{Q}_-X_W$$

With this definition the variation of $S_{\rm PF}$ reads

$$\begin{aligned}\delta S_{\rm PF} &= \phi^\dagger\delta\hat{Q}_-X_W + X_W^\dagger\delta\hat{Q}_+\phi - Y_W^\dagger\delta\hat{Q}_-X_W - X_W^\dagger\delta\hat{Q}_+Y_W \\ &= (\phi - Y_W)^\dagger\delta\hat{Q}_-X_W + X_W^\dagger\delta\hat{Q}_+(\phi - Y_W) \\ &= 2\,\mathrm{Re}[(\phi - Y_W)^\dagger\delta\hat{Q}_-X_W]\,.\end{aligned} \tag{69}$$

Defining analogously to eq. (65) two full vectors X, Y as follows

$$\begin{aligned}Y &= \begin{pmatrix} -(1 + i\mu\gamma_5 + T_{ee}(x))^{-1}M_{eo}(\phi - Y_W) \\ (\phi - Y_W)\end{pmatrix}\,, \\ X &= \begin{pmatrix} -(1 - i\mu\gamma_5 + T_{ee}(x))^{-1}M_{eo}X_W \\ X_W\end{pmatrix}\,,\end{aligned} \tag{70}$$

we get

$$\delta S_{\rm PF} = 2\,\mathrm{Re}[Y^\dagger\,\delta Q\,X]\,, \tag{71}$$

where again

$$\delta Q = \gamma_5\begin{pmatrix}\delta T_{ee} & \delta M_{eo} \\ \delta M_{oe} & \delta T_{oo}\end{pmatrix}\,.$$

C.4 Rational HMC

For the heavy doublet one may alternatively use a rational approximation

$$\mathcal{R}(\hat{Q}_h^2) \;=\; \prod_{i=1}^{N}\frac{\hat{Q}_h^2 + a_{2i-1}}{\hat{Q}_h^2 + a_{2i}} \approx \quad \frac{1}{\sqrt{\hat{Q}_h^2}}$$

where we used the shorthand notation

$$\hat{Q}_h^2 \;=\; \gamma_5\hat{D}_h\tau^1\gamma_5\hat{D}_h\tau^1$$

and $\hat{Q}_h = \gamma_5\hat{D}_h\tau^1$ is the even/odd preconditioned version of Q_h defined in Eq. (5). Obviously, we have $\hat{Q}_h^\dagger = \hat{Q}_h$. We are using the Zolotarev solution for the optimal rational approximation to $1/\sqrt{y}$. The coefficients a_i fulfill the property

$$a_1 > a_2 > ... > a_{2N} > 0\,.$$

We use the partial fraction expansion to re-express

$$\mathcal{R}(\hat{Q}_h^2) \;=\; 1 + \sum_{i=1}^{N} \frac{q_i}{\hat{Q}_h^2 + \mu_i^2}\,.$$

The coefficients r_i are given as

$$q_i = (a_{2i-1} - a_{2i}) \prod_{m=1, m\neq i}^{N} \frac{a_{2m-1} - a_{2i}}{a_{2m} - a_{2i}}\,, \quad i = 1, ..., N\,.$$

If we defined – following Lüscher – $\mu_i = \sqrt{a_{2i}}$ and $\nu_i = \sqrt{a_{2i-1}}$, we may rewrite q_i as

$$q_i = (\nu_i^2 - \mu_i^2) \prod_{m=1, m\neq i}^{N} \frac{\nu_m^2 - \mu_i^2}{\mu_m^2 - \mu_i^2}\,, \quad i = 1, ..., N\,.$$

For the heatbath step we need to generate pseudo-fermion fields from Gaussian random fields R

$$R^\dagger R = \phi^\dagger \mathcal{R} \phi$$

and, therefore, we need operators $C^\dagger, C$ with

$$\mathcal{R}^{-1} = C^\dagger \;\cdot\; C\,, \qquad \phi = C \cdot R\,.$$

C is given by (inspired by twisted mass)

$$C \;=\; \prod_{i=1}^{N} \frac{\hat{Q}_h + i\mu_i}{\hat{Q}_h + i\nu_i}$$

which can again be written as a partial fraction

$$C \;=\; 1 + i \sum_{i=1}^{N} \frac{r_i}{\hat{Q}_h + i\nu_i}\,,$$

with

$$r_i = (\mu_i - \nu_i) \prod_{m=1, m\neq i}^{N} \frac{\mu_m - \nu_i}{\nu_m - \nu_i}\,, \quad i = 1, ..., N\,.$$

The rational approximation $\mathcal{R}$ can be applied to a vector using a multi-mass solver and the partial fraction representation. The same works for C: after solving N equations simultaneously for $(\hat{Q}_h^2 + \nu_i^2)^{-1}$, $\quad i = 1, ..., N$, we have to multiply every term with $(\hat{Q}_h - i\nu_i)$. The hermitian conjugate of C is given by

$$C^\dagger \;=\; 1 - i \sum_{i=1}^{N} \frac{r_i}{\hat{Q}_h - i\nu_i}\,,$$

using $\hat{Q}_h^\dagger = \hat{Q}_h$.

For the acceptance step one just needs an application of $\mathcal{R}$.

C.4.1 Force Computation

For the derivative and the force computation we have to consider terms of the form

$$\phi^\dagger \frac{q_i}{\hat{Q}_h^2 + \mu_i^2} \phi \,,$$

and its variation with respect to the gauge fields:

$$\begin{aligned}\delta_U \ \phi^\dagger \frac{q_i}{\hat{Q}_h^2 + \mu_i^2} \phi &= q_i \phi^\dagger \frac{1}{\hat{Q}_h + i\mu_i} \frac{1}{\hat{Q}_h - i\mu_i} (-\delta_U \hat{Q}_h) \frac{1}{\hat{Q}_h - i\mu_i} \phi \ + \ \text{h.c.} \\ &= -2\,\text{Re}\left(q_i \phi^\dagger \frac{1}{\hat{Q}_h^2 + \mu_i^2} (\delta_U \hat{Q}_h) \frac{1}{\hat{Q}_h - i\mu_i} \phi \right)\end{aligned}$$

C.4.2 Splitting of the Rational

For preconditioning the fermion determinant it is useful to split the rational into several products

$$\mathcal{R}(\hat{Q}_h^2) \ = r_0^l(\hat{Q}_h^2) \cdot r_l^k(\hat{Q}_h^2) \cdot ...$$

with terms

$$r_{c_0}^{c_1} = \ \prod_{i=c_0}^{c_1} \frac{\hat{Q}_h^2 + a_{2i-1}}{\hat{Q}_h^2 + a_{2i}} \,.$$

Every term $r_{c_0}^{c_1}$ can then again be written as a partial fraction with the same coefficients as given above. In Ref. [36] it was shown that the different partial fractions contribute quite differently in their magnitude of the corresponding force to the MD evolution: the smallest shifts and, therefore, most expensive ones contribute the least to the force. Hence, those can be integrated on a larger timescale than the larger shifts, which contribute significantly more to the total MD force.

C.4.3 Correction Monomial

The rational approximation has a finite precision. In the HMC one can account for this effect by estimating

$$1 - |\hat{Q}_h| \mathcal{R} \,,$$

which can be done in different ways:

- we include an additional monomial for

  $$\det(|\hat{Q}_h| \mathcal{R})$$

  in the Hamiltonian. If the rational apprximation is precise enough, it is sufficient to only include this in the heatbath and acceptance step and ignore the contribution to the derivative. For generating the pseudo-fermion field for this monomial, one needs to find

  $$B \cdot B^\dagger = |\hat{Q}_h| \mathcal{R},$$

  which, following Ref. [37], can be expanded in terms of

  $$Z = \hat{Q}_h^2 \mathcal{R}^2 - 1 \,.$$

The series

$$B = (1+Z)^{1/4} = 1 + \frac{1}{4}Z - \frac{3}{32}Z^2 + \frac{7}{128}Z^3 + ...$$

is rapidly converging and can usually be truncated after the Z^2 or latest Z^3 term, see Refs. [37, 38]. We then obtain the pseudo-fermion field ϕ by

$$\phi = B \cdot R\,,$$

where R is again a random Gaussian field. For the acceptance step one needs to compute

$$\phi^\dagger (|\hat{Q}_h|\mathcal{R})^{-1}\phi\,,$$

which, again expanding in Z is obtained by

$$\phi^\dagger (1+Z)^{-1/2}\phi = \phi^\dagger (1 - \frac{1}{2}Z + \frac{3}{8}Z^2 - \frac{5}{16}Z^3 + ...)\phi\,.$$

Also here the series can be truncated after the first few terms. Since the correction monomial is not used in the force computation of MD, its final purpose for the HMC is to compute the energy difference

$$dH_{corr} = R^\dagger \left(1 - (1+Z_{old})^{1/4}(1+Z_{new})^{-1/2}(1+Z_{old})^{1/4}\right) R\,.$$

Considering $\mathcal{O}(Z_{old}) = \mathcal{O}(Z_{new}) = \mathcal{O}(Z)$ and using the previous series expansions, we obtain

$$\begin{aligned} dH_{corr} = {} & R^\dagger \left(\frac{1}{2}Z_{old} - \frac{1}{2}Z_{new}\right) R \\ & + R^\dagger \left(-\frac{1}{8}Z_{old}^2 - \frac{1}{8}\{Z_{old}, Z_{new}\} + \frac{3}{8}Z_{new}^2\right) R \\ & + R^\dagger \left(\frac{1}{16}Z_{old}^3 + \frac{3}{64}\{Z_{old}^2, Z_{new}\} - \frac{1}{32}Z_{old}Z_{new}Z_{old} + \frac{3}{32}\{Z_{old}, Z_{new}^2\} - \frac{5}{16}Z_{new}^3\right) R \\ & + \mathcal{O}(Z^4). \end{aligned}$$

The coefficients in front of the terms $R^\dagger Z_{old}^n R$ are given by the series of

$$(1+Z_{old})^{1/2} - 1 = \frac{1}{2}Z_{old} - \frac{1}{8}Z_{old}^2 + \frac{1}{16}Z_{old}^3 + ...$$

For this reason, computing $\phi = B(Z_{old}) \cdot R$, we use as a stopping criterium

$$c_n R^\dagger Z_{old}^n R < \text{tolerance}$$

where c_n are the coefficients from the series of $(1+Z_{old})^{1/2}$. Since Z is hermitian, we can compute in advance the next order correction of the series evaluating

$$c_n (R Z_{old})^\dagger \cdot (Z_{old}^{n-1} R) < \text{tolerance}\,;$$

in this way we save an application of Z in the evaluation of $\phi = B(Z_{old}) \cdot R$.

Exploting the hermiticity of Z, we can also save applications of it in the computation of

$$dH_{corr} = R^\dagger R - \phi^\dagger \left((1+Z_{new})^{-1/2}\right)\phi\,,$$

which is done in the acceptance step. Indeed defining $\chi_i = Z_{new}^i \phi$, dH_{corr} reads as

$$dH_{corr} = R^\dagger R - \phi^\dagger \phi + \frac{1}{2}\phi^\dagger \chi_1 \phi - \frac{3}{8}\chi_1^\dagger \chi_1 + \frac{5}{16}\chi_1^\dagger \chi_2 - \dots ,$$

that requires n applications of Z_{new} for computing dH_{corr} up to $\mathcal{O}(Z_{new}^{2n})$. Here we use as stopping criterium,

$$c_n \phi^\dagger Z_{new}^n \phi < \text{tolerance} ;$$

where c_n are the coefficients from the series of $(1 + Z_{new})^{-1/2}$.

- the second possibility is to include this correction as a reweighting factor.
- the third is to use a more precise rational approximation for the heatbath and acceptance steps.

For evaluating the rational approximation $\mathcal{R}$ applied to a spinor field ψ a multi-mass or multi-shift solver (see algorithm 4) can be used, see Ref. [10] and references therein. However, a little care is needed as the shift vary over several orders of magnitudes.

The original Krylov space is build for the shift smallest in modulus. This will converge slowest and, therefore, the other shifts will have the same or better precision guaranteed. But, if the range in the shifts is too large, one needs to remove the highest shifts in the course of the CG solve before the smallest shift is converged. This will prevent the appearance of double precision underflow and hence the appearance of exact zeros in $\zeta^{k_{\max}}$, which would lead to NaNs in the solution vectors.

In order to avoid to compute the residue for all the shift frequently during the CG-MMS solve, one can rather monitor the norm of the correction vector p^σ of the currently biggest shift σ still in the process. The CG works such that the correction decreases with decreasing residue. Therefore, one can remove the shift σ when

$$\|\alpha^\sigma p^\sigma\| < \delta ,$$

where one could for instance chose $\delta = c \cdot \epsilon$. ϵ is the desired precision of the CGMMS solve and $0 < c \leq 1$ some suitably chosen constant. Removing converged solutions has the side effect of speeding up the CGMMS solve in terms of computing time.

D Deflation

D.1 Implementing Deflation

We are aiming to solve

$$D\psi = \eta$$

using Lüschers deflation method. Note that D is the non-hermitian (twisted) Wilson Dirac operator.

Lets assume we have devided the whole lattice completely into blocks $\Lambda(\vec{b})$ on a four dimensional grid. Every block has a grid coordinate $\vec{b}$ and we have a total number of N_b blocks. (This is actually what we have in the MPI environment) Lets assume we have

found N approximate (global) eigenvectors $\psi_l, l = 1, ..., N$ and we have restricted them to the blocks via

$$\phi_l^{\vec{b}}(x) = \begin{cases} \psi_l(x) & \text{if } x \in \Lambda(\vec{b})\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

obtaining in total $N_b \cdot N$ fields. And we have already orthonormalised them using Gram-Schmidt or whatever.

Construction of the little Dirac operator

The *little Dirac operator* A is then computed from

$$A_{(\vec{a},k)(\vec{b},l)} = \langle \phi_k^{\vec{a}} | D \phi_l^{\vec{b}} \rangle$$

$A_{(\vec{a},k)(\vec{b},l)}$ is non-zero only for $\vec{a} = \vec{b}$ or $\vec{b} = \vec{a} \pm \vec{\mu}, \mu = 1, ..., 4$, where $\vec{\mu}$ is a unit vector in block space, because D involves only next neighbour interaction.

All elements with $\vec{a} = \vec{b}$ can be computed by applying the local Dirac operator $D^{\vec{a}}$ (i.e. all exterior boundaries set to zero, because $\phi_l^{\vec{a}}$ has support only on block $\Lambda(\vec{a})$)

$$\varphi_l^{\vec{a}} = D\phi_l^{\vec{a}} = D^{\vec{a}}\phi_l^{\vec{a}}, \quad l = 1, ..., N$$

and computing the scalar products $(\phi_l^{\vec{a}}, \varphi_k^{\vec{a}})$ for all combination of l, k then. For the terms with $\vec{a} \neq \vec{b}$ we have to be more carefully. Probably it is best done by looping over all directions $\pm\mu$ and computing

$$\langle \phi_l^{\vec{a}} |\ \varphi_k^{\vec{a}+\vec{\mu}} \rangle_{\partial_\mu \Lambda(\vec{a})}$$

where $\partial_\mu \Lambda(\vec{a})$ denotes the inner boundary in μ-direction of block $\Lambda(\vec{a})$, where $\varphi_k^{\vec{a}+\vec{\mu}}$ is non-zero.

The action of the little Dirac operator on a *little quark field* w (complex field of length $N = N_S \cdot N_b$) on block $\Lambda(\vec{a})$ reads:

$$\begin{aligned} v_k^{\vec{a}} &= A_{(\vec{a},k)(\vec{b},l)} w_l^{\vec{b}} \\ &= \sum_{l=1}^{N} A_{(\vec{a},k)(\vec{a},l)} w_l^{\vec{a}} + \sum_{\mu} \sum_{l=1}^{N} [A_{(\vec{a},k)(\vec{a}+\vec{\mu},l)} w_l^{\vec{a}+\vec{\mu}} + A_{(\vec{a},k)(\vec{a}-\vec{\mu},l)} w_l^{\vec{a}-\vec{\mu}}] \end{aligned}$$

or in matrix notation

$$v^{\vec{a}} = A^{\vec{a},\vec{a}} w^{\vec{a}} + \sum_{\mu} [A^{\vec{a},\vec{a}+\vec{\mu}} w^{\vec{a}+\vec{\mu}} + A^{\vec{a},\vec{a}-\vec{\mu}} w^{\vec{a}-\vec{\mu}}]$$

This involves again only next neighbour (block) interaction. Every block matrix $A^{\vec{a},\vec{b}}$ is a $N \times N$ complex matrix.

D.1.1 global mode deflation

We want to use the global fields ψ_l to deflate the little Dirac operator A. This requires to find vectors χ_l, which fulfill

$$B_{kl} = \langle \psi_k | D \psi_l \rangle = \langle u_k | A u_l \rangle \tag{72}$$

for $k, l = 1, ..., N_s$. We recall that the fields ϕ_i, $i = 1, ..., N_b \cdot N_s$ were obtained by restricting the fields ψ_l to the blocks $\vec{a}$ followed by an orthonormalisation process. So the complex vectors $u_l^{\vec{a}}$ of length $N = N_s \cdot N_b$ on block $\vec{a}$ are computed from

$$(u_l^{\vec{a}})_i = \langle \phi_i | \psi_l^{\vec{a}} \rangle, \qquad \forall \phi_i \text{ has support on block } \Lambda(\vec{a}) \tag{73}$$

where the notation $(u^{\vec{a}})_i$ means the i-th component of vector u on block $\vec{a}$. The little little Dirac operator B is then given by

$$\begin{aligned} B_{kl} &= \langle u_k | A u_l \rangle = \langle (u_k)_i | \langle \phi_i | D \phi_j \rangle (u_l)_j \rangle \\ &= \langle \psi_k | \phi_i \rangle \langle \phi_i | D \phi_j \rangle \langle \phi_j | \psi_l \rangle = \langle \psi_k | D \psi_l \rangle \end{aligned}$$

where summing over equal indices is understood. The little Dirac operators is then deflated using the little oblique projectors p_L and p_R:

$$\begin{aligned} p_L v &= v - \sum_{k,l}^{N_s} A u_k (B^{-1})_{kl} \langle u_l | v \rangle \\ p_R v &= v - \sum_{k,l}^{N_s} u_k (B^{-1})_{kl} \langle u_l | A v \rangle \end{aligned}$$

and the same algebra as before.

What needs to be done

- implement a suitable preconditioner
- ...

E Solvers

In this section, we give details of some of the solvers which are implemented in tmLQCD. In particular, we clarify some of the conventions used and how these map over to the external library interfaces.

E.1 CGMMS

The multi-shift CG implementation in tmLQCD is referred to as *CGMMS* since it was originally developped to solve a multi-system of equations of the form

$$(A + \mathbb{I}\mu_k^2) = b\,, \tag{74}$$

where A can be Q_+Q_- or M_+M_- and the squared shifts μ_i^2 can be naturally interpreted as different twisted quark masses (in the case of M_+M_-, appropriate factors of γ^5 must be inserted as required).

$$\begin{aligned} (M_{\mathrm{w}} + i\mu\gamma^5)(M_{\mathrm{w}}^\dagger - i\mu\gamma^5) &= b \\ (M_{\mathrm{w}} M_{\mathrm{w}}^\dagger + \cancel{i\mu\gamma^5 M_{\mathrm{w}}^\dagger - i\mu M_{\mathrm{w}}\gamma^5} + \mu^2) &= b \\ (M_{\mathrm{w}} M_{\mathrm{w}}^\dagger + \mu^2) &= b\,, \end{aligned} \tag{75}$$

where in the last line γ_5-hermiticity of M_w was used. With the clover term, T, in the operator, the calculation goes through in the same way, with the result

$$\begin{aligned}&(M_\mathrm{w}M_\mathrm{w}^\dagger + M_\mathrm{w}T + TM_\mathrm{w}^\dagger + \cancel{i\mu\gamma^5 M_\mathrm{w}^\dagger - i\mu M_\mathrm{w}\gamma^5} + i\mu\gamma_5 T - i\mu T\gamma_5 + T^2 + \mu^2) = b\\ &(M_\mathrm{sw}M_\mathrm{sw}^\dagger + \mu^2) = b\,,\end{aligned} \tag{76}$$

where $M_\mathrm{sw} = M_\mathrm{w} + T$.

The algorithm is listed below in Algorithm 4 (see also Ref. [10] and references therein), with the identification $\sigma_k = \mu_k^2$. Note that in line 6 below, $\alpha_{n-1}(1+\sigma_k\alpha_n)$ is correct, in contrast to Ref. [10].

Algorithm 4 CGMMS algorithm

1: $n = 0, x_0^k = 0, r_0 = p_0 = p_0^k = b, k_\mathrm{max}, \delta, \epsilon$
2: $\alpha_{-1} = \zeta_{-1}^k = \zeta_0^k = 1, \beta_0^k = \beta_0 = 0$
3: **repeat**
4: $\quad\alpha_n = (r_n, r_n)/(p_n, Ap_n)$
5: $\quad$**for** $k = 1$ to k_max **do**
6: $\quad\quad\zeta_{n+1}^k = (\zeta_n^k\alpha_{n-1})/(\alpha_n\beta_n(1-\zeta_n^k/\zeta_{n-1}^k) + \alpha_{n-1}(1+\sigma_k\alpha_n))$
7: $\quad\quad\alpha_n^k = (\alpha_n\zeta_{n+1}^k)/\zeta_n^k$
8: $\quad\quad x_{n+1}^k = x_n^k + \alpha_n^k p_n^k$
9: $\quad\quad$**if** $\|\alpha^{k_\mathrm{max}}p^{k_\mathrm{max}}\| < \delta$ **then**
10: $\quad\quad\quad k_\mathrm{max} = k_\mathrm{max} - 1$
11: $\quad\quad$**end if**
12: $\quad$**end for**
13: $\quad x_{n+1} = x_n + \alpha_n p_n$
14: $\quad r_{n+1} = r_n - \alpha_n Ap_n$
15: $\quad\beta_{n+1} = (r_{n+1}, r_{n+1})/(r_n, r_n)$
16: $\quad\beta_{n+1}^k = \frac{\beta_{n+1}\zeta_{n+1}^k\alpha_n^k}{\zeta_n^k\alpha_n}$
17: $\quad p_{n+1}^k = \zeta_{n+1}^k r_{n+1} + \beta_{n+1}^k p_n^k$
18: $\quad n = n + 1$
19: **until** $\|r_n\| < \epsilon$

The implementations in `solver/cg_mms_tm.c` and `solver/cg_mms_tm_nd.c` use a slightly different approach in that the lowest shift is included in the operator A, such that the higher shifts are $\sigma_k - \sigma_0\,\forall k > 0$.

It should be noted that $\sqrt{\sigma_k}$ are passed to `cg_mms_tm` and the solver internally squares these and shifts them by σ_0.

E.1.1 Single flavour Wilson (clover) fermions in the rational approximation

For details about the rational approximation in tmLQCD, see Section C.4.

QPhiX interface: For the HMC with a single flavour of Wilson (clover) fermions (`RAT` or `CLOVERRAT` monomials), the function `solve_mshift_oneflavour` of `solver/monomial_solve.c` provides a wrapper for tmLQCD or external multi-shift solvers.

Note that it passes the shifts as expected by tmLQCD's `cg_mms_tm`, which means that they need to be squared. For the QPhiX normalisation, the QPhiX solver interface also divides them by $4\kappa^2$. The shifts are taken as is and not shifted by σ_0.

E.1.2 Two flavour Wilson twisted mass (clover) fermions in the rational approximation

QPhiX interface: For the HMC with non-degenerate twisted mass (clover) doublets (`NDRAT` and `NDCLOVERRAT` monomials, exactly the same approach is used.