Todos *[2015-01-19 Mon]*

Pascal Huber

February 16, 2015

Contents

1 DONE *<2015-01-19 Mon>* Write extensive documentation about the Hessians

- CLOSING NOTE *[2015-02-16 Mon 13:49]*
  Wrote an overview of all code related to the computation of Hessians.

1.1 Hessian data structures *<2015-02-13 Fri>*

The Hessians data is stored in form of local Hessians (matrices of size NDIMMAT).

1.1.1 Data structures for storing Hessians:

In `struct Particle` (see data.h):

`double *localHessians` Pointer to an array of length `NDIMMAT x neighbors` which stores the entries of the local Hessians. For every neighbor Particle the Hessian $\partial_{\mathrm{p}}\ \partial_{\mathrm{q}}$ is stored, where p is the Particle represented by the struct. All entries are stored in a rowwise manner: $\partial_{\mathrm{p1}}\partial_{\mathrm{q1}}\ \partial_{\mathrm{p1}}\partial_{\mathrm{q2}}$ $\partial\{\mathrm{p1}\}\partial\{\mathrm{q3}\}\ \partial_{\mathrm{p2}}\partial\{\mathrm{q1}\}\ \partial_{\mathrm{p2}}\partial\{\mathrm{q2}\}\ \ldots$

`trx_htab *hessianIndex` Pointer to a hash table which maps particle indices to the corresponding array entry in `localHessians`. This allows quick retrieval of local Hessians for each pair of Particle structs.

`unsigned int sizeOfLocalHessians` Stores the maximal number of entries (not Hessians!) that can be stored by the `localHessian` array. This is important for cleaning of the data structures.

In `struct Problem` (see data.h):

`int computeHessians` Flag that indicates if Hessians are computed (=1) in the simulation or not (=0).

1.1.2 Function for data structure handling:

In order to work with the Hessian data structures some "comfort functions" are implemented in `particle.c`:

`void createLocalHessians(struct Particle *p, int numberOfHessians)` Allocates memory for the `localHessians` array and creates the `hessianIndex` hash table.

`void destroyLocalHessians(struct Particle *p)` Frees memory for all entries of `localHessians` and destroys the `hessianIndex` hash table.

`int getLocalHessian(struct Particle *p, unsigned int qIndex, double *values)` Gets the whole local Hessian $\partial_{\mathrm{p}\partial\mathrm{q}}$ and stores it in `values`.

`double getLocalHessianComponent(struct Particle *p, unsigned int qIndex, unsigned int i` Returns the (i,j)-th entry of the local Hessian $\partial_{\mathrm{p}\partial\mathrm{q}}$.

`void cleanHashTable(trx_htab *table)` Frees the memory of all entries in `table`. The hash table itself is not destroyed. This is used to reinitialize `hessianIndex`.

`void cleanLocalHessians(struct Particle *p)` Reinitializes the data structure for the Particle struct p.

`void addLocalHessian(struct Particle *p, unsigned int qIndex, double *values)` Adds a new local Hessian matrix to Particle p.

1.1.3 Initialization of Hessian data structures:

The Hessian data structures are initialized during the creation of the Particle structs. A short overview of the overall structure of the initialization:

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.
2. (tremolo.c): `static void Run(int argc, char **argv)` calls `void Init(struct Problem *P)` for the initialization of the main data structures.
3. (init.c): `void Init(struct Problem *P)` calls `void InitSimBox(struct Problem *P)` to add the Particle structs to the simulation.
4. (particle.c) `void InitSimBox(struct Problem *P)` calls `int ReadParticles(struct Problem *P, FILE *f, struct ParseInfoBlock *PIB)` which adds Particle structs according to the input file `*f`
5. (particle.c) `int ReadParticles(struct Problem *P, FILE *f, struct ParseInfoBlock *PIB)` creates all Particle structs for the simulation using `struct Particle *CreateParticleNoSpeStr(const struct Problem *P)` which allocates memory for a Particle struct and initializes some member variables.

The actual initialization is done in `struct Particle *CreateParticleNoSpeStr(const struct Problem *P)`. For this, the `computeHessian` flag of the Problem struct is checked. According to the flag memory for the Hessian data structures is allocated or not. The size of the `localHessian` array is roughly estimated using the total number of Particles and the number of cells in the simulation.

1.1.4 Deletion of Hessian data structures:

The Hessian data structures are destroyed whenever the associated Particle struct is deleted. This is done especially at the end of the program:

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.

2. (tremolo.c): `static void Run(int argc, char **argv)` calls `void RemoveEverything(struct Problem *P)` at the end of the program to free all allocated memory.

3. (helpers.c): `void RemoveEverythin(struct Problem *P)` calls `void DeleteLists(struct Problem *P)` to delete the linked cell structure.

4. (helpers.c): `void DeleteLists(struct Problems *P)` calls `void DeleteAllParticles(struct Problem *P, struct LCStructData *LCS)` in order to delete all Particle structs.

5. (helpers.c): `void DeleteAllParticles(struct Problem *P, struct LCStructData *LCS)` calls `void DeleteLCListRec()` which in turn calls `void DeleteParticle(const struct Problem * UNUSED(P), struct Particle *p)`.

The actual deletion of the Hessian data structures is done in `void DeleteParticle(const struct Problem * UNUSED(P), struct Particle *p)` (particle.c). For this, a NULL-check for the `localHessian` array is performed and then the memory of the Hessian data structures is freed.

1.2 Data structures for Hessian calculation *<2015-02-16 Mon>*

For the computation of Hessians every potential provides a function used for the calculation. These functions are stored in lists of structs which store the function pointers. Tremolo iterates at every time step over these lists and calls the function pointers. The struct used for Hessian calculation is defined in lcforces.h

`struct LCHessianList` Struct similar to `LCForceList` storing the function pointers and data necessary for the computation of Hessians. Every struct represents a given potential and a given pair of particles.

The list of `LCHessianList` structs is stored in the `LCForceParams` struct:

`LCForceParams` is stored in the `Problem` struct (for different "stages").

`LCForceData` Every `LCForceParams` struct stores an array of `LCForceData` structs. Every `LCForceData` struct represents a pair of particles.

`LCHessianList` For every potential the `LCForceData` struct points to a `LCForceList` and to a `LCHessianList` struct which in turn store the function pointer for the calculation.

1.3 Registration of Hessian computation *<2015-02-16 Mon>*

For the computation of Hessians every potential must provide a function `Calc<Potentialname>Hessian` (cf. e.g. `CalcLennardJonesHessian()`). This function must be registered such that tremolo can call it during the simulation. Since the potentials file is parsed before the parameter file the registration of the Hessian calculation function is done in two steps:

1. For all potentials in the potentials file (that provide Hessian calculation functions) the corresponding Hessian calculation function is registered regardless if Hessians are computed during the simulation.

2. If in the parameter file Hessian computation is disabled, all registered Hessian calculation functions are unregistered.

The registration process is done in the function `void ReadParameters(struct Problem * const P, const char *const filename)`.

1.3.1 Set default behaviour for Hessian computation.

If no tag "hessians" is provided in the parameters file then the Hessian computation is disabled:

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.

2. (tremolo.c): `static void Run(int argc, char **argv)` calls `void ReadParameters(struct Problem *const P, const char *const filename)` in order to parse the input files.

3. (init.c): `void ReadParameter(struct Problem *const P, const char *const filename)` calls the function `void FirstInit(struct Problem *const P)`.

4. (init.c): `void FirstInit(struct Problem *const P)` sets `P->computeHessians = 0`: by default no Hessians are computed.

1.3.2 Registration of Hessian calculation functions

The actual registration of the Hessian calculation functions is done in the function `int ParsePotentialFiles(struct Problem *P)` (parse.c). The registration is done by each potential itself. For this the potential must implement the Hessian registration in the corresponding `Store<Potentialname>Data()` function (c.f. `StoreLennardJonesData()` (twobody.c)).

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.

2. (tremolo.c): `static void Run(int argc, char **argv)` calls `void ReadParameters(struct Problem *const P, const char *const filename)` in order to parse the input files.

3. (init.c): `void ReadParameter(struct Problem *const P, const char *const filename)` calls `int ParsePotentialFiles(struct Problem *P)` which registers and calls for all potential the corresponding registration routines.

For the Lennard-Jones potential the registration is done in the following steps:

1. (init.c): `int ParsePotentialFiles(struct Problem *P)` calls `int Read2BodyPotentials(struct Problem *P, FilePosType *filePos, parse_data *pd)` which then in turn calls `int ReadLennardJonesData()`.

2. (twobody.c): `int ReadLennardJonesData()` calls `int StoreLennardJonesData()` which performs the registration of the function `static void CalcLennardJonesHessian()` using the function `int RegisterLCHessian()` (implemented in lcforces.c).

1.3.3 Check and possible deregistration of Hessian calculation functions

After parsing the potentials file and the registration of the Hessian calculation functions, the parameter file is parsed in `int ParseParameterFiles(struct Problem *P)`. If the parameter file contains a `hessians` tag in the `analyze` block the `int StoreAnalyzeHessians()` function (in groupmeas.c) is called which sets the `computeHessians` flag of the Problem struct.

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.

2. (tremolo.c): `static void Run(int argc, char **argv)` calls `void ReadParameters(struct Problem *const P, const char *const filename)` in order to parse the input files.

3. (init.c): `void ReadParameter(struct Problem *const P, const char *const filename)` calls `int ParseParameterFiles(struct Problem *P)`.

4. (parse.c): `int ParseParameterFiles(struct Problem *P)` parses the parameter file and calls `int StoreOutputAnalyze(struct Problem *P, FilePosType *filePos, parse_data *pd)`.

5. (generalmeas.c): `int StoreOutputAnalyze()` calls `int StoreAnalyzeHessians(struct Problem *P, FilePosType *filePos, parse_data *pd)` if a `hessians` tag is found in the parameter file. If no `hessians` tag is found the function is not called and the default value (= 0) is used.

6. (groupmeas.c): `int StoreAnalyzeHessians(struct Problem *P, FilePosType *filePos, parse_data *pd)` sets `P->computeHessians` to one or zero according to the value set in the parameter file.

After parsing the parameter file some control function are called within `int ParseParameterFiles(struct Problem *P)`. If no Hessians are supposed to be computed the Hessian calculation functions are here deregistrated:

1. (init.c): `void ReadParameter(struct Problem *const P, const char *const filename)` calls `int ParseParameterFiles(struct Problem *P)`.

2. (parse.c): `int ParseParameterFiles(struct Problem *P)` calls `void ControlParseParameterFiles(struct Problem *P)`.

3. (parse.c): `void ControlParseParameterFiles(struct Problem *P)` calls all `ControlParameterRecord` function pointers of the `ParamInit` array. This also includes the function `int ControlGroupMeasureRecord(struct Problem *P)`.

4. (groupmeas.c): `int ControlGroupMeasureRecord(struct Problem *P)` calls `int ControlHessianRecord(struct Problem *P)`.

5. (groupmeas.c): `int ControlHessianRecord(struct Problem *P)` deregisters all Hessian calculation functions if the `computeHessian`-flag is set to 0.

1.4 Computation and reinitialization of Hessians

The Hessian calculation is done for every time step in the `RunSim()` function:

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.
2. (tremolo.c): `void Run(int argc, char **argv)` calls `static void RunSim(struct Problem *P)` which does all time steps.
3. (tremolo.c): `static void RunSim(struct Problem *P)` calls `void UpdateMeasureVisData(struct Problem *P)` which iterates over all `Particle` structs and reinitializes the `localHessians` arrays.
4. (tremolo.c): `static void RunSim(struct Problem *P)` calls `void MainLCForce(struct Problem *P)` which is the main function for linked cell force calculation.
5. (lcforces.c): `void MainLCForce(struct Problem *P)` calls `void CalcLCForce()` which iterates over all cells.
6. (lcforces.c): `void CalcLCForce()` calls for every cell `static void CalcLCForceForCell()` which iterates over all `Particles` in this cell.
7. (lcforces.c): `static void CalcLCForceForParticle()` iterates for all neighboring particles over all `LCForceList` and `LCHessianList` structs and calls the force/Hessian calculation function.

The calculation of the Hessians itself is done by a function which has to be defined for each potential. A pointer of this function is stored in the corresponding `LCHessianList` struct. An example of such a function `static void CalcLennardJonesHessian(void *data, struct LCForceFunctionData *const LFFD)`: The function computes for the given `Particle` pair (p,q) all local Hessians ∂_{pp}, ∂_{pq}, ∂_{qp}, ∂_{qq} which are then added to the `localHessians` arrays of the corresponding `Particle` structs. The Hessians ∂_{pp} and ∂_{pq} are added to `Particle` p and the other two to q.

1.5 Output of Hessians

The Hessian-file suffix is declared data.h in the `enum OutputFileSuffixes` as `hessiansfile` and then defined in the function `void ReadParameters(struct Problem *const P, const char * const filename)`. The output itself is done in a function `static void OutputFileHessians(struct Problem * P)`:

1. (tremolo.c): `int main()` calls `void Run(int argc, char **argv)` to start the simulation.

2. (tremolo.c): `void Run(int argc, char **argv)` calls `static void RunSim(struct Problem * P)` which does all time steps.

3. (tremolo.c): `static void RunSim(struct Problem * P)` calls `void OutputBeforeUpdate(struct Problem * P)`.

4. (output.c): `void OutputBeforeUpdate(struct Problem * P)` calls `void OutputV(struct Problem * P)`.

5. (output.c): `void OutputV(struct Problem * P)` calls `void OutputVisualData(struct Problem * P)`.

6. (output.c): `void OutputVisualData(struct Problem * P)` calls `static void OutputFileHessians(struct Problem * P)` which opens the .hessians file and writes for all `Particles` the local Hessians to that file.