emDNA: Tetramer Functionality

ZOE WEFERS

MENTOR: WILMA OLSON

Tetramer: How does the math change?

$$E^i = \frac{1}{2}(\underline{p}^i - \underline{p}_0^{\,i})^T \mathbf{F}^i (\underline{p}^i - \underline{p}_0^{\,i}) \qquad \frac{\partial E^i}{\partial \underline{p}^i} = \mathbf{F}_s^i (\underline{p}^i - \underline{p}_0^{\,i})$$

- **Good news: Only p_0 and F changes!**
- In calculations p_0 and F obtained through BpCollection object using the following two methods:
  - **bp_step_intrinsic_parameters**
  - **bp_step_force_constants**
- So if I can make sure that these function return the correct values the rest of the program should work fine

- Recall that bp_step_intrinsic_parameters and bp_step_force_constants are implemented by calling the accessors intrinsic_bp_step_params and force_constants on the StepParameterDB and ForceConstantsDB attributes of the BpCollection object, respectively
- Want to be able to pass tetramers to instrinic_bp_params and force_constants instead of StepSequences (ie dimer object)

```
const BpStepParams
BpCollection::bp_step_intrinsic_parameters(Size step_index) const {

    // StepParameters data
    const VectorN p =
    m_step_seqdep.intrinsic_bp_step_params(m_step_sequences[step_index]).
    inline_vector();

    // convert to BpStepParams
    return BpStepParams(p);
```

Relevant Classes:

- BPCollection is an object that holds lots of information about a sequence of DNA including
  - Vector of BasePairs m_base_pairs
  - Vector of BpStepParams m_bp_step_params
  - Vector of BpStepDofs m_bp_step_dofs
  - Vector of SizePairs m_frozen_steps_domains
  - Vector of StepSequences m_step_sequences
  - StepParametersDB m_step_seqdep
  - ForceConstantsDB m_fmat_seqdep
- A StepParametersDB holds a matrix (originally 4x4) of the intrinsic step parameters for each dimer, and it holds a model name
- A ForceConstantsDB holds a matrix (originally 4x4) of the force field matrices for each dimer, and it holds a model name

What needed to be modified

- Need BP Collection object to hold a sequence of Tetramers analogous to the way it holds a sequence of dimers (ie StepSequences)
  - So need to make "x", imaginary base, for trimers
  - Need to make Tetramer object
  - Need to initialize the sequence of tetramers in BpCollection constructor
- Need to modify StepParametersDB and ForceConstantsDB to be able to hold more data
- Need to make new list TetramerDependence Models
- Need to change force field packager to be able to read both dimeric (16 line) filers and tetrameric files (400 lines)
- Need to change Serialization/Archive classes because StepParametersDB/ForceConstantsDB now bigger

Added "x" imaginary base to deal with trimers

```
/*** Base class ***/

// static methods
std::string Base::str(const BaseSymbol& base) {
    // the following array has to be ordered as the BaseSymbol enum
    const std::string str[4] = {"A", "C", "G", "T"};
    return str[static_cast<Size>(base)];
};
BaseSymbol Base::base_symbol_from_char(const char s) {
    if (s == 'A')
        return BaseSymbol::A;
    else if (s == 'C')
        return BaseSymbol::C;
    else if (s == 'G')
        return BaseSymbol::G;
    else if (s == 'T')
        return BaseSymbol::T;
    else
        DS_ASSERT(false, "wrong base symbol:\n"+std::string(&s));
};
BaseSymbol Base::complementary_base(const BaseSymbol& base) {
    if (base == BaseSymbol::A)
        return BaseSymbol::T;
    else if (base == BaseSymbol::T)
        return BaseSymbol::A;
    else if (base == BaseSymbol::C)
        return BaseSymbol::G;
    else if (base == BaseSymbol::G)
        return BaseSymbol::C;
    else
        DS_ASSERT(false, "wrong base symbol:\n"+str(base));
};
```

```
/*** Base class ***/

// static methods
std::string Base::str(const BaseSymbol& base) {
    // the following array has to be ordered as the BaseSymbol enum
    const std::string str[5] = {"A", "C", "G", "T", "x"};
    return str[static_cast<Size>(base)];
};
BaseSymbol Base::base_symbol_from_char(const char s) {
    if (s == 'A')
        return BaseSymbol::A;
    else if (s == 'C')
        return BaseSymbol::C;
    else if (s == 'G')
        return BaseSymbol::G;
    else if (s == 'T')
        return BaseSymbol::T;
    else if (s == 'x')  // Added by Zoe Wefers (McGill University, June 2021, DIMACS REU)
        return BaseSymbol::x;
    else
        DS_ASSERT(false, "wrong base symbol:\n"+std::string(&s));
};
BaseSymbol Base::complementary_base(const BaseSymbol& base) {
    if (base == BaseSymbol::A)
        return BaseSymbol::T;
    else if (base == BaseSymbol::T)
        return BaseSymbol::A;
    else if (base == BaseSymbol::C)
        return BaseSymbol::G;
    else if (base == BaseSymbol::G)
        return BaseSymbol::C;
    else if (base == BaseSymbol::x) //Added by Zoe Wefers (McGill University, June 2021, DIMACS REU)
        return BaseSymbol::x;
    else
        DS_ASSERT(false, "wrong base symbol:\n"+str(base));
};
```

Added Tetramer Class

```
//Added by Zoe Wefers (McGill University, June 2021, DIMACS REU)
//TetramerSequence class
class TetramerSequence{

public:
    //constructors
    TetramerSequence() = default;
    TetramerSequence(const BaseSymbol& n1, const BaseSymbol& n2, const BaseSymbol& n3, const BaseSymbol& n4);
    TetramerSequence(const TetramerSequence& tetra_sequence) = default;
    TetramerSequence(TetramerSequence&& tetra_sequence) = default;
    ~TetramerSequence() = default;

    // copy and move operators (not sure what these are for)
    TetramerSequence& operator=(const TetramerSequence& tetra_sequence) = default;
    TetramerSequence& operator=(TetramerSequence&& tetra_sequence) = default;

    // base accessors
    const BaseSymbol& first_base() const;
    const BaseSymbol& second_base() const;
    const BaseSymbol& third_base() const;
    const BaseSymbol& fourth_base() const;

private:
    std::tuple<BaseSymbol, BaseSymbol, BaseSymbol, BaseSymbol> m_bases;

};
```

Changes to BpCollection:

```
void BpCollection::set_collection_sequence(const std::string& sequence) {
    m_step_sequences.clear();
    for (Size i=0; i<sequence.size()-1; ++i) {
        m_step_sequences.
        push_back(StepSequence(Base::
                               base_symbol_from_char(sequence[i]),
                               Base::
                               base_symbol_from_char(sequence[i+1])));
    };

    // Added by Zoe Wefers (McGill University, June 2021, DIMACS REU)
    char x_char = 'x';
    m_tetramer_sequences.clear();
    for(Size i=0; i<sequence.size()-1; ++i){
        if (i == 0){
            m_tetramer_sequences.
            push_back(TetramerSequence(Base::base_symbol_from_char(x_char),
                                       Base::base_symbol_from_char(sequence[i]),
                                       Base::base_symbol_from_char(sequence[i+1]),
                                       Base::base_symbol_from_char(sequence[i+2])));
        }
        else if (i == sequence.size()-2) {
            m_tetramer_sequences.
            push_back(TetramerSequence(Base::base_symbol_from_char(sequence[i-1]),
                                       Base::base_symbol_from_char(sequence[i]),
                                       Base::base_symbol_from_char(sequence[i+1]),
                                       Base::base_symbol_from_char(x_char)));
        }
        else {
            m_tetramer_sequences.
            push_back(TetramerSequence(Base::base_symbol_from_char(sequence[i-1]),
                                   Base::base_symbol_from_char(sequence[i]),
                                   Base::base_symbol_from_char(sequence[i+1]),
                                   Base::base_symbol_from_char(sequence[i+2])));
        }
```

```
private:

    // collection data
    std::vector<BasePair> m_base_pairs;
    std::vector<BpStepParams> m_bp_step_params;
    std::vector<BpStepDofs> m_bp_step_dofs;
    std::vector<SizePair> m_frozen_steps_domains;

    // sequence-dependence data
    std::vector<StepSequence> m_step_sequences;
    std::vector<TetramerSequence> m_tetramer_sequences; //added by Zoe Wefers
    StepParametersDB m_step_seqdep;
    ForceConstantsDB m_fmat_seqdep;
```

Added sequence of tetramers as attribute to BPCollection (even if using dimeric model), and added accessor function for it

Changes to BPCollection:

```
const BpStepParams
BpCollection::bp_step_intrinsic_parameters(Size step_index) const {

    // StepParameters data
    const VectorN p =
    m_step_seqdep.intrinsic_bp_step_params(m_tetramer_sequences[step_index]).
    inline_vector(); //Changed by Zoe Wefers (McGill University, June 2021, DIMACS REU)

    // convert to BpStepParams
    return BpStepParams(p);

};
const MatrixN& BpCollection::bp_step_force_constants(Size step_index) const {
    return m_fmat_seqdep.force_constants(m_tetramer_sequences[step_index]); //Changed by Zoe Wefers (McGill University, June 2021, DIMACS REU)
};
```

Click to add text

Now passing Tetramer to intrinsic_bp_step_params/force_constants rather than StepSequence

Changes to StepParametersDB and ForceConstantsDB

```
// sequence-dependent data
StepParameters m_step_parameters[SEQ_DIM+1][SEQ_DIM][SEQ_DIM][SEQ_DIM+1]; //Changed by Zoe Wefers
```

```
// sequence-dependent data
MatrixN m_force_constants[SEQ_DIM+1][SEQ_DIM][SEQ_DIM][SEQ_DIM+1]; //Changed by Zoe Wefers
```

- Now holding 5x4x4x5 tensors instead of 4x4 matrices
- If using dimeric model then the inner 4x4 matrix will be the same for all layers of the tensor
  - ie for all 0 <= i, j < 5, m_step_parameters[i][:][:][j] is the regular 4x4 matrix of intrinsic step parameters vectors
  - So if using dimeric model doesn't matter what flaking base pairs are when using tetramer to access intrinsic parameters

Loading Intrinsic Data from Internal Model

```
// DB init methods
void StepParametersDB::init_with_model(const std::string& model_name) {


    // Added by Zoe Wefers (McGill University, June 2021, DIMACS REU)
    auto it1 = SequenceDependenceModelList.find(model_name);
    auto it2 = TetramerDependenceModelList.find(model_name);

    if (it1 != SequenceDependenceModelList.end()){
        init_data_from_list_of_string(it1->second._step_parameters_data);
    }
    else if (it2 != TetramerDependenceModelList.end()) {
        init_tetrameric_data_from_list_of_string(it2->second._step_parameters_data);
    }
    else {
        DS_ASSERT(false,
        "non-existing sequence-dependence model name: " + model_name);
    }


    // model name
    m_model_name = model_name;

};
```

```
// Tetramer-dependence model structure
TetramerDependenceModelData::
TetramerDependenceModelData(const std::string step_params[400],
                            const std::string force_constants[400],
                            const std::string& description) :
    _step_parameters_data(),
    _force_constants_data(),
    _description(description) {
        for (Size i=0; i<400; ++i) {
            _step_parameters_data[i] = step_params[i];
            _force_constants_data[i] = force_constants[i];
        };
};
```

```
// sequence-dependence model structure
SequenceDependenceModelData::
SequenceDependenceModelData(const std::string step_params[16],
                            const std::string force_constants[16],
                            const std::string& description) :
    _step_parameters_data(),
    _force_constants_data(),
    _description(description) {
        for (Size i=0; i<16; ++i) {
            _step_parameters_data[i] = step_params[i];
            _force_constants_data[i] = force_constants[i];
        };
};
```

Now there are two lists of internal models, one for dimeric and one for tetrameric. Had to do this because each holds a different amount of information

Change Serialization Class:

```
// StepParametersDB
#define SEQ_DIM 4

//Changed by Zoe Wefers (McGill University, June 2021, DIMACS REU)
template <class ArchiveType>
void save(ArchiveType& archive, const StepParametersDB& steps_db) {

    // model name
    archive(cereal::make_nvp("model_name", steps_db.model_name()));

    // intrinsic step parameters
    for (Size i=0; i<SEQ_DIM+1; ++i)
        for (Size j=0; j<SEQ_DIM; ++j)
            for (Size k=0; k<SEQ_DIM; ++k)
                for (Size l=0; l<SEQ_DIM+1; ++l) {

                    // nucleotides
                    const BaseSymbol first = static_cast<BaseSymbol>(i);
                    const BaseSymbol second = static_cast<BaseSymbol>(j);
                    const BaseSymbol third = static_cast<BaseSymbol>(k);
                    const BaseSymbol fourth = static_cast<BaseSymbol>(l);
                    const TetramerSequence tetra_seq(first, second, third, fourth);

                    // save
                    archive(cereal::make_nvp("first_base",
                                             Base::str(first)),
                            cereal::make_nvp("second_base",
                                             Base::str(second)),
                            cereal::make_nvp("third_base",
                                             Base::str(third)),
                            cereal::make_nvp("fourth_base",
                                             Base::str(fourth)),
                            cereal::make_nvp("intrinsic_step",
                                             steps_db.
                                             intrinsic_bp_step_params(tetra_seq)));
                }

};
```

```
//Changed by Zoe Wefers (McGill University, June 2021, DIMACS REU)
template <class ArchiveType>
void load(ArchiveType& archive, StepParametersDB& steps_db) {

    // model name
    std::string model_name;
    archive(model_name);

    // intrinsic step parameters
    StepParameters db_data[SEQ_DIM+1][SEQ_DIM][SEQ_DIM][SEQ_DIM+1];
    for (Size i=0; i<SEQ_DIM+1; ++i)
        for (Size j=0; j<SEQ_DIM; ++j)
            for (Size k=0; k<SEQ_DIM; ++k)
                for (Size l=0; l<SEQ_DIM+1; ++l) {

                    // load
                    std::string first_str, second_str, third_str, fourth_str;
                    StepParameters step_params;
                    archive(first_str, second_str, third_str, fourth_str, step_params);

                    // nucleotide types
                    const char s1 = first_str.c_str()[0];
                    const char s2 = second_str.c_str()[0];
                    const char s3 = third_str.c_str()[0];
                    const char s4 = fourth_str.c_str()[0];
                    BaseSymbol first = Base::base_symbol_from_char(s1);
                    BaseSymbol second = Base::base_symbol_from_char(s2);
                    BaseSymbol third = Base::base_symbol_from_char(s3);
                    BaseSymbol fourth = Base::base_symbol_from_char(s4);

                    // array filling
                    db_data[(Size)first][(Size)second][(Size)third][(Size)fourth] = step_params;
                }
    // setup
    steps_db = StepParametersDB(model_name, db_data);
```

Loading Data from External Files: ffield-packager

- Changed function that parses text file to be able to handle either a 400 lines file with tetrameric data or a 16 line file with dimeric data
- Turn this data into StepParameterDB/ForceConstantDB objects accordingly

Open Questions

- What is the best way to handle end conditions
  - Get trimer data?
  - Just use dimers – what I have done currently
  - Command line toggle for dna minicircle and change values of 2 important trimers in StepParameterDB/ForceConstantsDB
  - Take averages of tetramers?
- What doe "Lego" Modules do?
  - Some use the intrinsic step parameter functions, but only by passing "AA" StepSequence to the function
  - This the reason I left in the intrinsic param function that takes in StepSequence, what is it doing, can I change it?

```cpp
// step parameters computation method
void LegoTwistedBender::compute_bp_collection() {

    // IdealDNA force field
    // this force field is uniform so we can use dummy sequence
    StepParametersDB model("IdealDNA");

    // default intrinsic step parameters
    BpStepParams p0(model.
                    intrinsic_bp_step_params(StepSequence(BaseSymbol::A,
                                                          BaseSymbol::A)).
                    inline_vector());
    const Real& tw0 = p0.value(TWIST)*DEG_2_RAD;

    // sizing
    const Size n_steps = m_n_bp-1;

    // filling
    std::vector<BpStepParams> prms;
    prms.reserve(n_steps);
    for (Size i=0; i<n_steps; ++i) {
        VectorN pvec = {
            m_bending_angle*std::sin(i*(tw0+m_added_twist*DEG_2_RAD)),
            m_bending_angle*std::cos(i*(tw0+m_added_twist*DEG_2_RAD)),
            m_added_twist,
            FLOAT_INIT,
            FLOAT_INIT,
            FLOAT_INIT
        };
        pvec += p0.inline_vector();
        prms.push_back(pvec);
    };

    // collection
    m_lego_coll = BpCollection::collection_from_bp_step_params(prms,
                                                               BasePair());

};
```

```cpp
// step parameters computation method
void LegoBender::compute_bp_collection() {

    // IdealDNA force field
    // this force field is uniform so we can use dummy sequence
    StepParametersDB model;
    model.init_with_model("IdealDNA");

    // default intrinsic step parameters
    BpStepParams p0(model.
                    intrinsic_bp_step_params(StepSequence(BaseSymbol::A,
                                                          BaseSymbol::A)).
                    inline_vector());
    const Real& tw0 = p0.value(TWIST)*DEG_2_RAD;

    // sizing
    const Size n_steps = m_n_bp-1;

    // filling
    std::vector<BpStepParams> prms;
    prms.reserve(n_steps);
    for (Size i=0; i<n_steps; ++i) {
        VectorN pvec = {
            m_bending_angle*std::sin(i*tw0),
            m_bending_angle*std::cos(i*tw0),
            FLOAT_INIT,
            FLOAT_INIT,
            FLOAT_INIT,
            FLOAT_INIT
        };
        pvec += p0.inline_vector();
        prms.push_back(pvec);
    };

    // collection
    m_lego_coll = BpCollection::collection_from_bp_step_params(prms,
                                                               BasePair());

};
```