

Implementation of time integration methods

- Restructuring of sample code using driver & layout
 - Separates scene layout from simulation algorithms
 - Compartmentalized, reusable operations
 - Switching between integration methods is more straightforward
 - Initially demonstrated on our Forward Euler example
- **WARNING: SOURCE CODE AHEAD!!**

Implementation of time integration methods

- Issues with flat code organization (i.e. everything in main.cpp)
 - Difficult to read (even more so, when we start increasing the complexity)
 - Algorithms and scene setup are not separated
- **WARNING: SOURCE CODE AHEAD!!**

```

#include <PhysBAM_Tools/Log/LOG.h>
#include <PhysBAM_Tools/Parsing/STRING_UTILITIES.h>
#include <PhysBAM_Tools/Read_Write/Utilities/FILE_UTILITIES.h>
#include <PhysBAM_Geometry/Geometry_Particles/GEOMETRY_PARTICLES.h>
#include <PhysBAM_Geometry/Geometry_Particles/REGISTER_GEOMETRY_READ_WRITE.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/SEGMENTED_CURVE.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/FREE_PARTICLES.h>
using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T;
    typedef float RW;

    RW rw=RW();STREAM_TYPE stream_type(rw);
    typedef VECTOR<T,3> TV;

    LOG::Initialize_Logging();
    Initialize_Geometry_Particle();Initialize_Read_Write_Structures();

    const int n=11; // Number of particles in wire mesh
    const int number_of_frames=100; // Total number of frames
    const T frame_time=.05; // Frame (snapshot) interval
    const T youngs_modulus=10.; // Elasticity and damping coefficients
    const T damping_coefficient=10.;
    const T wire_mass=1.; // Mass and length for entire wire
    const T wire_length=1.;
    const T mass=wire_mass/(T)n; // Mass (per each particle)
    const T restlength=wire_length/(T)(n-1); // Restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;particles.Store_Velocity();
    SEGMENTED_CURVE<TV>& wire_curve=*SEGMENTED_CURVE<TV>::Create(particles);
    wire_curve.mesh.Initialize_Straight_Mesh(n);particles.array_collection->Add_Elements(n);
    for(int p=1;p<=n;p++) particles.X(p)=TV(0,(T)(1-p)/(T)(n-1),.5);
    FREE_PARTICLES<TV>& wire_particles=*FREE_PARTICLES<TV>::Create(particles);
    for(int p=1;p<=n;p++) wire_particles.nodes.Append(p);

    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
    collection.Add_Structure(&wire_curve);collection.Add_Structure(&wire_particles);

```

```

DEFORMABLE_GEOMETRY_COLLECTION<TV> collection(particles);
collection.Add_Structure(&wire_curve);collection.Add_Structure(&wire_particles);

ARRAY<TV> force(n),dX(n),dV(n);

T dt_damping=mass*restlength/damping_coefficient;
T dt_elastic=damping_coefficient*restlength/youngs_modulus;
T CFL_number=0.5;
T dt_max=CFL_number*std::min(dt_damping,dt_elastic),dt;
T time=0.;

LOG::cout<<"dt_damping="<<dt_damping<<std::endl;
LOG::cout<<"dt_elastic="<<dt_elastic<<std::endl;
LOG::cout<<"Maximum dt="<<dt_max<<std::endl;

FILE_UTILITIES::Create_Directory("output/0");collection.Write(stream_type,"output",0,0,true);

for(int frame=1;frame<=number_of_frames;frame++){
    T frame_end_time=frame_time*(T)frame;

    for(;time<frame_end_time;time+=dt){
        dt=std::min(dt_max,(T)1.001*(frame_end_time-time));

        // Set upper endpoint position and velocity
        T angular_velocity=two_pi/(frame_time*(T)number_of_frames);
        particles.X(1)=TV(.5*sin(time*angular_velocity),0,.5*cos(time*angular_velocity));
        particles.V(1)=TV(.5*angular_velocity*cos(time*angular_velocity),0,
            -.5*angular_velocity*sin(time*angular_velocity));

        force.Fill(TV()); // Clear all forces from previous iterations

        // Add spring force
        for(int s=1;s<=wire_curve.mesh.elements.m;s++){
            int p1,p2;wire_curve.mesh.elements(s).Get(p1,p2);
            TV X1=particles.X(p1),X2=particles.X(p2);
            TV normal=(X1-X2).Normalized();
            T length=(X1-X2).Magnitude();
            TV f=-normal*youngs_modulus*(length/restlength-1.);
            force(p1)+=f;force(p2)-=f;}
    }
}

```

```
force.Fill(TV()); // Clear all forces from previous iterations
```

```
// Add spring force
```

```
for(int s=1;s<=wire_curve.mesh.elements.m;s++){  
    int p1,p2;wire_curve.mesh.elements(s).Get(p1,p2);  
    TV X1=particles.X(p1),X2=particles.X(p2);  
    TV normal=(X1-X2).Normalized();  
    T length=(X1-X2).Magnitude();  
    TV f=-normal*youngs_modulus*(length/restlength-1.);  
    force(p1)+=f;force(p2)-=f;}
```

```
// Add damping force
```

```
for(int s=1;s<=wire_curve.mesh.elements.m;s++){  
    int p1,p2;wire_curve.mesh.elements(s).Get(p1,p2);  
    TV X1=particles.X(p1),X2=particles.X(p2);  
    TV V1=particles.V(p1),V2=particles.V(p2);  
    TV normal=(X1-X2).Normalized();  
    T vrel=TV::Dot_Product(normal,V1-V2);  
    TV f=-damping_coefficient*vrel*normal;  
    force(p1)+=f;force(p2)-=f;}
```

```
// Add gravity
```

```
force+=-TV::Axis_Vector(2)*mass*9.81;
```

```
// Apply the Forward Euler method
```

```
dX=dt*particles.V; // Compute position change  
dV=(dt/mass)*force; // Compute velocity change  
dX(1)=dV(1)=TV(); // First particle has externally prescribed motion; do not alter  
particles.X+=dX; // Update particle positions and velocities  
particles.V+=dV;
```

```
FILE_UTILITIES::Create_Directory("output/"+STRING_UTILITIES::Value_To_String(frame));  
collection.Write(stream_type,"output",frame,0,true);
```

```
}
```

```
}
```

```
LOG::Finish_Logging();
```

```
}
```

Implementation of time integration methods

- Improvement : Factor out a “Layout” class
 - Includes : Description of simulated geometry
 - Includes : Definition of forces
 - Includes : Definition of kinematic constraints
 - Includes : User-specified simulation parameters
 - Frame rate, # of frames, CFL number, etc.
 - Includes : (Almost) all the state in our virtual world
 - Does NOT include : Time integration algorithms

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    const int number_of_frames; // Total number of frames
    const T frame_time; // Frame (snapshot) interval
    const T CFL_number; // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

const STREAM_TYPE stream_type;

const int n; // Number of particles in wire mesh
const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
const T wire_mass,wire_restlength; // Mass and length for entire wire
ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

GEOMETRY_PARTICLES<TV> particles;
DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

const int number_of_frames; // Total number of frames
const T frame_time; // Frame (snapshot) interval
const T CFL_number; // CFL number (not to exceed 1)

SEGMENTED_CURVE<TV>* wire_curve;
FREE_PARTICLES<TV>* wire_particles;

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

};
}

```



```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    const int number_of_frames; // Total number of frames
    const T frame_time; // Frame (snapshot) interval
    const T CFL_number; // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    const int number_of_frames; // Total number of frames
    const T frame_time; // Frame (snapshot) interval
    const T CFL_number; // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    const int number_of_frames; // Total number of frames
    const T frame_time; // Frame (snapshot) interval
    const T CFL_number; // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:

```

```

    typedef VECTOR<T,3> TV;

```

```

COR template<class T>
COR SIMULATION_LAYOUT<T>::SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input)
COR     :stream_type(stream_type_input),
COR     n(11),youngs_modulus(100.),damping_coefficient(100.),
ARR     wire_mass(1.),wire_restlength(1.),
GEO     collection(particles),
DEF     number_of_frames(100),frame_time(.05),CFL_number(.5)
COR     {
COR     Initialize_Geometry_Particle();Initialize_Read_Write_Structures();
COR     }
COR

```

```

SEGMENTED_CURVE<TV>* wire_curve;
FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);

```

```

void Initialize();

```

```

void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);

```

```

void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);

```

```

void Add_External_Forces(ARRAY<TV>& force);

```

```

T Maximum_Dt();

```

```

void Write_Output(const int frame);

```

```

void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);

```

```

void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);

```

```

void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

```

```

};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:

```

```

    typedef VECTOR<T,3> TV;

```

```

COR template<class T>
COR SIMULATION_LAYOUT<T>::SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input)
COR :stream_type(stream_type_input),
COR n(11),youngs_modulus(100.),damping_coefficient(100.),
ARR wire_mass(1.),wire_restlength(1.),
GEO collection(particles),
DEF number_of_frames(100),frame_time(.05),CFL_number(.5)
COR {
COR Initialize_Geometry_Particle();Initialize_Read_Write_Structures();
COR }
COR }

```

```

SEGMENTED_CURVE<TV>* wire_curve;
FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);

```

```

void Initialize();

```

```

void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);

```

```

void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);

```

```

void Add_External_Forces(ARRAY<TV>& force);

```

```

T Maximum_Dt();

```

```

void Write_Output(const int frame);

```

```

void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);

```

```

void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);

```

```

void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

```

```

};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:

```

```

    typedef VECTOR<T,3> TV;

```

```

    template<class T>
    SIMULATION_LAYOUT<T>::SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input)
        :stream_type(stream_type_input),
          n(11),youngs_modulus(100.),damping_coefficient(100.),
          wire_mass(1.),wire_restlength(1.),
          collection(particles),
          number_of_frames(100),frame_time(.05),CFL_number(.5)
    {
        Initialize_Geometry_Particle();Initialize_Read_Write_Structures();
    }

```

```

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

```

```

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);

```

```

    void Initialize();

```

```

    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);

```

```

    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);

```

```

    void Add_External_Forces(ARRAY<TV>& force);

```

```

    T Maximum_Dt();

```

```

    void Write_Output(const int frame);

```

```

    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);

```

```

    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);

```

```

    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

```

```

};
}

```

```
namespace PhysBAM{
```

```
template<class T>  
class  
{  
public
```

```
void SIMULATION_LAYOUT<T>::Initialize()  
{  
    particles.Store_Velocity();  
  
    wire_curve=SEGMENTED_CURVE<TV>::Create(particles);  
    wire_curve->mesh.Initialize_Straight_Mesh(n);particles.array_collection->Add_Elements(n);  
    for(int p=1;p<=n;p++) particles.X(p)=TV(0,(T)(1-p)/(T)(n-1),.5);  
  
    wire_particles=FREE_PARTICLES<TV>::Create(particles);  
    for(int p=1;p<=n;p++) wire_particles->nodes.Append(p);  
  
    collection.Add_Structure(wire_curve);collection.Add_Structure(wire_particles);  
  
    mass.Resize(n);mass.Fill(wire_mass/(T)n);  
    restlength.Resize(n-1);restlength.Fill(wire_restlength/(T)(n-1));  
}
```

```
SEGMENTED_CURVE<TV>* wire_curve,  
FREE_PARTICLES<TV>* wire_particles;
```

```
SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
```

```
void Initialize();
```

```
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
```

```
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,  
    ARRAY<TV>& force);
```

```
void Add_External_Forces(ARRAY<TV>& force);
```

```
T Maximum_Dt();
```

```
void Write_Output(const int frame);
```

```
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
```

```
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
```

```
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
```

```
};  
}
```

```
namespace PhysBAM{
```

```
template<class T>  
{  
public:
```

```
void SIMULATION_LAYOUT<T>::Initialize()  
{  
    particles.Store_Velocity();  
  
    wire_curve=SEGMENTED_CURVE<TV>::Create(particles);  
    wire_curve->mesh.Initialize_Straight_Mesh(n);particles.array_collection->Add_Elements(n);  
    for(int p=1;p<=n;p++) particles.X(p)=TV(0,(T)(1-p)/(T)(n-1),.5);  
  
    wire_particles=FREE_PARTICLES<TV>::Create(particles);  
    for(int p=1;p<=n;p++) wire_particles->nodes.Append(p);  
  
    collection.Add_Structure(wire_curve);collection.Add_Structure(wire_particles);  
  
    mass.Resize(n);mass.Fill(wire_mass/(T)n);  
    restlength.Resize(n-1);restlength.Fill(wire_restlength/(T)(n-1));  
}
```

```
SEGMENTED_CURVE<TV>* wire_curve,  
FREE_PARTICLES<TV>* wire_particles;
```

```
SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
```

```
void Initialize();
```

```
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
```

```
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,  
    ARRAY<TV>& force);
```

```
void Add_External_Forces(ARRAY<TV>& force);
```

```
T Maximum_Dt();
```

```
void Write_Output(const int frame);
```

```
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
```

```
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
```

```
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
```

```
};  
}
```



```
namespace PhysBAM{
```

```
template<class T>  
class SIMULATION_LAYOUT<T>:  
{  
public:
```

```
    template<class T>  
    void SIMULATION_LAYOUT<T>::Initialize()  
    {  
        particles.Store_Velocity();  
  
        wire_curve=SEGMENTED_CURVE<TV>::Create(particles);  
        wire_curve->mesh.Initialize_Straight_Mesh(n);particles.array_collection->Add_Elements(n);  
        for(int p=1;p<=n;p++) particles.X(p)=TV(0,(T)(1-p)/(T)(n-1),.5);  
  
        wire_particles=FREE_PARTICLES<TV>::Create(particles);  
        for(int p=1;p<=n;p++) wire_particles->nodes.Append(p);  
  
        collection.Add_Structure(wire_curve);collection.Add_Structure(wire_particles);  
  
        mass.Resize(n);mass.Fill(wire_mass/(T)n);  
        restlength.Resize(n-1);restlength.Fill(wire_restlength/(T)(n-1));  
    }  
};
```

```
SEGMENTED_CURVE<TV>* wire_curve,  
FREE_PARTICLES<TV>* wire_particles;
```

```
SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
```

```
void Initialize();
```

```
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
```

```
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,  
    ARRAY<TV>& force);
```

```
void Add_External_Forces(ARRAY<TV>& force);
```

```
T Maximum_Dt();
```

```
void Write_Output(const int frame);
```

```
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
```

```
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
```

```
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
```

```
};  
}
```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:

```

```

template<class T>
void SIMULATION_LAYOUT<T>::Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force)
{
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        TV X1=X(p1),X2=X(p2);
        TV normal=(X1-X2).Normalized();
        T length=(X1-X2).Magnitude();
        TV f=-normal*youngs_modulus*(length/restlength(s)-1.);
        force(p1)+=f;force(p2)-=f;
    }
}

```

```

SEGMENTED_CURVE<TV>* wire_curve;
FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);

```

```

void Initialize();

```

```

void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);

```

```

void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);

```

```

void Add_External_Forces(ARRAY<TV>& force);

```

```

T Maximum_Dt();

```

```

void Write_Output(const int frame);

```

```

void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);

```

```

void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);

```

```

void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

```

```

};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:

template<class T>
void SIMULATION_LAYOUT<T>::Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force)
{
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        TV X1=X(p1),X2=X(p2);
        TV normal=(X1-X2).Normalized();
        T length=(X1-X2).Magnitude();
        TV f=-normal*youngs_modulus*(length/restlength(s)-1.);
        force(p1)+=f;force(p2)-=f;
    }
}
}

```

$$f_1 = -k \left(\frac{l}{l_0} - 1 \right) \frac{x_1 - x_2}{\|x_1 - x_2\|}$$

```

SEGMENTED_CURVE<TV>* wire_curve;
FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

template<class T>
void SIMULATION_LAYOUT<T>::Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force)
{
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        TV X1=X(p1),X2=X(p2);
        TV V1=V(p1),V2=V(p2);
        TV normal=(X1-X2).Normalized();
        T vrel=TV::Dot_Product(normal,V1-V2);
        TV f=-damping_coefficient*vrel*normal;
        force(p1)+=f;force(p2)-=f;
    }
}
}

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

template<class T>
void SIMULATION_LAYOUT<T>::Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force)
{
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        TV X1=X(p1),X2=X(p2);
        TV V1=V(p1),V2=V(p2);
        TV normal=(X1-X2).Normalized();
        T vrel=TV::Dot_Product(normal,V1-V2);
        TV f=-damping_coefficient*vrel*normal;
        force(p1)+=f;force(p2)-=f;
    }
}
}

```

$$f_1^d = -bnn^T(v1 - v2)$$

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int
    const T y
    const T w
    ARRAY<T> 1

    GEOMETRY_
    DEFORMABL

    const int
    const T f
    const T CFL_number;          // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);

};
}

```

```

template<class T>
void SIMULATION_LAYOUT<T>::Add_External_Forces(ARRAY<TV>& force)
{
    for(int p=1;p<=n;p++)
        force(p)-=TV::Axis_Vector(2)*mass(p)*9.81;
}

```

per each spring)

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

```

```

template<class T>
T SIMULATION_LAYOUT<T>::Maximum_Dt()
{
    T maximum_dt=frame_time;
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        T spring_mass=1./(1./mass(p1)+1./mass(p2));
        maximum_dt=std::min(maximum_dt,spring_mass*restlength(s)/damping_coefficient);
        maximum_dt=std::min(maximum_dt,damping_coefficient*restlength(s)/youngs_modulus);
    }

    return CFL_number*maximum_dt;
}

```

ng)

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

```

```

template<class T>
T SIMULATION_LAYOUT<T>::Maximum_Dt()
{
    T maximum_dt=frame_time;
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        T spring_mass=1./(1./mass(p1)+1./mass(p2));
        maximum_dt=std::min(maximum_dt,spring_mass*restlength(s)/damping_coefficient);
        maximum_dt=std::min(maximum_dt,damping_coefficient*restlength(s)/youngs_modulus);
    }

    return CFL_number*maximum_dt;
}

```

$$dt < \frac{ml_0}{b} \quad \text{AND} \quad dt < \frac{bl_0}{k}$$

```

SIMULATION_LAYOUT(const STREAM_TYPE stream
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```



```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients

```

```

template<class T>
void SIMULATION_LAYOUT<T>::Write_Output(const int frame)
{
    FILE_UTILITIES::Create_Directory("output/"+STRING_UTILITIES::Value_To_String(frame));
    collection.Write(stream_type,"output",frame,0,true);
}

```

```

FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire

    template<class T>
    void SIMULATION_LAYOUT<T>::Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X)
    {
        // Set upper endpoint position
        T angular_velocity=two_pi/(frame_time*(T)number_of_frames);
        particles.X(1)=TV(.5*sin(time*angular_velocity),0,.5*cos(time*angular_velocity));
    }

    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients

```

```

template<class T>
void SIMULATION_LAYOUT<T>::Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V)
{
    // Set upper endpoint position
    T angular_velocity=two_pi/(frame_time*(T)number_of_frames);
    particles.V(1)=TV(.5*angular_velocity*cos(time*angular_velocity),0,
        -.5*angular_velocity*sin(time*angular_velocity));
}

```

```

FREE_PARTICLES<TV>* wire_particles;

```

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    template<class T>
    void SIMULATION_LAYOUT<T>::Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array)
    {
        // Only the first particle is constrained
        array(1)=TV();
    }

    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

    const STREAM_TYPE stream_type;

    const int n; // Number of particles in wire mesh
    const T youngs_modulus,damping_coefficient // Elasticity and damping coefficients
    const T wire_mass,wire_restlength; // Mass and length for entire wire
    ARRAY<T> mass,restlength; // Mass (per each particle), restlength (per each spring)

    GEOMETRY_PARTICLES<TV> particles;
    DEFORMABLE_GEOMETRY_COLLECTION<TV> collection;

    const int number_of_frames; // Total number of frames
    const T frame_time; // Frame (snapshot) interval
    const T CFL_number; // CFL number (not to exceed 1)

    SEGMENTED_CURVE<TV>* wire_curve;
    FREE_PARTICLES<TV>* wire_particles;

    SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
    void Initialize();
    void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
    void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
        ARRAY<TV>& force);
    void Add_External_Forces(ARRAY<TV>& force);
    T Maximum_Dt();
    void Write_Output(const int frame);
    void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
    void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
    void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

Implementation of time integration methods

- Improvement : Factor out a “Layout” class
 - main.cpp now only calls the LAYOUT class
 - main.cpp retains the time integration algorithm (for now)

```

#include "SIMULATION_LAYOUT.h"
using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T; typedef float RW; typedef VECTOR<T, 3> TV;
    RW rw=RW(); STREAM_TYPE stream_type(rw);

    LOG::Initialize_Logging();

    SIMULATION_LAYOUT<T> layout(stream_type);
    layout.Initialize(); layout.Write_Output(0);

    T dt_max=layout.Maximum_Dt(), dt, time=0.;

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

[... ]
        }

        layout.Write_Output(frame);}

    LOG::Finish_Logging();
}

```

```

#include "SIMULATION_LAYOUT.h"
using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T; typedef float RW; typedef VECTOR<T, 3> TV;
    RW rw=RW(); STREAM_TYPE stream_type(rw);

    LOG::Initialize_Logging();

    SIMULATION_LAYOUT<T> layout(stream_type);
    layout.Initialize(); layout.Write_Output(0);

    T dt_max=layout.Maximum_Dt(), dt, time=0.;

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

[... ]
        }

        layout.Write_Output(frame);}

    LOG::Finish_Logging();
}

```



```

#include "SIMULATION_LAYOUT.h"
using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T; typedef float RW; typedef VECTOR<T, 3> TV;
    RW rw=RW(); STREAM_TYPE stream_type(rw);

    LOG::Initialize_Logging();

    SIMULATION_LAYOUT<T> layout(stream_type);
    layout.Initialize(); layout.Write_Output(0);

    T dt_max=layout.Maximum_Dt(), dt, time=0.;

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

[... ]
        }

        layout.Write_Output(frame);}

    LOG::Finish_Logging();
}

```

```

[...]  

int main(int argc, char* argv[])  

{  

[...]  

    for(int frame=1; frame<=layout.number_of_frames; frame++){  

        T frame_end_time=layout.frame_time*(T)frame;  
  

        for(; time<frame_end_time; time+=dt){  

            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));  
  

            layout.Set_Kinematic_Positions(time, layout.particles.X);  

            layout.Set_Kinematic_Velocities(time, layout.particles.V);  
  

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);  

            layout.Add_Elastic_Forces(layout.particles.X, force);  

            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);  

            layout.Add_External_Forces(force);  
  

            // Apply the Forward Euler method  

            dX=dt*layout.particles.V; // Compute position change  

            for(int p=1; p<=layout.n; p++){  

                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change  
  

            layout.Clear_Values_Of_Kinematic_Particles(dX);  

            layout.Clear_Values_Of_Kinematic_Particles(dV);  
  

            layout.particles.X+=dX; // Update particle positions and velocities  

            layout.particles.V+=dV;  
  

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);  

            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}  
  

        layout.Write_Output(frame);}  
  

    LOG::Finish_Logging();  

}

```

```

[...]
```

```

int main(int argc, char* argv[])
{
[...]
```

```

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

```

[... ]
int main(int argc, char* argv[])
{
[... ]
    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

```

[... ]
int main(int argc, char* argv[])
{
[... ]
    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

```

[...]
```

```

int main(int argc, char* argv[])
{
[...]
```

```

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

```

[...]
```

```

int main(int argc, char* argv[])
{
[...]
```

```

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

```

[... ]
int main(int argc, char* argv[])
{
[... ]
    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```



```

[...]
```

```

int main(int argc, char* argv[])
{
[...]
```

```

    for(int frame=1; frame<=layout.number_of_frames; frame++){
        T frame_end_time=layout.frame_time*(T)frame;

        for(; time<frame_end_time; time+=dt){
            dt=std::min(dt_max, (T)1.001*(frame_end_time-time));

            layout.Set_Kinematic_Positions(time, layout.particles.X);
            layout.Set_Kinematic_Velocities(time, layout.particles.V);

            ARRAY<TV> force(layout.n), dX(layout.n), dV(layout.n);
            layout.Add_Elastic_Forces(layout.particles.X, force);
            layout.Add_Damping_Forces(layout.particles.X, layout.particles.V, force);
            layout.Add_External_Forces(force);

            // Apply the Forward Euler method
            dX=dt*layout.particles.V; // Compute position change
            for(int p=1; p<=layout.n; p++)
                dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

            layout.Clear_Values_Of_Kinematic_Particles(dX);
            layout.Clear_Values_Of_Kinematic_Particles(dV);

            layout.particles.X+=dX; // Update particle positions and velocities
            layout.particles.V+=dV;

            layout.Set_Kinematic_Positions(time+dt, layout.particles.X);
            layout.Set_Kinematic_Velocities(time+dt, layout.particles.V);}

        layout.Write_Output(frame);}

LOG::Finish_Logging();
}

```

Implementation of time integration methods

- Improvement : Factor out a “Driver” class
 - DRIVER contains all the functionality for advancing the simulation forward in time
 - DRIVER does NOT contain the state of the system
 - DRIVER is reusable, and scene-independent

```

namespace PhysBAM{

template<class T> class SIMULATION_LAYOUT;

template<class T>
class SIMULATION_DRIVER
{
public:
    typedef VECTOR<T,3> TV;

    SIMULATION_LAYOUT<T>& layout;
    T time;

    SIMULATION_DRIVER(SIMULATION_LAYOUT<T>& layout_input)
        :layout(layout_input) {}

    void Run()
    {
        layout.Initialize();layout.Write_Output(0);time=0;

        for(int frame=1;frame<=layout.number_of_frames;frame++){
            Simulate_Frame(frame);layout.Write_Output(frame);}
    }

    void Simulate_Frame(const int frame);
    void Simulate_Time_Step(const T time,const T dt);

};

}

```

```

namespace PhysBAM{

template<class T> class SIMULATION_LAYOUT;

template<class T>
class SIMULATION_DRIVER
{
public:
    typedef VECTOR<T,3> TV;

    SIMULATION_LAYOUT<T>& layout;
    T time;

    SIMULATION_DRIVER(SIMULATION_LAYOUT<T>& layout_input)
        :layout(layout_input) {}

    void Run()
    {
        layout.Initialize();layout.Write_Output(0);time=0;

        for(int frame=1;frame<=layout.number_of_frames;frame++){
            Simulate_Frame(frame);layout.Write_Output(frame);}
    }

    void Simulate_Frame(const int frame);
    void Simulate_Time_Step(const T time,const T dt);

};

}

```

```

namespace PhysBAM{

template<class T> class SIMULATION_LAYOUT;

template<class T>
class SIMULATION_DRIVER
{
public:
    typedef VECTOR<T,3> TV;

    SIMULATION_LAYOUT<T>& layout;
    T time;

    SIMULATION_DRIVER(SIMULATION_LAYOUT<T>& layout_input)
        :layout(layout_input) {}

    void Run()
    {
        layout.Initialize();layout.Write_Output(0);time=0;

        for(int frame=1;frame<=layout.number_of_frames;frame++){
            Simulate_Frame(frame);layout.Write_Output(frame);}
    }

    void Simulate_Frame(const int frame);
    void Simulate_Time_Step(const T time,const T dt);

};

}

```

```

namespace PhysBAM{

template<class T> class SIMULATION_LAYOUT;

template<class T>
class SIMULATION_DRIVER
{
public:
    typedef VECTOR<T,3> TV;

    SIMULATION_LAYOUT<T>& layout;
    T time;

    SIMULATION_DRIVER(SIMULATION_LAYOUT<T>& layout_input)
        :layout(layout_input) {}

void Run()
{
    layout.Initialize();layout.Write_Output(0);time=0;

    for(int frame=1;frame<=layout.number_of_frames;frame++){
        Simulate_Frame(frame);layout.Write_Output(frame);}
}

void Simulate_Frame(const int frame);
void Simulate_Time_Step(const T time,const T dt);

};

}

```

```
namespace PhysBAM{
```

```
template<class T> class SIMULATION_LAYOUT;
```

```
template<class T>
```

```
class
```

```
{
```

```
publ
```

```
template<class T>
```

```
void SIMULATION_DRIVER<T>::Simulate_Frame(const int frame)
```

```
{
```

```
    T frame_end_time=layout.frame_time*(T)frame,dt_max,dt;
```

```
    for(;time<frame_end_time;time+=dt){
```

```
        dt_max=layout.Maximum_Dt();
```

```
        dt=std::min(dt_max,(T)1.001*(frame_end_time-time));
```

```
        Simulate_Time_Step(time,dt);}
```

```
}
```

```
layout.Initialize();layout.Write_Output(0);time=0;
```

```
for(int frame=1;frame<=layout.number_of_frames;frame++){
```

```
    Simulate_Frame(frame);layout.Write_Output(frame);}
```

```
}
```

```
void Simulate_Frame(const int frame);
```

```
void Simulate_Time_Step(const T time,const T dt);
```

```
};
```

```
}
```

```

namespace ...
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    ARRAY<TV> force(layout.n),dX(layout.n),dV(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,force);
    layout.Add_Damping_Forces(layout.particles.X,layout.particles.V,force);
    layout.Add_External_Forces(force);

    // Apply the Forward Euler method
    dX=dt*layout.particles.V; // Compute position change
    for(int p=1;p<=layout.n;p++)
        dV(p)=(dt/layout.mass(p))*force(p); // Compute velocity change

    layout.Clear_Values_Of_Kinematic_Particles(dX);
    layout.Clear_Values_Of_Kinematic_Particles(dV);

    layout.particles.X+=dX; // Update particle positions and velocities
    layout.particles.V+=dV;

    layout.Set_Kinematic_Positions(time+dt,layout.particles.X);
    layout.Set_Kinematic_Velocities(time+dt,layout.particles.V);
}
};
}

```



```

namespace PhysBAM{

template<class T> class SIMULATION_LAYOUT;

template<class T>
class SIMULATION_DRIVER
{
public:
    typedef VECTOR<T,3> TV;

    SIMULATION_LAYOUT<T>& layout;
    T time;

    SIMULATION_DRIVER(SIMULATION_LAYOUT<T>& layout_input)
        :layout(layout_input) {}

    void Run()
    {
        layout.Initialize();layout.Write_Output(0);time=0;

        for(int frame=1;frame<=layout.number_of_frames;frame++){
            Simulate_Frame(frame);layout.Write_Output(frame);}
    }

    void Simulate_Frame(const int frame);
    void Simulate_Time_Step(const T time,const T dt);

};

}

```

Implementation of time integration methods

- Improvement : Factor out a “Driver” class
 - DRIVER contains all the functionality for advancing the simulation forward in time
 - DRIVER does NOT contain the state of the system
 - DRIVER is reusable, and scene-independent
 - `main()` effectively becomes a stub function

```

#include <PhysBAM_Geometry/Geometry_Particles/REGISTER_GEOMETRY_READ_WRITE.h>
#include <PhysBAM_Geometry/Solids_Geometry/DEFORMABLE_GEOMETRY_COLLECTION.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/SEGMENTED_CURVE.h>
#include <PhysBAM_Geometry/Topology_Based_Geometry/FREE_PARTICLES.h>

#include "SIMULATION_LAYOUT.h"
#include "SIMULATION_DRIVER.h"

using namespace PhysBAM;

int main(int argc, char* argv[])
{
    typedef float T; typedef float RW; typedef VECTOR<T,3> TV;
    RW rw=RW(); STREAM_TYPE stream_type(rw);

    LOG::Initialize_Logging();

    SIMULATION_LAYOUT<T> layout(stream_type);
    SIMULATION_DRIVER<T> driver(layout);
    driver.Run();

    LOG::Finish_Logging();
}

```

Implementation of time integration methods

- Swapping out Forward Euler for a different integration method

- e.g.
$$\begin{aligned}x^{n+1} &= x^n + \frac{dt}{2} (v^n + v^{n+1}) \\v^{n+1} &= v^n + \frac{dt}{m} f(x^n, v^{n+1})\end{aligned}$$

- or, for a mass spring system

$$\begin{aligned}x^{n+1} &= x^n + \frac{dt}{2} (v^n + v^{n+1}) \\v^{n+1} &= v^n + \frac{dt}{m} \{ f^{el}(x^n) + G(x^n)v^{n+1} \}\end{aligned}$$

- which leads to:
$$\left(I - \frac{dt}{m} G(x^n) \right) v^{n+1} = v^n + \frac{dt}{m} f(x^n)$$

```

namespace PhysBAM{
template<class T>
class SIMULATION_LAYOUT
{
public:
    typedef VECTOR<T,3> TV;

```

```

template<class T>
T SIMULATION_LAYOUT<T>::Maximum_Dt()
{
    T maximum_dt=frame_time;
    for(int s=1;s<=wire_curve->mesh.elements.m;s++){
        int p1,p2;wire_curve->mesh.elements(s).Get(p1,p2);
        T spring_mass=1./(1./mass(p1)+1./mass(p2));
        maximum_dt=std::min(maximum_dt,2*damping_coefficient*restlength(s)/youngs_modulus);
    }

    return CFL_number*maximum_dt;
}

```

$$dt < \frac{2bl_0}{k}$$

```

SIMULATION_LAYOUT(const STREAM_TYPE stream_type_input);
void Initialize();
void Add_Elastic_Forces(const ARRAY_VIEW<TV>& X,ARRAY<TV>& force);
void Add_Damping_Forces(const ARRAY_VIEW<TV>& X,const ARRAY_VIEW<TV>& V,
    ARRAY<TV>& force);
void Add_External_Forces(ARRAY<TV>& force);
T Maximum_Dt();
void Write_Output(const int frame);
void Set_Kinematic_Positions(const T time,ARRAY_VIEW<TV>& X);
void Set_Kinematic_Velocities(const T time,ARRAY_VIEW<TV>& V);
void Clear_Values_Of_Kinematic_Particles(ARRAY<TV>& array);
};
}

```

```

namespace ...
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    << Somehow solve for  $dV=V^{n+1}-V^n$  >>

    // Trapezoidal position
    dX=(dt/2)*layout.particles.V;
    layout.particles.V+=dV;
    dX+=(dt/2)*layout.particles.V;

    layout.Clear_Values_Of_Kinematic_Particles(dX);
    layout.Clear_Values_Of_Kinematic_Particles(dV);

    layout.particles.X+=dX;    // Update particle positions and velocities
    layout.particles.V+=dV;

    layout.Set_Kinematic_Positions(time+dt,layout.particles.X);
    layout.Set_Kinematic_Velocities(time+dt,layout.particles.V);
}

void Sim
void Sim
};
}

```