CSC 2549 project: FEPR implementation

Ray Wu
rwu@cs.toronto.edu
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada

ABSTRACT

We present an implementation of the Fast Energy Projection for Real-Time Simulation of Deformable objects, featuring two mass-spring models: one of a tetrahedron and one of the Stanford bunny.

1 INTRODUCTION

In traditional applications of numerical integration, issues such as accuracy and order of convergence (among others) are of paramount importance. However, in interactive applications such as computer games, the most pertinent concern is that the computed result must be able to be displayed to the user. Modern applications such as computer games or interactive physics typically require a rate of 30fps or 60fps which leaves about 33 or 17 ms of time for each update to finish. Therefore, the primary concern of any integrator is its stability (an unstable result cannot be shown to the user) and speed (lagging degrades the user experience) in solving for the next update. For stabilty purposes, most applications rely on dissipative integrators such as Backwards Euler. However, as the description "dissipative integrator" states, the stability comes at a cost of numerical damping; energy in the system is removed when it shouldn't be, leading to unrealistic simulations. Additionally, due to the impracticality of computing a timestep to satisfy stability restrictions, there is always the risk of a timestep that is too large for a less stable integrator such as Linearized Backwards Euler which lead to instabilities in the form of a numerical explosion.

The authors of [1] present a novel projection scheme which takes the computed result of any integrator and projects the result back onto a space where conservation of energy is maintained, subject to some constraints on momentum and angular momentum. By projecting the computed result back onto a manifold where energy is preserved, the most serious issues of excessive damping and numerical explosions are avoided.

2 DESCRIPTION OF THE ALGORITHM

FEPR is intended to be an add-on to process the computed result of any integrator. In our example, we implement FEPR as an add-on to a linearly implicit integrator (Linearized Backwards Euler). We introduce the variables and important quantities. We represent the position and velocity of each point as the vectors x and v respectively. The total energy of our system is defined as $H(x,v) = E(x) + \frac{1}{2}v^T Mv$ which is the sum of potential and kinetic energies. We also define the total linear momentum $P(v) = \sum m^i v^i$ and the total angular momentum $L(x,v) = \sum x^i \times m^i v^i$. Given some timestep h, FEPR projects the result of our linearly implicit integrator onto the constant energy manifold by solving the following optimization problem:

$$\begin{aligned} \min_{x,v,s,t} \frac{1}{2}\|x - x_{n+1}\|_M^2 + \frac{h^2}{2}\|v - v_{n+1}\|_M^2 + \frac{\epsilon}{2}(s^2 + t^2) \qquad (1)\\ \text{subj. to } H(x,v) = H(x_n, v_n)\\ P(v) = P(v_{n+1}) + s(P(v_n) - P(v_{n+1}))\\ L(x,v) = L(x_{n+1}, v_{n+1}) + t(L(x_n, v_n) - L(x_{n+1}, v_{n+1})) \end{aligned}$$

Note that when A is positive-definite, $\|x\|_A$ is the A-norm or energy norm of the vector x, defined by $\sqrt{x^T Ax}$.

The above set of equation and constraints (in plain words) state that the FEPR algorithm computes a state x and v that is as close to the result of the integrator as possible, subject to the constraints that energy must be maintained, and that the linear and angular momenta must be an affine combination of the momenta at the current and previous timesteps. Note that since our simulation is in three-dimensional space, P and L have outputs in $\mathbb{R}^3$. Additionally, the parameters s and t are regularized to coerce them to be close to zero (we do not want large shifts in linear or angular momentum)

In [1], the authors choose $\epsilon = 10^{-3}$ and we keep this choice for our implementations.

Like what the authors do in their paper, in our presentation of the algorithm we simplify the notation. Let q be defined as the stacked state vector: $q = [x, v, s, t]$. Additionally, define D to be a diagonal matrix of the same size as q, with the diagonal elements being from the diagonal elements of M, $h^2 M$, and ϵ, ϵ. Additionally, we define a vector-valued function $c(q) = [c_1, c_2, \ldots, c_7]$ which is zero when the constraints are satisified: i.e.

$$\begin{aligned} c_1(q) &= H(x,v) - H(x_n, v_n)\\ c_{2,3,4}(q) &= P(v) - P(v_{n+1}) - s(P(v_n) - P(v_{n+1}))\\ c_{5,6,7}(q) &= L(x,v) - L(x_{n+1}, v_{n+1}) - t(L(x_n, v_n) - L(x_{n+1}, v_{n+1})) \end{aligned}$$

Then, our problem simplifies to

$$\begin{aligned} \min_q \frac{1}{2}\|q - q_{n+1}\|_D^2 \qquad (2)\\ \text{subj. to } c(q) = 0. \end{aligned}$$

To solve this problem, the authors used an iterative method. The authors' first attempt to solve the problem is a sequential quadratic programming approach, but each iterate required solving a dense system of equations that is proportional to the number of unknowns (i.e. particles or finite elements in our simulation) so the speed was unacceptably slow. To resolve this, the authors then attempted to use a quasi-Newton approach, but the number of iterations took excessively long to converge. Finally, the authors

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

settled on a modification of the problem:

$$q^{(k+1)} = \text{argmin}_q \frac{1}{2}\|q - q^{(k)}|_D^2$$
$$\text{subj to } c(q^{(k)}) + \nabla c(q^{(k)})^T(q - q^{(k)}) = 0$$

As the authors explain, each iterate no longer minimizes the D-norm distance to the computed result of the integrator but to the previous timestep. Visual tests by the authors indicate that this is not a big problem.

As with the previous problem formulations (for conciseness reasons, they are not reproduced here), the authors solve their modified problem using Newton's method for convex optimization with constraints, which leads to a sequence of iterates that converge to the optimal solution. This leads to the rise of Lagrange multipliers, and the Karush-Kuhn-Tucker (KKT) conditions lead to the following linear system:

$$\begin{bmatrix} D & \nabla c(q^{(k)}) \\ \nabla c(q^{(k)})^T & 0 \end{bmatrix} \begin{bmatrix} q^{(k+1)} - q^{(k)} \\ \lambda^{(k+1)} \end{bmatrix} = - \begin{bmatrix} 0 \\ c(q^{(k)}) \end{bmatrix} \quad (3)$$

Next, we present the algorithm to solve the modified problem.

(1) The result of the base integrator x_{n+1}, v_{n+1} is computed. Set s and t equal to zero.
(2) Initialize the first guess $q^{(k)} = q_{n+1}$.
(3) We compute c and check that it is sufficiently small (i.e. $\|c(q^{(k)})\|_1 \leq 10^{-7}$).
(4) We solve the linear system (3) for $q^{(k+1)}$ and $\lambda^{(k+1)}$. Since D is constant and the matrix is symmetric, we can use the Schur Complement Lemma to solve for the Lagrange multiplier $\lambda^{(k+1)}$ efficiently:
   (a) Compute the Schur complement: $S = \nabla c(q^{(k)})^T D^{-1} c(q^{(k)})$
   (b) Check that S is full rank. If not, regularize it with $S + 10^{-7} I$, where I is the 7x7 identity matrix.
   (c) Solve the linear system $S\lambda^{(k+1)} = c(q^{(k)})$.
   (d) Obtain the new iterate as $q^{(k+1)} = q^{(k)} - D^{-1}\nabla c(q^{(k)})\lambda^{(k+1)}$. Note that in practice, since we never actually use the matrix D itself but only its inverse, we compute and store its inverse. This is easy because D is diagonal, so its inverse is just the reciprocal of each entry.
(5) return to step (3).

3 RESULTS

We will proceed to show the difference in results between the Linearly Implicit Euler with and without FEPR. Since FEPR uses a quasi-Newton iteration, we would expect a very fast convergence. We also would expect that FEPR preserves the total energy of a simulated system, unlike the Linearly Implicit Euler without FEPR. We show two mass-spring examples, one of a tetrahedron and the other of a bunny.

3.1 Tetrahedron

It is difficult to determine visually the differences between FEPR and no FEPR, because the tetrahedron's deformation is not obvious in video. Therefore, we will just show a graph of the energy levels to show the energy-preserving nature of FEPR. We see that in Figure 1, the integration method with FEPR (blue) keeps the same energy level throughout the integration period (FEPR is only activated after a mouse drag, as otherwise the object will just stay in rest position forever), whereas we see oscillations in the energy level of the integration method with no FEPR (red line) due to stretching.

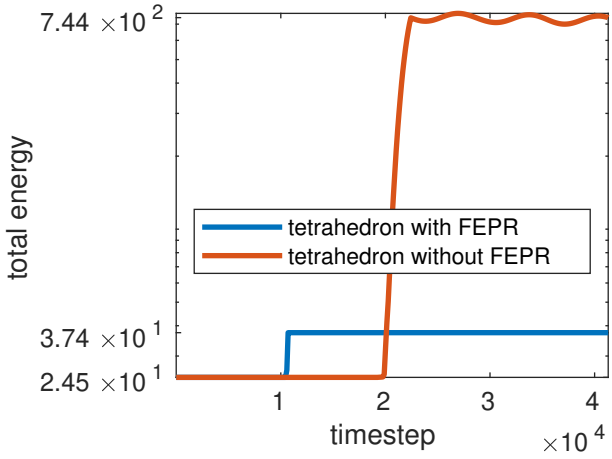

Figure 1: Energy levels of Tetrahedron

3.2 Bunny

For the visual differences in the bunny, please see the accompanying video, where it is clear that FEPR preserves the swaying motion in the bunny's ears after the user drags it and without FEPR the bunny's ears slowly return to their original (rest) position. For the energy levels, please see Figure 2. The animation with FEPR is more lively and the energy levels are preserved, whereas without FEPR the energy level decays back down near the original rest state of 6.04×10^5. The dissipative nature of Linearized Backwards Euler plays the key role in damping the energy level when we don't use the FEPR method.

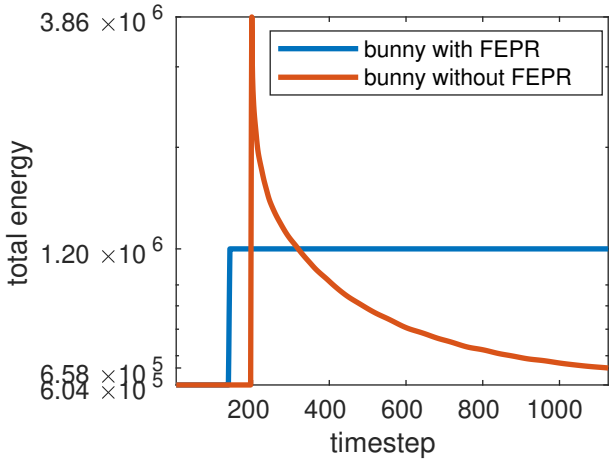

Figure 2: Energy levels of Bunny

3.3 FEPR Iterations per timestep

Based on the theory of convex optimization, quasi-Newton has superlinear convergence and should not take many iterations to

stabilize. Based on our numerical results, the tetrahedron could take many FEPR iterations to stabilize – sometimes up to 100. However, the model is very simple, and thus may not be a representative case. For the more realistic bunny, FEPR only takes several iterations to converge. In my simulation I found that it takes less than 6 iterations to converge, which is to be expected from a quasi-Newton method.

4 CONCLUSION

In conclusion, we have seen how the FEPR projection method allows us to keep the energy level constant during time-integration, so that numerical damping and numerical explosions can both be avoided. One of the drawbacks of the project is that I wish I had done the application of the FEPR method to the finite element armadillo with Linearized Backwards Euler, which as we saw in assigment three exploded unless the user was very careful with dragging. The authors of FEPR claim to be able to preserve stability (see their hippo animation, Figure 11 and Figure 12) and it would have been nice to reproduce this result.

REFERENCES

[1] Dimitar Dinev, Tiantian Liu, Jing Li, Bernhard Thomaszewski, and Ladislav Kavan. 2018. FEPR: Fast energy projection for real-time simulation of deformable objects. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–12.

A FORMULATION OF THE MASS-SPRING SYSTEM

Here, we briefly discuss the modelling of the mass-spring system and computing the gradient of the constraint function c.

In our mass-spring system, the bunny is represented by point-masses each with mass 1 (hence, could be ignored), and the spring connecting them all have the same strain constant but different rest lengths. The bunny's surface is defined by the triangle shapes forming on the boundary.

To take the gradient of c, we need to consider the gradient of each component of c (energy, momentum, angular momentum) with respect to the state variables x, v, s, t.

- For the energy, first we note that the kinetic energy depends only on v and the potential energy only depends on x. We have library functions (the files we implemented in assignment 2: dV...dq) to take the derivative of the potential energy with respect to x. For the derivative with respect to v, only the kinetic energy depends on v so we have from our basic formulas Mv as our derivative.
- For the linear momentum constraint, it does not depend on x or t so those blocks of the Jacobian are zero. The gradient of the linear momentum with respect to each of the v^i's is an identity matrix of size 3, which fills in that $3m \times 3$ block. The gradient of the momentum constraint with respect to s is $-(P(v_n) - P(v_{n+1}))$.
- For the angular momentum constraint, it does not depend on s and its derivative with respect to t is analogous to the linear momemtum with respect to s: $-(L(x_n, v_n) - L(x_{n+1}, v_{n+1}))$. For the derivative with respect to v, note that $L(x^i, v^i)$ is defined as $x^i \times v^i$. Therefore, we can represent this as a matrix-vector multiplication operaton $L(x^i, v^i) = [X^i] \times v^i$ where $[X^i]$, as is convention in the course for rigid bodies, is the cross product matrix. Then we can take the gradient as we do normally with our matrix-vector identites. Note an important detail that while the vectors x and v are vertical in our simulation, in the gradient c they become horizontal, and so we should take the transpose (or the negative, since the transpose of the cross-product matrix is the negative) of the cross-product matrix when assigning the derivative of that block. For the partial derivative with respect to x, we note that the cross product is anticommutative ($a \times b = -b \times a$) so the gradient is $-[V^i]$.

Note that we have fixed points in our simulation (the tetrahedron's first vertex is fixed, as are the vertices around the bunny's feet), therefore we have to keep track of two vectors: a full vector containing all the points and a smaller vector containing only the vertices which are free to move. When computing the energy and momenta, we need to compute from the full vector. However, when computing partial derivatives, we only need to compute those partial derivatives with respect to the points that can move freely.

B RUNNING THE CODE

To run the code, download the repository for assignment 2 (use `git clone –recursive`), and unzip the `*.cpp` files to the `src/` directory (except for `main.cpp` which replaces the `main.cpp` file in the root directory), and unzip the `*.h` files to the `include/` directory. Compiling instructions are the same as the assignment.

To run the code, open a console and run `./<filename> a b` where `filename` is the compiled file, `a` is `tet` for the tetrahedron, anything else for the bunny, and `b` is `no_fepr` for no use of FEPR, and anything else for FEPR.