

Kurt Anderson · Rudranarayan Mukherjee
James Critchley · John Ziegler · Scott Lipton

POEMS: parallelizable open-source efficient multibody software

Received: 28 November 2005 / Accepted: 6 April 2006 / Published online: 14 July 2006
© Springer-Verlag London Limited 2006

Abstract This paper introduces POEMS, parallelizable open-source efficient multibody software, as a modular research package being developed to serve as a backbone software for collaborative research in multibody dynamics. POEMS aims to serve as a repository of efficient implementations of relevant algorithms and be applied to varied applications of interest in research, academia and industry. It allows the user to model systems with varied topologies, and choose between different algorithms for generating and solving the equations of motion as well as for time integration. POEMS utilizes a growing number of solvers, integrators, and system types, which lays a good foundation towards its future development as an optimized, customizable, and parallelizable computational tool for multibody dynamics research and application.

strained to work within the package's given limitations. On the other hand, some of the research packages contain implementation of only specific algorithms, thereby limiting the scope of the package to specific applications.

Several attempts [4] have been made to develop generic multibody packages that allow researchers access to source code and also contain implementation of a variety of different algorithms. These packages have met with varying degrees of success. In this paper we discuss the development of a research package called parallelizable open source efficient multibody software (POEMS). It serves as an environment for programming new solvers, integrators, and relationships (topologies, constraints, force elements). The primary objective of POEMS is to promote research and education in multibody dynamics by providing implementations of different multibody dynamics algorithms in a common package. Additionally, the software is open-source, allowing users access to the source code for customization and collaborative development.

1 Introduction

Multibody dynamics has been and continues to be an active area of research with numerous applications such as robotics, spacecraft and vehicle dynamics, gait analysis, and biomolecular and materials research. There are several commercially available packages including ADAMS [1], Autolev [2], and Dymola [3], that can model and simulate multibody systems. There also exist a number of packages developed in academia which possess similar capabilities. One drawback of the commercially available packages is that they do not allow access to source code and the user is con-

In order to apply an algorithm for a particular multibody dynamics application, a developer normally has to start from an empty directory and face large implementation overheads. POEMS aims to fill this empty directory and cut overheads by providing a user access to an open source research package which can be easily customized for their application. Furthermore, it is equipped with different algorithms for efficiently generating and solving the equations of motion of multibody systems as well as diverse integration schemes and data structure required to function as a full fledged dynamics simulation tool. In this paper, we discuss the structure of the software, its present capabilities, and ongoing developments. In the next section, an overview of the overall structure of POEMS is presented. This is followed by a brief description on the important modules in POEMS. Lastly, the current applications and upcoming features in the software are discussed.

K. Anderson (✉) · R. Mukherjee · J. Critchley
J. Ziegler · S. Lipton
Computational Dynamics Laboratory,
Department of Mechanical Aerospace and Nuclear Engineering,
Rensselaer Polytechnic Institute, 110 8th Street,
Troy, NY 12180, USA
E-mail: anderk5@rpi.edu
URL: <http://www.rpi.edu/anderk5/lab/>
E-mail: mukher@rpi.edu

2 POEMS overview

POEMS is an object-oriented C++ research package for simulating the forward dynamics of multibody systems. It can be compiled and run from both Unix and Windows environments. Additionally, the software can be compiled using most commercially or freely available C++ compilers. It is written with an inherent modularity which allows future developments to be easily added to the main body of the software. Similarly, it can be easily coupled with existing software for applications to different areas. Commonly used kinematic and dynamic identities, organization of multibody topologies, and data structures with matrix manipulations which are generic to most multibody algorithms are built into the software and can be called by different modules. This allows for quick implementations of new algorithms. Equally important, the open-source nature of this software encourages different researchers to access and customize the source code according to their specific applications as well as collaborate with the authors for further development of the package. It should be mentioned that although this package is open-source and all are encouraged to use it, the package is still copyright material and is distributed with a license. For a detailed understanding of the open-source paradigm, the interested reader is referred to reference [5].

3 Current capabilities

The software in its current form has algorithms for solving equations of motion of articulated multi-rigid body systems in chain and tree topologies. There are four primary algorithms that can be used to generate and solve the equations of motion.

- *KaneSolver*() : The $O(n^3)$ complexity solver based on Kane's method [6].
- *OnSolver*() : The $O(n)$ complexity recursive solver based on Anderson's algorithm [7].
- *DCASolver*() : The divide and conquer method of $O(\log(n))$ complexity based on Featherstone's algorithm [8].
- *FMMSolver*() : The $O(n)$ complexity fast particle solver based on the fast multipole method [9].

Along with the solution of equations of motion, the software has time integration algorithms for temporal simulations. These include the following algorithms

- Runge Kutta 4-5
- Predictor corrector
- Verlet and Velocity Verlet
- Lobatto Partitioned III a-b Runge Kutta 4-5

Along with integration schemes, there are step drivers viz. *FixedStepDriver*() for fixed integration time step and *VariableStepDriver* for error checking and variable

step integration step sizes. The software also has a dedicated set of specialized data structures in matrix form for handling vector and tensor quantities typical of multibody systems. The capabilities and the structure of the software are discussed in detail in the following sections.

4 The POEMS structure

POEMS has an applications programming interface (API), and thus is initiated and run from a main line program. The user can initialize POEMS from a main line or use the POEMS user interface to automatically create one. In either case, the main line program is an interface with the rest of the software where the user may specify input and output file names and parameters required for the simulation such as the system description, choice of solver and integrator, length of simulation, desired outputs, etc. Once the main line is executed, the rest of the simulation is organized through a hierarchy of object-oriented classes that contain the simulation details and processes such as system description, generating and solving the equations of motion, integrating the state derivatives, and post-processing. The most prominent of these classes are shown in Fig. 1. Pointers are used generously in POEMS in order to allow for efficient and generic collaboration of the objects.

The main ingredients in a multibody dynamics simulation are the description of the system under study, an algorithm for generating and solving the equations of motion, and a time integration scheme to advance the simulation over time. Correspondingly, POEMS contains classes called *System*, *Solver*, and *Integrator* to accomplish each task.

Every simulation in POEMS begins with a *Workspace* object. This object is the highest in the hierarchy of objects within POEMS. As the name suggests, it behaves

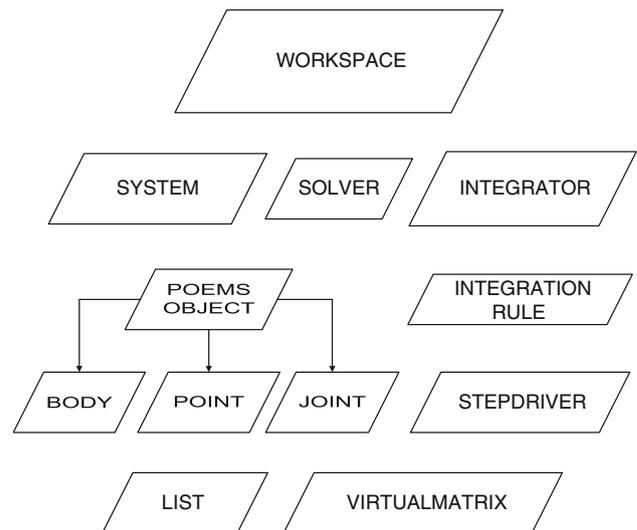


Fig. 1 Higher order POEMS classes

very much like a work space, forming a base of operations for all other processes. It contains and initializes the specific objects required to run the simulation described by the information in the main line program. This class has no derived classes, but organizes the simulation through the use of pointers to the *System*, *Solver*, and *Integrator* objects and has functions by which these objects are added. Furthermore, the *Workspace* hosts the function *Run()*, which contains the total length of the simulation and the integration time-step. *Run()* is used to loop through the solve and integrate process for each time step until the end of the simulation. The *Workspace* also contains functions required to output the specific data during or at the end of the simulation. While there are currently no classes derived from *Workspace*, there are future plans for such classes. Some examples of these include a *RealTimeWorkSpace* that synchronizes the time marching of the solution with real time and an *AnimationWorkSpace* that stores data required for a 3D animation.

The *System* class is the second most important class in POEMS. It is used to generate and maintain the description and topology of the multibody system. It contains and manipulates mass and inertia properties of constituent bodies, kinematic joints which connect the bodies, topological information such as locations of joints and reference points, and forces acting on constituent bodies. This information is contained in two *List* objects, generic linked lists, that organize the multibody system spatially. One *List* named *Bodies* contains the bodies of the system, and another named *Joints* contains the inter-connecting joints of the system. The *System* can also interface any generic force calculation algorithm with the system description.

The *Solver* object is a virtual base class which defines the interface of a generic solver. It is a generic class from which specific algorithms for generating and solving the equations of motion are derived. Each of these algorithms represents a different *Solver* class that implements methods specific to that algorithm by using its derived classes.

The *Solver* class links a solution algorithm to the system contained in the *System* class. Additionally, it contains a map of the state and the state derivatives (the generalized coordinates and their derivatives) of the constituent bodies in the system in order to interface with the *Integrator* class. In POEMS the general solve process is *ComputeKinematics()*, *ComputeForces()*, and then *ComputeSolution()*, each performing the processes described by their names. The details in each of these steps depends on which specific solver is being utilized.

The *Integrator* class is similar to the *Solver* class in that it is a virtual base class which defines the interfaces of specific integration schemes that are derived.

Each integration scheme represents a specific implementation of the virtual base class and relies on polymorphism to implementing steps specific to that scheme. This class contains pointers to the *Solver* which are used

to access state and state derivatives information. The other objects associated with the *Integrator* class are *Stepdriver* and *IntegrationRule*. Their roles in the integration process and association with *Integrator* class are discussed in detail in later sections. Along with these primary objects, there are several other classes or objects which work in association with the primary objects. These are now explained in detail.

5 Describing the multibody system

The organization of the multibody system description begins at the *POEMSObject* class. This is the parent class for all of the derived objects which make up a multibody system. This class is fairly simple, for all it includes is a name and ID (identification). Its purpose is to give a common point from which all other topology classes can be derived.

Three types of *POEMSObject* objects are directly derived from the *POEMSObject* class. These include the *Body*, *Joint*, and *Point* classes, as shown in Fig. 2. From these classes, specific types of bodies, joints, and points are derived.

Although the *Body* class is conceptually of a higher order because all bodies have joints and points, all three of these objects are on the same order within the code's structure. They are linked to each other through the use of *List* objects of the list template class. The *Body* class data include two *List* objects, one of the joint type called *Joints* and one of the point type called *Points*. As implied by the names, these lists contain information about all the joints and points on a specific body in a *System* object. There are currently three types of *Body* objects; the *InertialFrame*, *Particle*, and *RigidBody*. All *Body* types have some common data. For example, every *Body* has position, velocity, and acceleration. However, specific *Body* types may also contain data exclusively associated to that type. For example, only the *InertialFrame* types have a constant acceleration field like

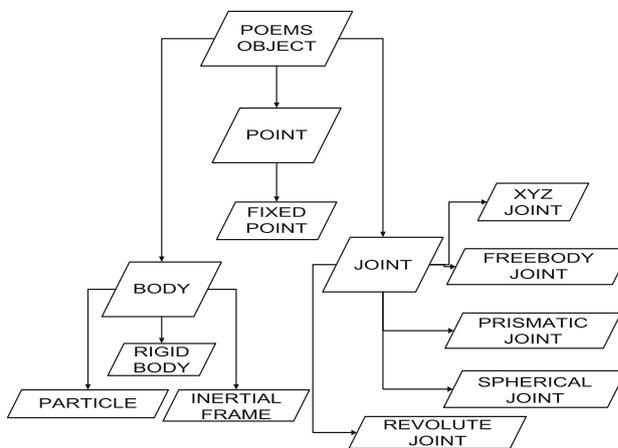


Fig. 2 The multibody objects in POEMS

gravity, the *RigidBody* types have inertias of their own, while *Particle* types just have a mass. The *Body* object also contains the force acting on the *Body* designated as *BodyForce*. This stores the state dependent active forces such as those from an actuator, spring, or damper, and body forces originating from external potential fields as obtained from the force calculation module.

The *Point* class is used for locating important reference points on a body such as joint location or center of mass. It is used in conjunction with the *Body* and *Joint* classes. Currently there exists only one type of point, the *FixedPoint*. At the *Point* level the only data stored are the point's position. The functions contained here allow this data to be set and stored, through the use of the input and output files.

There are currently five types of Joint objects; *FreeBodyJoint*, *XYZJoint*, *PrismaticJoint*, *SphericalJoint*, and *RevoluteJoint*. Similar to the *Body* object, there are some data common to all *Joint* objects while other data are joint type specific. Each joint type has pointers to its parent and child bodies, called *body1* and *body2*. Similarly, each joint type has pointers to a parent point and a child point called *point1* and *point2* where the joints are located on the respective bodies. Also included are their positions, the initial and current values of the states and state derivatives, coordinate transformations between the reference frames of the two connected bodies, and the number of degrees of freedom. As the names suggest, the *PrismaticJoint* and *RevoluteJoint* objects only provide one degree of freedom. The *XYZJoint* and *SphericalJoint* objects provide three degrees of freedom while the *FreeBodyJoint* object provides six degrees of freedom. To avoid singularities arising from the kinematic differential equations, Euler parameters are used in the modelling of *Spherical* and *FreeBodyJoint* objects.

5.1 System topology

The topology for a multibody system is controlled by the algorithm chosen for generating and solving the equations of motion. For example, the recursive $O(N)$ algorithm works in a series of three linear sweeps of all bodies while the divide and conquer scheme works in a binary tree structure. Each solver is associated with its derived classes which are used to describe the topology. For example, for the $O(N)$ algorithm, the *Onsolver* class and its associated class, *OnBody*, are used to describe the topology and recursive properties of the system. The topology handling for the *OnSolver* and the *DCASolver* are discussed next.

5.1.1 Topology for the OnSolver

When a system is added to a *Workspace* and an *Onsolver* object (the chosen solution algorithm) is created and added to the *Workspace*, the *CreateModel()* function is

called to start creating the system topology. The *OnBody* (body type object associated with *Onsolver*) *InertialFrame* is added first. For the *InertialFrame*, a new *RecursiveSetup()* function is called which sets gravity and the pointers of the *OnBody* object to its *Body*, *Joint*, and children objects. Then, starting at the first joint from *InertialFrame*, the first *OnBody* child object is accessed using the *RecursiveSetup()* function. For this *OnBody* object, the required kinematic properties for the $O(N)$ solver are added. The process repeats until all *OnBody* object properties have been defined and the recursion stops.

The *CreateModel()* function now calls on the function *CreateTopologyArray()*. An array of *OnBody* type, *bodyarray*, is first constructed with the *CreateTopologyArray()* function using recursion. This function begins with the *OnBody*, named *Inertialframe*, and then cycles through all the *OnBody* objects beginning at the first child *OnBody* object. This continues until all *OnBody* objects have been added to *bodyarray*. If the system is a chain system, the *OnBody* objects would be numbered in *bodyarray* from 0 starting at *InertialFrame* and ending with the *OnBody* object at the end of the chain. If the system is a tree system, then when the function reaches the body where the system breaks into different chains, it would continue numbering the bodies on the ensuing chains in the system until all bodies have been accounted for.

In the next step of the *CreateModel()* function, the link between the solver state variables and all of the *OnBody* objects located in *bodyarray* are set up using pointers and the *ColMatMap* class by calling on the *CreateStateMatrixMaps()* function. The state variables of *ColMatMap* type, located in the *Solver* class (the parent class of a *Onsolver*), are directly linked to the state variables of each *OnBody* object in a system. This allows for the integrator and solver classes to work seamlessly without having to copy or pass the system states repeatedly between the two classes. Now that the system topology has been defined, the *Solver* can recursively cycle through the body's joints to obtain a solution. The *CreateModel()* function ends at this point. When this function is finished, the pointers that link the *System* object to the *Solver* object are created with the *CreateStateMatrixMaps()* function, and the model is then finished and ready to be solved.

5.1.2 Topology for the DCASolver

In the *DCASolver* which sets up a binary tree structure, the process is very much the same, apart from the *CreateTree()* function where instead of using an array, a tree structure is used to define the arrangement of the system topology.

Beginning at the *CreateModel()* function of the *DCASolver*, similar to the $O(N)$ solver, the *CreateTopologyArrays()* function is called. Here, the solver's pointer to the system is used to access the lists of *Joint*

and *Body* objects. These lists are stored as indexed arrays called *joint* and *body*, which will be used to initialize the *DCABody* objects. The next step in the model creating process is to begin the construction of the balanced binary tree structure. This is done by using the *AVLTree* class to create a *DCATree* object. This class uses *balancefactor* variables to rotate and sort nodes in the tree, in order to always keep it in an optimized configuration. In this tree object, the *DCABody* objects will be the nodes of the tree. Then all of the currently uninitialized *DCABody* nodes are inserted one at a time while the *AVLTree* class keeps the tree balanced (Fig. 3).

Now that the tree is created, the previously created *joint* and *body* arrays can be used to initialize the base nodes of the tree, the physical non-composite bodies. A post-order tree traversal, which begins at the base of the tree and moves up is used, and the *SetPointer()* function is called. In this function, the right and left pointers of the *DCABody* node are checked and if they are NULL, which means it is a base body, the *localbody* and the *localjoint* pointers are set in succession for each base body at a time. At this point, the balanced *DCATree* is created and initialized with pointers to the appropriate data and the system topology is almost completed. In the last step, just as was done for the $O(N)$ solver, the *CreateStateMatrixMaps()* function is called and the system is now ready for the *DCASolver* to compute the solution. This process is summarized in Fig. 4.

6 Solving the equations of motion

There are several different algorithms in POEMS for generating and subsequently solving the equations of motion of the system. The simplest of these is *ParticleSolver*, which solves Newton's equations of motion by a scalar division of the force by mass. A fast particle solver based on the fast multipole method [9] is available in POEMS for efficient $O(N)$ solution in inverse-square type force fields. For articulated multi-rigid body systems in chain and tree configuration, a state-space $O(N)$ recursive algorithm [7] is used for serial implementation. A traditional *KaneSolver* of $O(n^3)$ complexity [6] based on the classical velocity projection method, also known as Kane's method [6], is also available for articulated systems. Featherstone's divide and conquer algorithm [8] is a featured solver for parallel implementations. The implementations of the *OnSolver* and *DCASolver* in

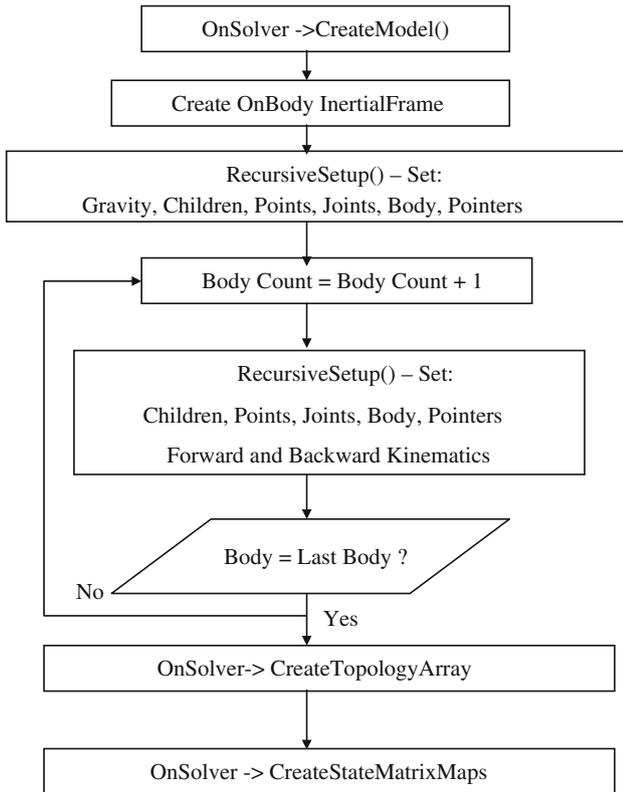


Fig. 3 The CreateTopology() process for the OnSolver

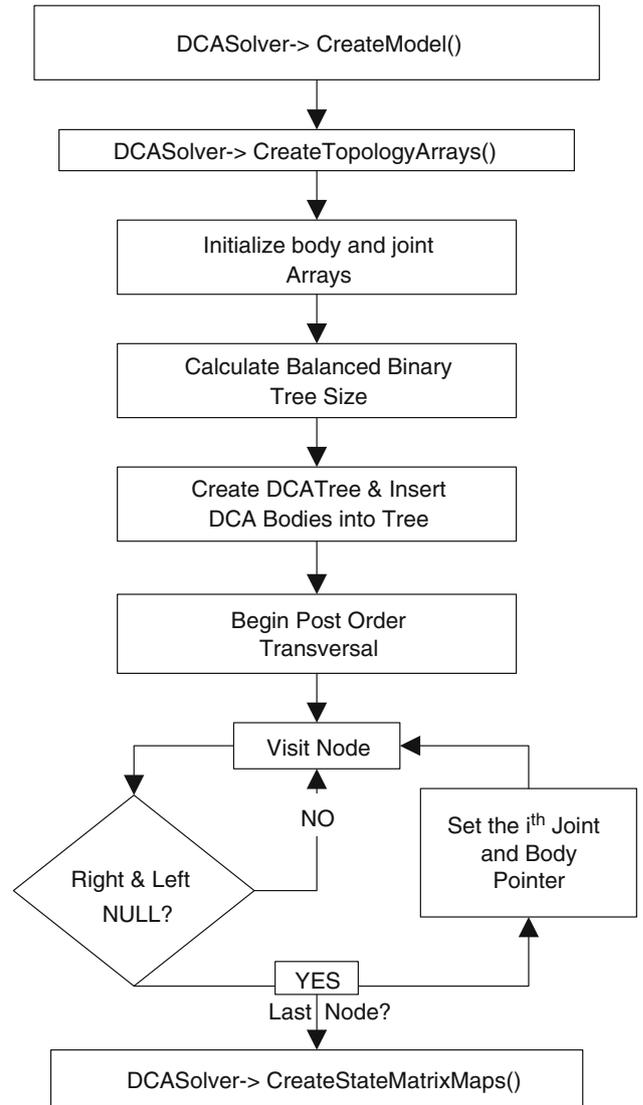


Fig. 4 The CreateTopology() process for the DCASolver

POEMS are described in detail in the following paragraphs. The fast particle solver based on the fast multipole method is discussed in detail in a later section.

6.1 OnSolver

In the *OnSolver* solution process, the array of *OnBody* objects, called *bodyarray*, is used to recursively sweep through the chains and trees of bodies in the system. After the input and topology creation stage is completed, the three sweeps associated with the $O(N)$ recursive solver are carried out. The first sweep starts at the base of *bodyarray*, and works outward to the terminal body, the second coming from the terminal body back to the base, and the last travelling back to the terminal body. A detailed flow chart of this process is shown in Fig. 5.

- The first sweep occurs in the *OnBody* function *LocalKinematics()*. It is called recursively for each *OnBody* object in the array. In this sweep the recursive kinematic properties are calculated. These include the calculation of the velocities, partial velocities, and kinematical relations for relative body accelerations.

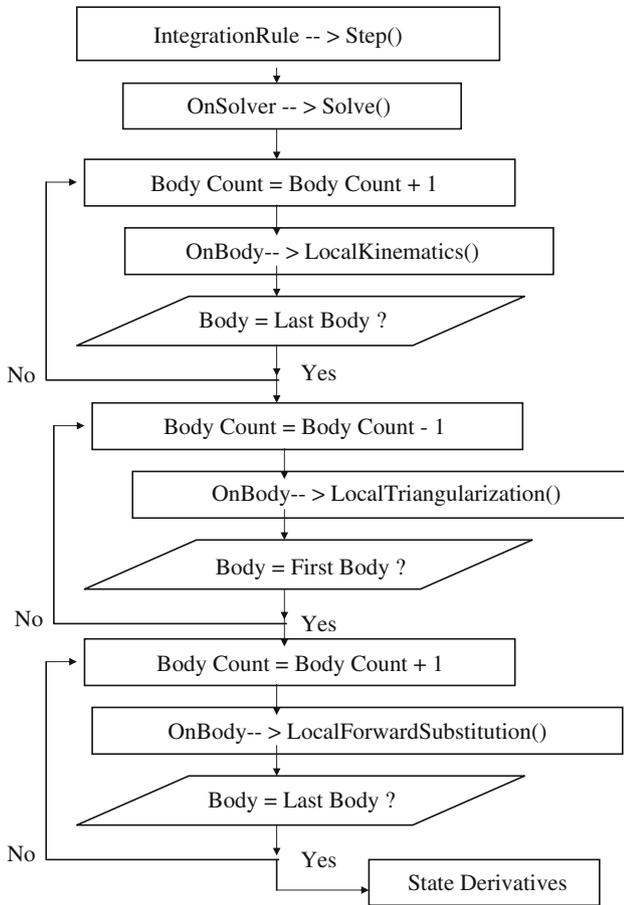


Fig. 5 OnSolver process

- The second sweep is the recursive triangularization sweep and occurs in the *LocalTriangularization()* function. It calculates the generalized inertia and active forces and the articulated inertia values and conducts a recursive triangularization of the equations of motion.
- The third sweep is a recursive forward substitution that recursively calculates the generalized state derivatives for each body in the *LocalForwardSubstitution()* function.

6.2 DCASolver

In the *DCASolver* solution process, four traversals of the balanced AVL binary tree, *DCATree*, are used. This tree is initialized in the create model process as described earlier. These traversals are similar to the commonly known C++ pre-order and post-order binary tree traversals. The leaf nodes of the tree are the actual physical bodies of the system while the top is the single composite assembly of the entire system. The pre-order traversal is used to traverse from the root of the tree to the leaf nodes of the tree, while the post-order traversal is used to traverse from the leaf nodes of the tree to the root. The four sweeps of this divide and conquer algorithm are consecutively post-order, pre-order, post-order, pre-order traversals, with the last traversal yielding the accelerations of the body's joint handles.

- In the first sweep of the DCA algorithm, only the base *DCABody* nodes are visited consecutively by the *InitializeNode()* function. The *InitializeNode()* function is contained in the *DCABody* class. In this function, the quantities needed exclusively for the DCA algorithm are calculated. This includes the articulated inertia terms for the body handles for each of the joints and the joint motion subspace relations.
- In the sweep two, the *ComputeForces()* function is called using a pre-order traversal. This function calculates any actuator forces between the bodies or any force field relations.
- In sweep three, the *Sweep3()* function is used in a post-order traversal, which visits all nodes except the base nodes to build the articulated composite inertias terms for each of the composite bodies. Also in this function, pointers are passed up from the left child nodes in the tree, so that the composite *DCABody* nodes can have access to the desired *localjoint* and *localbody* pointers. At the conclusion of the third sweep, the single composite body for the system is constructed, and the first handle acceleration and force values can be calculated using the boundary conditions of the system.
- Sweep four begins in the *Sweep4()* function, where only the composite nodes are visited. In this function, the left and right nodes of each *DCABody* object are accessed and the handle forces and accelerations between each body are calculated in a pre-order traversal.

It is noted that in this last sweep, pointers are used for each node to point each *DCABody* node's state derivative variable to the respective *Joint* object's variable. This allows for the coordination of updating the system, solver, and integrator values of the state derivatives as explained earlier.

Although, the DCA solver has been implemented only in serial, it has the infrastructure that allows for easy extension to parallel implementation. Due to this algorithm's inherent balanced binary tree structure, it can be parallelized on an optimal number of processors equal to the number bodies in the system. Depending on actual communication costs additional parallelization with individual bodies may be realized. Each of the four sweeps described above can be parallelized, breaking up the computing tasks at each level of the tree.

7 Force calculations

The active forces on the system can be of varied types such as actuator force, springs, dampers, friction or gradients of some inter-body potential field. Given the varied nature of these forces, the user has to specify how forces are calculated within the POEMS software by writing an application-specific force calculation function. This may seem complicated, but to maintain flexibility in applications of POEMS, allowing the user to write the force field calculations is the best option. This also allows the user to interface POEMS with other software dedicated for calculating forces such as CHARMM [10] in molecular dynamics simulations.

Constant force fields like uniform gravity are easily accommodated into the software. This is carried out by accelerating the inertial frame in the opposite direction of the gravity force vector. The user simply has to specify the constant acceleration value once during the input stage and it is accommodated into the system description during the *CreateModel()* function.

Other active forces depend on the state of the system, i.e., the position and velocity data for the component bodies in the system. Any custom force calculation scheme introduced by the user has to access this data. The *System* object maintains a list of bodies. The elements in these lists are the component body instances created in the simulation. The body object stores the position and velocity of all points on it. Thus by accessing the list of bodies from the *System*, the position and velocity data for each body in the system can be accessed. This data can now be used by the user's custom force calculation scheme to generate the forces. The calculated forces need to be passed to the *Solver* for setting up the equations of motion. One way to do that is to pass the calculated forces as an input in the *Run()* command. Alternately, one could create an array of forces for each body in the *Body* object and store the forces there at every integration step. The open-source nature of this code provides flexibility to the user to

create data structures and functions for force calculation for their specific requirement.

8 Time integration

POEMS has several integrators and step drivers in its library. The integrators include Runge–Kutta 4/5, Runge–Kutta–Fehlberg 4/5, Lobatto, Predictor–Corrector, Verlet schemes, and simple Forward Euler. Different step drivers also exist, including variable step and fixed step drivers. New integration schemes and step drivers can be easily added to the POEMS library by the user for a custom simulation.

Integration in POEMS is run by the general classes of the *Integrator*, along with the specific *IntegrationRule* (e.g. *RungeKutta 45*) and the specific *Stepdriver* class (e.g. *VariableStep*). The bulk of the integration process takes place in the specific *IntegrationRule* object where the integration scheme is implemented. An *IntegrationRule* object also contains the pointers and functions that are used to link all of these classes together with the system and solver.

The integration process begins when the *Integrate()* function of the *Integrator* class which is called from the *Workspace*'s *Run()* function. From the *Integrate()* function, a pointer to the *Stepdriver* object is used to access the driver's *Step()* function. The *Stepdriver* class is a virtual class and is only used to organize the different types of drivers. Consequently, the *Step()* function of a specific derived driver class (e.g. *VariableStep*) is accessed. Here, the current step size, δt , is set using the using the properties of the specific step control scheme. After getting the δt value, a pointer to the object, *Intrule*, of the *IntegrationRule* type is used to run the *Step()* function of the specific *IntegrationRule* object. The system is simulated here by calling the solver and running the integration scheme. The process ends here for a constant step driver. For step drivers with error control, the error is estimated while still in the *Step()* function of the specific driver object. If the error is acceptable, then the *Step()* function finishes and the next integration is started by using the *Integrate()* function again. If it is not acceptable, then the *Reset()* function of the specific *IntegrationRule* object is called where the time step δt is reduced and the integration step is carried out till the acceptable error level is reached. In this fashion integration continues for each iteration in the loop of the *Run()* function until the maximum time of simulation is reached.

9 The fast multipole method

POEMS employs an implementation of fast multipole method (FMM) [9] for calculating 3-D particle interactions. It follows through the primary steps of the upward pass, downward pass, and direct neighbor interactions as prescribed in [9]. For a detailed analysis of the theory

and principles behind this algorithm, the interested reader is referred to [9]. The FMM code used in POEMS utilizes its own unique set of data structures and functions not used elsewhere in the program, which are explained next.

The two important classes are the *Octree* class and the *Box* class. The *Octree* class possesses a function *Subdivide()* which subdivides a *Box* by creating eight new boxes which represent its children. The class makes use of two other essential functions, *Neighbors()* and *InteractionList()*. *Neighbors()* takes a given *Box* and inserts a list of its neighbors into an array. *InteractionList()* generates an array for a *Box* and inserts a list of boxes within its interaction list by using the *Neighbors()* function.

While the *Octree* serves as the foundation of the code, the *Box* objects themselves contain most of the primary data and functions to make it work. Each *Box* represents a cube of some size within the entire computational space. It holds information pertaining to its number, level, center coordinates, and parent, which are specified upon the object's creation as well as specific data such as numbers, strengths, locations, and charges of particles located within the *Box*. All of these computational boxes are organized spatially using bit interleaving procedures to create uniquely ordered box numbers. Two functions which are not a member of *Box*, *BitInterleave()* and *BitDeinterleave()*, are used for this. The *Box* objects employ many functions which are crucial to the code. These are

- *CalcMultipole()* -Calculates the multipole expansion of the *Box* based on the locations and strengths of particles within it
- *ShiftMultipole()* -Shifts the multipole expansion of the *Box* to another *Box* which is specified as an argument of the function
- *CalcLocal()* -Calculates the local expansion of the *Box* based upon the multipole expansions and locations of all *Boxes* within its interaction list
- *ShiftLocal()* -Shifts the local expansion of the *Box* to another *Box* which is specified as an argument of the function
- *EvalPotentials()* -Evaluates the potentials of each particle within the *Box* based upon the particle location and the local expansion of the *Box*
- *EvalDirectOutside()* -Directly evaluates the potential force upon each particle in the *Box* from those in another *Box* based upon the locations and strengths of the particles in each
- *EvalDirectInside()* -Directly evaluates the potential force upon each particle in the *Box* from other particles within the same *Box* based upon their locations and strengths

On its own, the *Box* object bears most of the essential data and methods used in the FMM code. The main code itself runs through seven primary steps, much like the description given by [9]. The two initial function calls

to *ScaleFactor()* and *CoordShift()* are used to modify the particle locations such that all particle coordinates are positive and less than unit dimension. The *Octree* is created and *Subdivide()* is used until the proper mesh levels exist. The function *BoxOf()* takes the new coordinates and returns the number of the box at the finest level of the mesh in which the particles reside. The function *ATERM()* is then called to create an array of the *A* terms which are used in the FMM. This completes the preprocessing steps (Fig. 6).

The first step is to form the multipole expansion for each *Box* at the finest level of refinement, accomplished by calling *CalcMultipole()* upon them. The second step involves shifting the multipole expansions to each parent at each level. This is performed by calling *ShiftMultipole()* numerous times, starting at the finest level and proceeding to coarser levels until the expansion for every single *Box* has been formed. It is to be noted that several assistant functions facilitate this process. Among these are *SphereCoord()*, which takes Cartesian coordinates and converts them to spherical coordinates, *SphereHarm()*, which returns the spherical harmonic of a given θ and ϕ . The third step involves use of both *CalculateLocal()* and *ShiftLocal()*. It involves finding the local expansion of each *Box* at a given mesh level, starting with the first level, and then shifting these newly

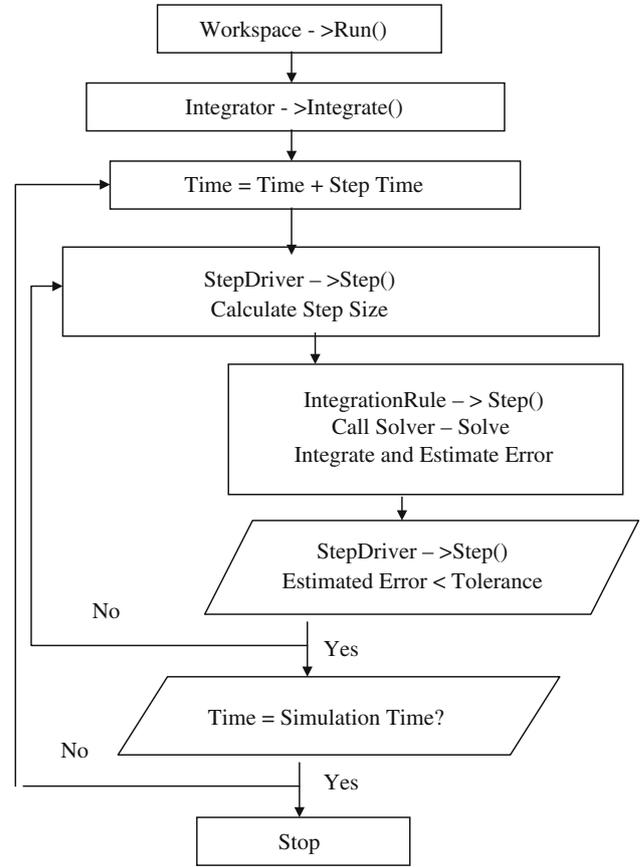


Fig. 6 The integrator class

found expansions to each child *Box* at the next level of refinement. This is repeated until the fourth step at the finest level, where only *CalcLocal()* is used since there are no children. Step five is a simple calling of *EvalPotentials()* upon each *Box* to compute the particle potentials from the local expansions. The only remaining steps are to find the direct interactions from neighbor *Boxes* and within each *Box* using *EvalDirectOutside()* and *EvalDirectInside()*, respectively. Once summed, the algorithm is complete and the particle potentials have been found. This completes the FMM algorithm.

10 Data structures in POEMS

The *List* object is a class of its own and allows the user to add items to a dynamic list of elements. The use of lists in POEMS is hardwired into the software, as one can manipulate these lists by using the add and remove element functions of the class. The organization of *List* objects is shown in the Fig. 7. Two *List* objects are stored in a *System* object, a *List* of all the *Body* objects organized as they appear spatially in the system, and a *List* of *Joint* objects that follows the organization of the *Body* objects. These *List* objects follow a specific numbering scheme to organize the description of the multi-body topology. The other two *List* objects are the aforementioned *joints*, and *points* lists which contain topology information locally on each body.

The heart of all mathematical operations in POEMS is located in matrix operations, and thus matrices are the primary data structure used in the program. Matrix operations, much like a Matlab software [11] environment, are programmed into the POEMS software. Operator functions allow all the basic arithmetic and additional optimized operations such as cross products, stacking and unstacking matrices, Matrix inversions, LU factorizations, and Euler parameter operations. In addition, special optimized matrix operations are utilized from the *FastMatrixops* file. Also, within the specific matrix type classes, the data of each matrix are directly stored in appropriately sized arrays with double precision.

POEMS uses an object-oriented approach to store matrix objects in order to allow inheritance of different matrix types. There are many different matrix types, some of which are shown in Fig. 8. Specific matrix classes are derived from virtual parent classes and allow concise definition of all of the matrix properties. By defining specifically sized matrix objects, optimization of the software increases. The *VirtualMatrix* class is the parent class for all of general Matrix types: 3×3 , 4×4 , banded matrix, 6×6 , or any general size. The *Matrix* type is any general sized matrix, the *Banded Matrix* type is a general size banded matrix, and the other three are 3×3 , 4×4 , and 6×6 square matrices. Examples of 3×3 matrices which are often used are inertia, transformation, identity, zero, and shift symmetric (used for the cross product). The 4×4 matrices are specifically used for Euler parameters. Sometimes a spatial matrix is required, which warrants the use of a 6×6 matrix. Some examples of this case are an inertia spatial matrix, with both inertia and mass terms, a shift matrix, and a spatial transform matrix.

The *VirtualColMatrix* and the *VirtualRowMatrix* classes are very similar to the *VirtualMatrix* class and lead to the derivation of the column and row matrices. The *RowMatrix* type is derived from its parent *VirtualRowMatrix* class and the *ColMatrix*, *Vect3*, *Vect4*, *Vect6*, and *ColMatMap* types are derived from the parent *VirtualColMatrix* class. The *RowMatrix* type is a row matrix of any arbitrary length. The *ColMatrix* type is a column matrix of arbitrary length; it can be used as a vector or in data manipulation when using matrices that do not have column or row lengths of 3, 4, or 6.

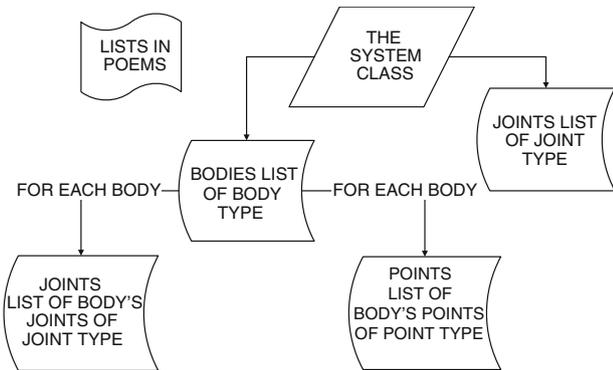


Fig. 7 Organization with list class

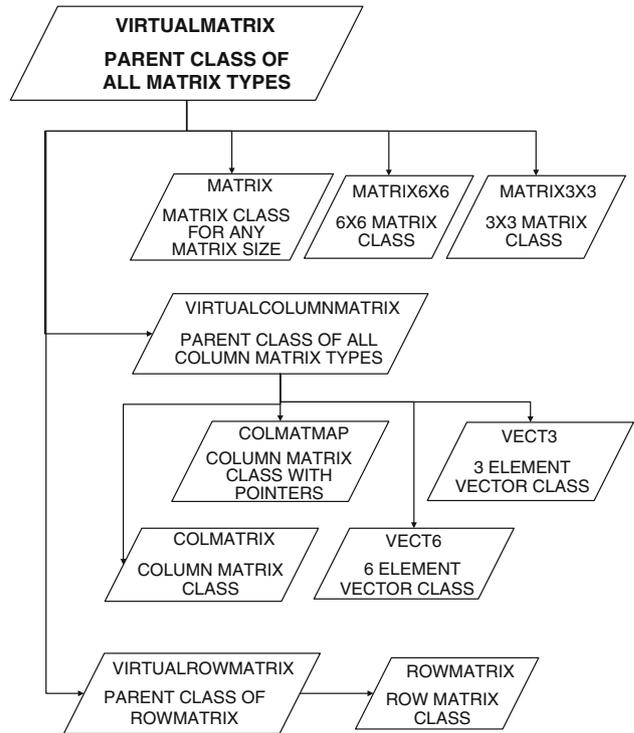


Fig. 8 Using matrices in POEMS

11 The user interface

POEMS' user interface is designed for user-friendliness and simplicity. It was created with Qt4, a C++ GUI software toolkit, and uses a straightforward design that allows one to construct a dynamics simulation with ease. It consists of three widgets (Qt objects): a *MainWindow*, *BodyEditor*, and *JointEditor*. They are discussed in some detail below.

The *MainWindow* widget serves as the center of operations for the program and it is laid out with different sections containing selections for each aspect of the simulation as shown in Fig. 9. The first group in the upper left shows possible types of problems for the user to solve, including rigid body, flexible body, both, or particle. Depending on this choice, the next group to the right will show different solvers available to handle the type of problem chosen. The next two sections possess choices on whether to use Cartesian or relative coordinates and what type of step driver to be used. Several numerical inputs follow, including length of simulation, step size, absolute tolerance, and relative tolerance. Two more sections present choices of integrator and forces. For the selection of forces, the user may choose to use uniform gravity or browse for an external file containing a custom force calculation algorithm.

The input of the system data is handled using the *BodyEditor* and the *JointEditor* as shown in Fig. 10. These are initiated by selecting the *User Typed* option in the main menu. This manual input system in POEMS allows for quick and efficient creation of joints and bodies. There are push buttons which add, edit, or delete from a list of bodies and a list of joints. In the *BodyEditor* window the user may specify a name and type of body, choosing from inertial-frame, rigid body, or particle. Depending on this second choice, other parameters of the body will appear to be created or changed. Some of these include center of mass, handles, mass, inertia, velocity, and angular velocity. Upon confirming the

choices, the body is either created or changed as specified. A similar process occurs when the user desires to create or edit a joint in the list and a *JointEditor* widget is called. The primary difference, however, is that the *JointEditor* calls up a current list of the bodies and allows the user to select one which is its parent body and one which is its child body, choosing the handle point on each where it is connected. Additionally, the joint's axis with respect to the global axis is specified. Through the use of these two processes, a simple system may be defined rapidly and changed easily. Moreover, manually entered bodies and joints are saved along with the rest of the simulation, so the user may add or remove them over several sessions as desired. However, manually entering very large systems can be very costly and time-consuming, and such cases would call for external data input. Hence the *External File* option, as with the forces, allows the user to browse for an existing data file. This option is for either restarting a simulation where the system data were stored earlier or for the user to start the simulation from a custom data file. For systems with a large number of bodies, the *POEMS Generated* option can be used. This is for reading from different data files that contain the system data and writing the POEMS specific data file. There are several utilities that support this feature, but not all data input scenarios are covered.

Finally, there is a group of push buttons in the lower right corner which are used to run the simulation, start a fresh one, save the current one, load another one from an external file or exit the program. These functions may also be called from the File group at the top menu.

12 Verification of the software

The recursive $O(N)$, Kane, and divide and conquer solvers for articulated body systems have been extensively tested and verified against the industrial strength

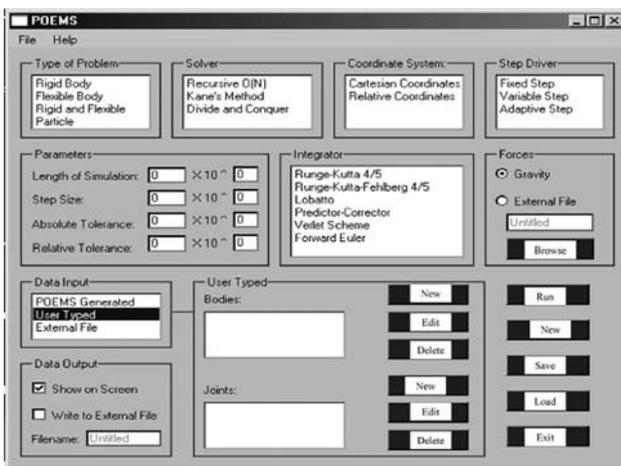


Fig. 9 Main menu of POEMS GUI

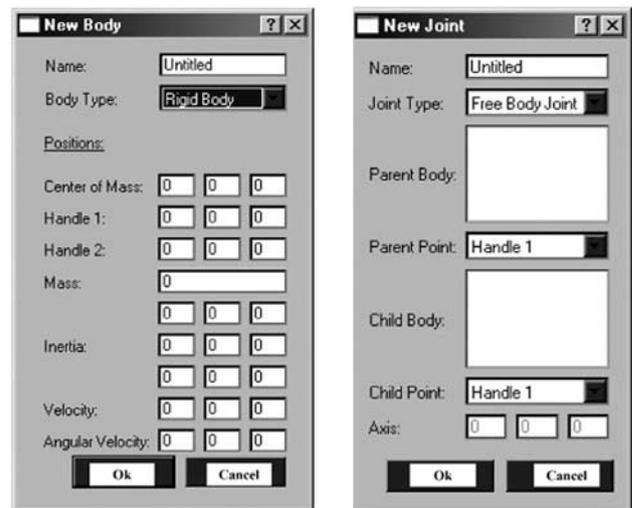


Fig. 10 Other menus of POEMS GUI

multibody software Autolev™. The systems compared are two- and three-dimensional many-body articulated systems connected by different kinematic joints including revolute, prismatic, spherical and six degree of freedom free-body joints. The test results compared the system states and state derivatives generated by POEMS with identical systems simulated using Autolev™. For all of the test cases, the states and state derivatives matched up to machine precision for the same error bounds on the integrators. The result of the fast multipole method based *FMMSolver* compared with a direct inter-particle solver is presented in the Fig. 11 which shows the absolute error for each particle. The implementation of the *FMMSolver* is based on a five term multipole expansion at three levels of discretization.

13 Application in simulation polymer melts

The dynamics of the bead-spring model of polymer chains is simulated in this application. In order to increase the time step of integration as compared to traditional molecular dynamics simulations, the inter-bead axial vibrations are constrained. In the analysis, the bead spring model of the polymer chains is replaced by an articulated body model made of beads connected by massless rigid links. The connection of successive links to beads is modelled using kinematic joints and relative degrees of freedom.

The system discussed in this paper consists of 32 polymer chains and each chain has 16 beads. The interaction between the beads is modelled as non-bonded interatomic interactions, by using the normalized Lennard–Jones pairwise potential. The system is modelled under periodic boundary conditions and as a constant energy ensemble. For the simulations discussed here, the simulation box is a cube and the material density is maintained at about 0.85. In the model, the chains are free floating. The base body is modelled as

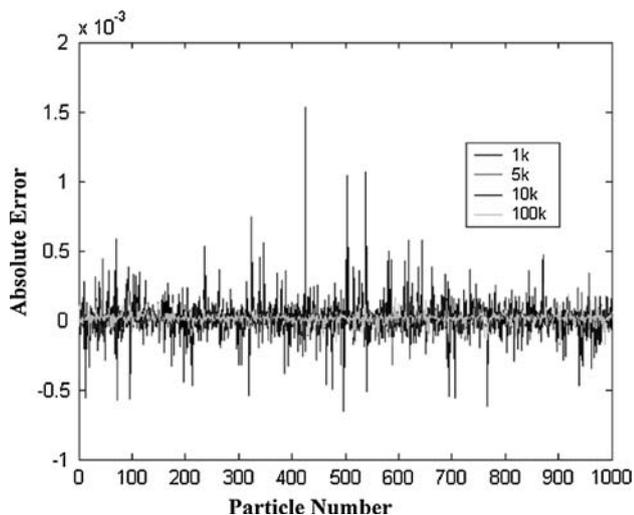


Fig. 11 Error from the FMMSolver

connected to the inertial reference frame by a six degree of freedom joint allowing relative translational and rotational degrees of freedom. The successive bodies are connected to the base body by spherical joints allowing only rotational degrees of freedom. The time integration is carried out using the Lobatto integration scheme as found in [12]. The implementation of this work in parallel and some of the results are discussed next.

13.1 Coarse-grain parallel implementation

In our parallel implementation, a Linux cluster using four processors is used. The parallel communications are handled using message passing interface (MPI). The 32 chains are equally divided and 8 chains are assigned to each processor.

Figure 12 shows a flow chart of the work done on each processor. The simulation begins with the identification of the number of processors available and identification of the rank of individual processors. Based on the rank of a processor, it is assigned certain chains in the system. The processors all load the same data file which contains the position and velocity of all chains in the system. Each processor calculates the forces acting on its assigned chains from the position data of all atoms in the system. The processors then compute the system description in relative coordinates of only the chains assigned to it. This is a fixed one time cost. At this point a loop begins as shown in the flow chart. Each processor formulates, solves and integrates the equations of motion for the chains assigned to it. Since each chain is an independent articulated body system, the processors are independent of each other. The new states, obtained from the time integration, are now modified to enforce periodic boundary conditions. Each processor then communicates with all other processors and exchanges the position data. After the communication with all processors is complete, the processor calculates forces acting on the chains assigned to it for the next time step. The processors loop through this process until the total number of time integrations are complete.

From the above scheme, it is clear that there is a need for one communication per integration step where the positions of the beads on the chains are exchanged. Apart from this one communication overhead, the processes assigned to each processor needs no further communications. Each processor independently and serially implements the $O(n)$ solver and the force calculation algorithm. This reduces the loss of efficiency due to communication and allows for an effective speedup on a modest number of processors.

13.2 Validation and results

Figure 13 shows a plot of kinetic, potential and total energy of the system over long temporal scales for a constant energy ensemble. In the figure, a small drift in the total energy can be observed. However the drift is

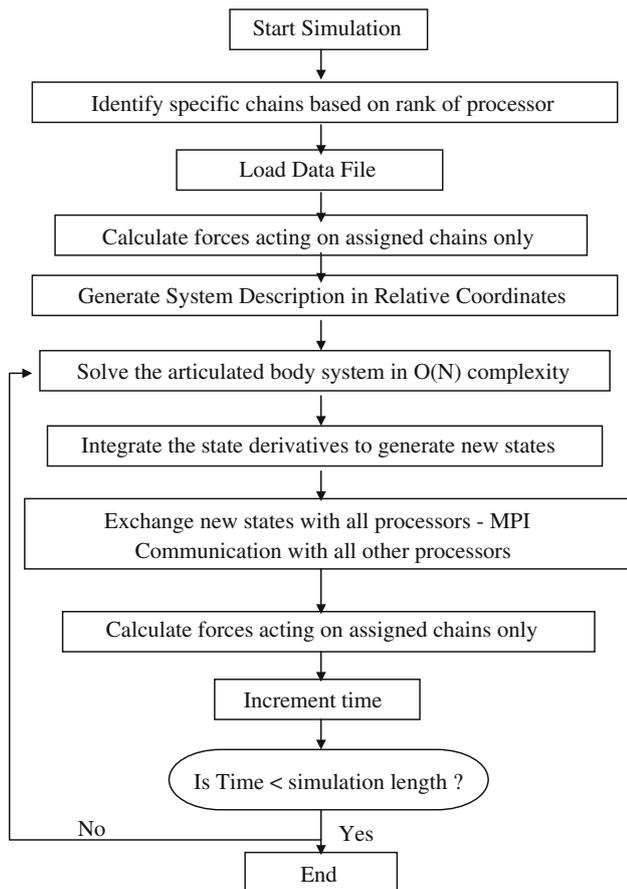


Fig. 12 Work done on each processor

significantly small, the ratio of standard deviation to mean value is only about 0.63%. By eliminating the high frequency axial vibration of the beads, an increase in the time step by *an order of magnitude* is obtained.

A few critical structural properties commonly studied in polymer melt simulations are also presented. Figure 14 shows the plot of the mean-square displacement of the center of mass of the system $\langle g \rangle^3$ and that of the middle beads in the chain $\langle g \rangle^1$ on a log-log scale. The g^3 values attain a slope of 1 and maintain a constant value of 1. This is in agreement with characteristic features of fluids, further indicating the validity of the model.

The mean square end-to-end vectors $\langle R2 \rangle$ and the mean square Radius of Gyration $\langle Rg2 \rangle$ calculated from the trajectory for polymer chains as shown in Fig. 15. As expected, both these values are varying with time but the moving average of both quantities maintain a constant value. Further, the ratio of $\langle R2 \rangle / \langle Rg2 \rangle$ is about six, as expected of Gaussian chains.

14 Future work and conclusions

As indicated earlier, POEMS is an ongoing collaborative effort aimed at developing an open source computa-

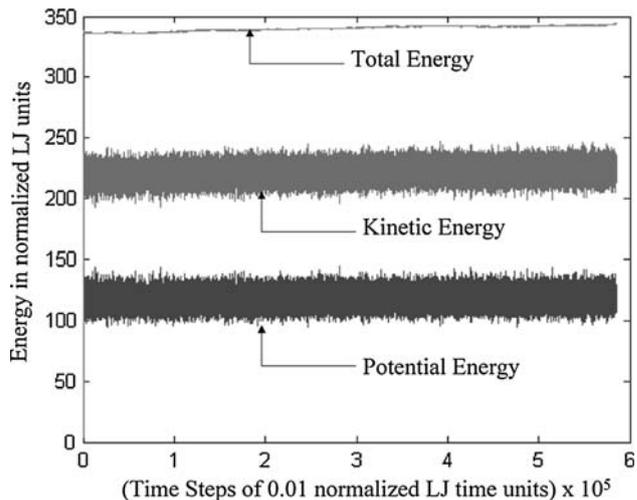


Fig. 13 Energy conservation over long temporal scales

tional multibody dynamics simulation tool. The present capabilities of POEMS have been discussed to some extent in the previous sections. For rigid body articulated systems there are three algorithms presently functional within the package. At present, the software has been implemented and verified for chain and tree systems. An immediate extension of these algorithms is to implement capabilities to handle systems with kinematically closed loops by implementing the algorithm presented in [13]. Currently, another key focus in the development of POEMS is its ability handle flexible body dynamics using the algorithm described in [14]. Similarly, at a later point capabilities to handle impact through collision detection and impact-momentum formulation will be added to POEMS. Sensitivity analysis [15] using direct differentiation is another area of interest and algorithms for the same are an upcoming feature in POEMS. One of the core applications of this package is in multi-scale molecular dynamics. For this application,

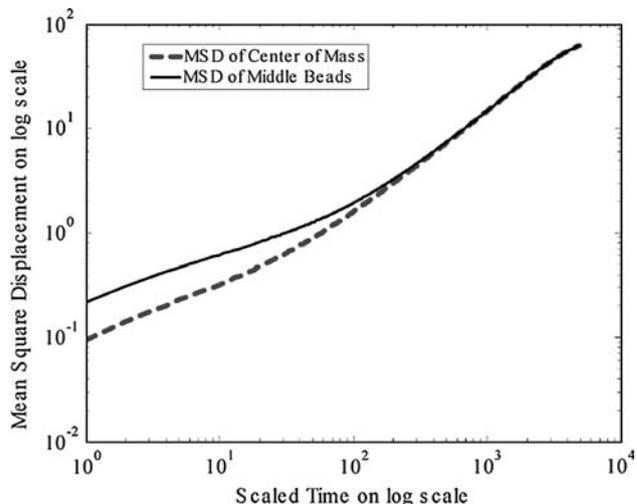


Fig. 14 Log-Log plot of mean square displacements of center of mass and middle beads

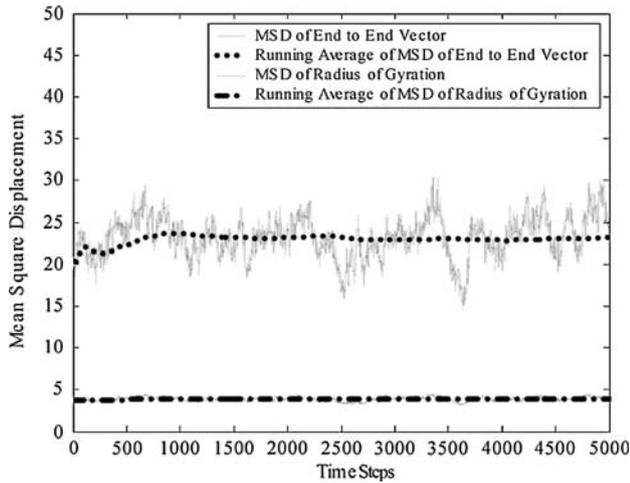


Fig. 15 Mean square end-to-end vector and radius of gyration

the POEMS package will be extended to include several multi time scale (MTS) integrators. POEMS is also being developed as an additional library to be added to the LAMMPS [16] code being developed by Sandia national laboratories to add to the capabilities of LAMMPS as a rigid body articulated system solver.

With the addition of these capabilities, POEMS will continue to grow in its abilities as a backbone research software package and serve as a new powerful tool to aid researchers and educators alike in the world of multi-body dynamics.

Acknowledgments The funding for this work was provided by NSF NIRT 0303902, NSF State-Time 0219734, NSF CAREER 9733684 and the funding provided by Sandia national laboratories (document number 220699) for this project. The authors would like to thank the funding agencies.

References

1. <http://www.mscsoftware.com>
2. <http://www.autolev.com>
3. <http://www.dynasim.se>
4. Masarati P, Moradini M, Quaranta G, Mantegazza P (2003) Open-source multibody analysis software. In: Conference proceedings, IDMEC/IST, Lisbon, July, 1–4
5. <http://www.opensource.org>
6. Kane TR, Levinson DA (1985) Dynamics: theory and application. McGraw-Hill, New York
7. Anderson KS (1993) An order-n formulation for the motion simulation of general multi-rigid-body tree systems. *Comput Struct* 46(3):547–559
8. Featherstone R (1999) A divide-and-conquer articulated body algorithm for parallel $O(\log(n))$ calculation of rigid body dynamics. Part 1: basic algorithm. *Int J Robot Res* 18(9):867–875
9. Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. *J Comput Phys* 73:325–348
10. Brooks BR, Bruccoleri RE, Olafson BD, States DJ, Swaminathan S, Karplus M (1983) Charmm: a program for macromolecular energy, minimization, and dynamics calculations. *J Comput Chem* 4:187–217
11. <http://www.mathworks.com>
12. Chun HN, Padilla C, Chin D, Masakatsuwatanabe, Valeri K, Alper H, Soosaar K, Blair K, Becker O, Caves L, Nagle R, Haney D, Farmer B (2000) MBO(N)D: a multibody method for long-time molecular dynamics simulations. *J Comput Chem* 21(3):159–184
13. Mukherjee R, Anderson KS (2005) An orthogonal complement based divide-and-conquer algorithm for constrained multibody systems. *Nonlin Dynam* (in press)
14. Mukherjee R, Anderson KS (2005) A logarithmic complexity divide-and-conquer algorithm for multi-flexible articulated body systems. *Comput Nonlin Dynam* (in press)
15. Bhalerao KD, Mukherjee RM, Anderson KS (2005) A divide and conquer direct differentiation approach for multibody system sensitivity analysis. In: Eleventh conference on nonlinear vibrations, stability, and dynamics of structures,
16. <http://www.cs.sandia.gov/sjplimp/lammps.html>