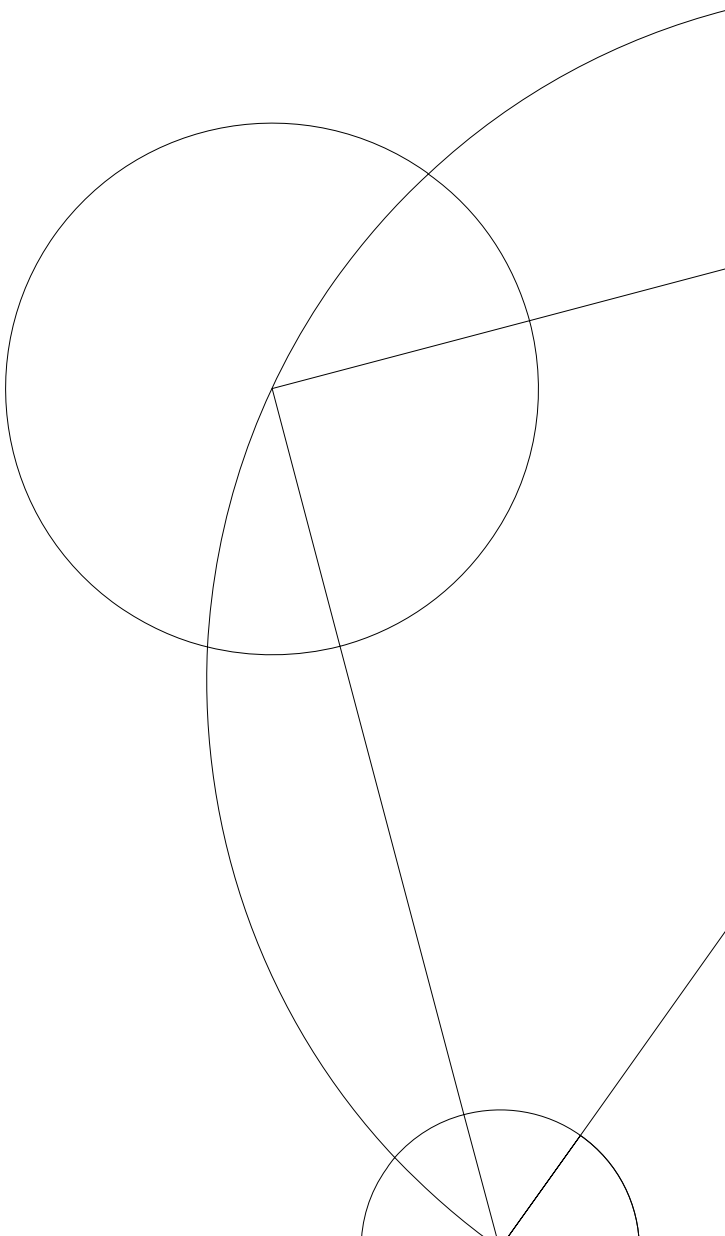
Master's Thesis

Tjørn Lynghus

Data Parallel Rigid Body Dynamics



Supervisor: Troels Henriksen

Datalogisk Institut. May 31, 2026

Contents

1 Abstract

Rigid body dynamics describe and compute the reactions to a given force (forward dynamics) and the required force to a given reaction (inverse dynamics) in a rigid body system. A rigid body system consists of one or many bodies connected by joints in a structure that resembles a tree. Tree structures are usually difficult to compute on in parallel. However, Blelloch's vector tree data structure enables parallel operations using associative operations on tree structures. In this thesis the vector tree data structure is used to parallelize the Recursive Newton Euler Algorithm (inverse dynamics) and the computation of the joint-space bias force and the joint-space inertia matrix of the Composite-Rigid-Body Algorithm. The Algorithms are implemented in the data parallel language Futhark and are shown to perform better than CPU implementations that use recursions for rigid body systems with over 10 000 bodies for the Recursive Newton Euler Algorithm which achieved a maximum speedup of 18x and 1000 bodies for the Composite-Rigid-Body Algorithm which achieved a maximum speedup of 163x.

2 Introduction

Rigid body dynamics are a tool used to analyze robotic systems and is used in e.g. spaceflight mechanics [1] and robot control [12]. Two common types of dynamics are forward and inverse dynamics. Both describe the relation between the forces acting on a rigid body system and the motion of the rigid bodies in the system. The inverse dynamics problem finds the forces acting on the bodies given the position, motion and accelerations. It then returns the torques, $\boldsymbol{\tau}$, being applied to the joints that connect the rigid bodies in the system. And the forward dynamics problem finds the acceleration of the bodies given their position, motion and the forces acting on them.

The connectivity of many rigid body systems can be represented by a tree. Computing the dynamics of systems with a large amount of bodies is computationally expensive. One way to speed up computation is to compute in parallel e.g. on a graphics processing unit (GPU). But, tree structures are usually not well suited for data parallelism on GPUs.

However, Blelloch has described a way to operate on trees using the Euler tour with the data parallel vector tree data structure [3]. One of the operations that the vector tree data structure provides is the rootfix operation which computes the propagation of some operator from the root to a node. If you imagine your body as a rigid body system and move your arm you may notice that your hand moves with. In this way the velocity is propagated from your arm to your hand in a similar way to how the rootfix operation works. If you now instead push your outstretched hand downwards you may feel that forces travel through your joints in your arm and into your shoulder joint. This is very much like how the vector trees' leaffix operation computes the propagation of an operator applied from the leaves of the tree to the root. Both the joint forces and the velocities of rigid bodies need to be computed by inverse dynamic algorithms like the Recursive Newton Euler Algorithm and forward dynamics algorithms like the Composite-Rigid-Body Algorithm. This gives an indication that the vector tree data structure might be a good fit for computing these algorithms.

Another property of vector trees is the fact that the rootfix and leaffix operations can be computed in a data parallel manner. The data parallel purely functional programming language Futhark [13] [14] is therefore a good candidate for exploring the relation between vector trees and rigid body dynamics.

In this thesis the relation between vector trees and rigid body dynamics is explored and dynamics algorithms using vector trees are successfully implemented in Futhark. The source code for the thesis

is publicly available and can be found using the following link: Code [1]. To describe the rigid body dynamics 6D spatial vectors are used since they result in simpler equations [8].

This thesis includes a description of preliminaries to Futhark. A theory chapter which includes the essentials for understanding the rigid body dynamics algorithms and the vector tree data structure. A description and an implementation of the Recursive Newton Euler Algorithm. A description and an implementation of the computation of the joint-space bias force and the joint-space inertia matrix of the Composite-Rigid-Body Algorithm. An experiments chapter that explores the implementations and the effect optimizations of scans and optimized data structures has on runtime. A conclusion. And a presentation of related works.

[1] The url to the code: https://github.com/Ritobusk/Data-Parallel-Rigid-Body-Dynamics

3 Preliminaries for Futhark

Futhark is a purely functional data parallel language and is therefore a good choice for implementing data parallel dynamics algorithms. It also supports sequential loops which enables the implementation of the original sequential dynamics algorithms which can be used as a comparison. Futhark can be compiled with many different backends such that the code can be compiled to be efficient parallel code for a GPU or efficient sequential code for a CPU. For this thesis the C, multicore and CUDA backends will be used.

3.1 Functions

This section details some of the most important second-order array combinators (SOACs) that can be compiled to data parallel code.

3.1.1 Map

Map has the type signature `map : (f : -> a -> b ) -> (xs : [n]a) -> [n]b`. It takes an array and a function and then applies that function to each element of the array. An example is adding a constant to the elements of an array: `map (+2) [1,2,3] = [3,4,5]`

3.1.2 Scatter

Scatter has type signature:
`scatter : (dest : *[k]t) -> (is: [n]i64) -> (vs: [n]t) -> *[k]t`. It takes in 3 arrays and uses the indices from `is` to insert the values from `vs` into the destination array. Here is an example: `scatter [0,1,2] [2,0] [50, 10] = [10, 1, 50]`

3.1.3 Reduce

Reduce has type signature: `reduce : (op : a -> a -> a) -> (ne: a) -> (as: [n]a) -> a`. It takes an array, an associative operator and a neutral element and returns the result of applying the operator over all the elements of `as`: `(((as[0] ‘op‘ as[1]) ‘op‘ as[2])...) ‘op‘ as[n-1])`. An example could be to get the product of the elements of an array:
`reduce (*) (1) [2, 5, 3] = ((1*2)*5)*3 = 30`

3.1.4 Scan

Scan has the same type signature as reduce except that it returns an array with type `[n]a` and functions in a similar way. However now all the intermediate applications of `op` are also returned. Using the example from reduce: `scan (*) (1) [2,5,3] = [2, 10, 30]`

Another function that is also often used is exclusive scan which shifts the resulting array to the right by 1 and has the first element be the neutral element. The example from before becomes: `exscan (*) (1) [2,5,3] = [1, 2, 10]`

3.2 Time Complexity

The time complexity of parallel algorithms are often described by their work and span [4], where work is the total amount of computations an algorithm does and span (or depth) is the deepest sequential dependency. The asymptotics of the previously discussed SOACs can be seen in Table 1.

Function	**Work**	**Span**
Map	$O(N \times W(f))$	$O(S(f))$
Scatter	$O(N)$	$O(1)$
Reduce	$O(N \times W(op))$	$O(\log N \times W(op))$
Scan	$O(N \times W(op))$	$O(\log N \times W(op))$

Table 1: The work and span complexity of the SOACs.

Term	Description	Unit
Rigid Body	A solid object that does not deform.	
Spatial Vector	A 6 dimensional vector which can either be a motion vector or a force vector.	
Spatial Velocity	A motion vector that describes the velocity of a rigid body. The first three dimensions are dedicated to describing its rotational velocity and the last three describe its linear velocity.	Rotation: $\frac{radians}{s}$ Linear velocity: $\frac{m}{s}$
Spatial Acceleration	A motion vector that describes the rate of change in a rigid body's spatial velocity.	Rotation: $\frac{radians}{s^2}$ Linear velocity: $\frac{m}{s^2}$
Spatial Force	A force vector that describes the force applied to a rigid body. The first three dimensions describe the couple (rotational force) and the last three describe the linear force.	Couple: $\frac{kg \cdot m^2}{s^2}$ Linear Force : $\frac{kg \cdot m}{s^2}$
Frame	A frame describes a cartesian coordinate system in 3 dimensions. All spatial vectors are relative to some frame.	
Coordinate Transforms	A Plücker coordinate transform is a 6 by 6 matrix that transforms a spatial vector to be relative to another frame.	
Inertia	A spatial inertia is a 6 by 6 matrix that maps a spatial velocity to its momentum (which is a spatial force).	
Rigid Body Model	A model of one or more rigid bodies that are connected with joints. The connectivity can be described by a tree structure.	
Joints	The ith joint in a rigid body model describes the vector subspace that restricts the ith body's relative movement to its parent in the rigid body model.	
Dynamics	The effects of forces on rigid bodies.	

Table 2: A table of some of the terms used in the thesis.

4 Theory

This chapter will contain the theory necessary for understanding the rigid body dynamics algorithms. The algebra used to describe the algorithms will be based on spatial vectors [8] [9] [10] since they provide simpler and more understandable equations in comparison to regular 3-dimensional vectors. An explanation of vector trees is also provided since they are a crucial part of implementing rigid body dynamics in a data parallel fashion.

In Table 2 you can see an overview of some of the terms used in rigid body dynamics. These are described in more detail in this chapter.

As for notation a vector, $\mathbf{v}$, will be written in bold lowercase while a matrix, $\mathbf{M}$, will be written in bold uppercase. Scalars, s, will be written in italic lowercase. Dot notation is also used to show derivatives. The velocity of a point $\mathbf{p}$ is therefore $\dot{\mathbf{p}}$.

A spatial vector will always be related to some frame O and is therefore written as ${}^{O}\mathbf{v}$. However, sometimes it is convenient to write down equations using spatial vectors in a "coordinate free" notation. This means that you assume that all spatial vectors are defined relative to the same frame located at O. Because of this you can avoid adding superscripts to the spatial vectors and the spatial operations you want to do on them which results in simpler equations. In coordinate free notation ${}^{O}\mathbf{v}$ would be written simply as $\mathbf{v}$.

In this chapter there will be 3d and 6d vectors. A distinction in notation is therefore made such that 3d vectors are written with an underline as $\underline{\mathbf{v}}$ and spatial vectors are written as $\mathbf{v}$.

4.1 Spatial Velocity, Spatial Force & Plücker Coordinates

The spatial vectors that are used are Plücker coordinates and inhabit two vector spaces: $\mathbf{M}^6$ and $\mathbf{F}^6$. One is motion vectors which are in the vector space $\mathbf{M}^6$. These describe the motion of some body relative to a cartesian frame with origin in the point O. An example of a motion vector is a

velocity vector which describes the velocity of a body. The velocity vector is defined by an angular velocity (rotation) $\boldsymbol{\omega}$ and a linear velocity $\underline{\mathbf{v}_O}$.

To get a sense of how the velocity vector describes the motion of a body look at Figure 1. Here a body B is depicted with an arbitrary point P inside it. The body is moving with a linear velocity described by the 3-dimensional vector, $\underline{\mathbf{v}_P}$, that goes through P and an angular velocity, $\boldsymbol{\omega}$, (also a 3d vector) that also goes through P. The units for the individual elements of these two vectors are meters pr. second and radians pr. second, respectively. Now this movement should be described by a spatial velocity in the cartesian frame with the body fixed point that currently coincides with O (the origin of the frame). The angular velocity stays the same. However, the linear velocity changes such that:

$$\underline{\mathbf{v}_O} = \underline{\mathbf{v}_P} + \underline{\boldsymbol{\omega}} \times \vec{OP} \tag{1}$$

Here $\vec{OP}$ is the 3d vector from O to P and $\times$ is the cross product. The spatial velocity of B is build from $\underline{\boldsymbol{\omega}}$ and $\underline{\mathbf{v}_O}$ and can be written as the 6d vector:

$${}^{O}\mathbf{v} = \begin{bmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\mathbf{v}_O} \end{bmatrix} \tag{2}$$

The superscript O indicates that this in frame O's coordinates.

The Plücker basis vectors of a spatial motion are therefore the rotation about each axis and the linear translation in each axis' direction:

$$\mathcal{D}_O = \{\mathbf{d}_{Ox}, \mathbf{d}_{Oy}, \mathbf{d}_{Oz}, \mathbf{d}_x, \mathbf{d}_y, \mathbf{d}_z\} \subset \mathbf{M}^6 \tag{3}$$

The basis vectors are also shown in Figure 2a.

Spatial forces are defined in a similar way using the force, $\underline{\mathbf{f}}$, acting on a body and the couple $\underline{\mathbf{n}_O}$ (the moment about O). The spatial force can therefore be described as the couples in the x, y and z directions of the coordinate with origin in O and the linear forces acting in each of these directions.

$${}^{O}\mathbf{f} = \begin{bmatrix} \underline{\mathbf{n}_O} \\ \underline{\mathbf{f}} \end{bmatrix} \tag{4}$$

The Plücker basis for spatial forces is show in Figure 2b and consist of:

$$\mathcal{E}_O = \{\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z, \mathbf{e}_{Ox}, \mathbf{e}_{Oy}, \mathbf{e}_{Oz}\} \subset \mathbf{F}^6 \tag{5}$$

$\mathbf{F}^6$ is the dual coordinate system of $\mathbf{M}^6$ and a scalar product is defined between them to connect them:

$$\mathbf{m} \cdot \mathbf{f} = \mathbf{f} \cdot \mathbf{m} = \mathbf{m}^T \mathbf{f} \tag{6}$$

The scalar product of a spatial force acting on a body and the spatial velocity of that body is equal to the power delivered onto the body by the spatial force. A byproduct of the relation in Equation 6 is that a motion vector will need different coordinate transforms, $\mathbf{X}$, and spatial cross products, $\times$, when compared with spatial forces. The coordinate transforms on spatial forces are:

$$\mathbf{X}^* = (\mathbf{X}^{-1})^T \tag{7}$$

Where $\mathbf{X}$ is the coordinate transform of a spatial motion. The spatial cross product on forces, $\times^*$, has a similar relation to $\times$.

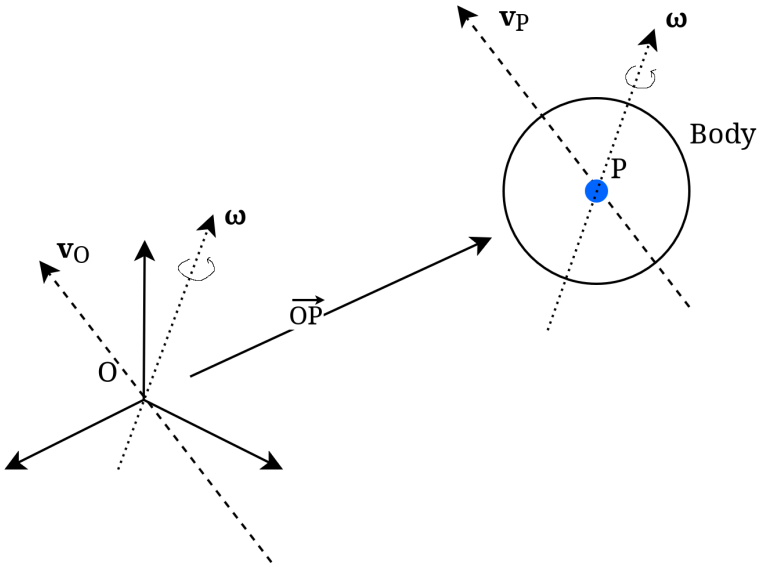
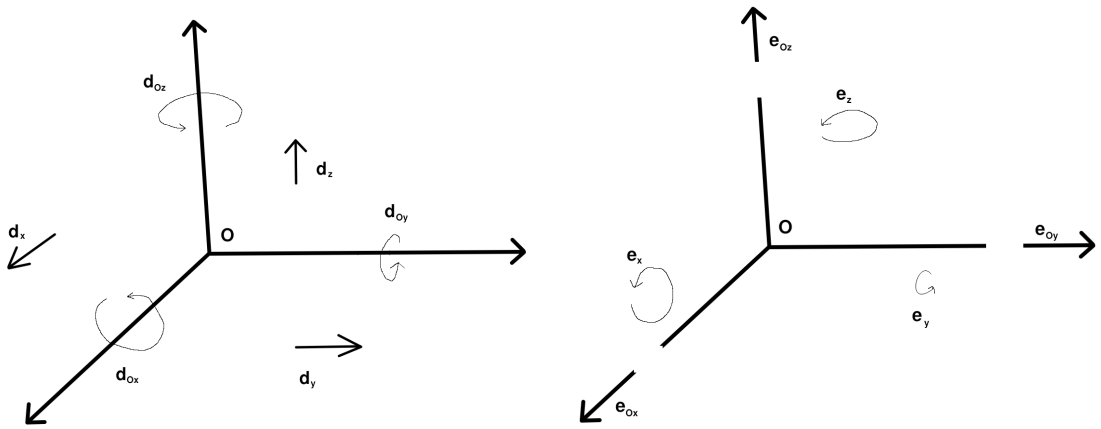


Figure 1: Showing how the velocity of a body B relates its spatial velocity in the coordinate system defined by the frame O_{xyz}. Only 3d vectors are depicted here so the underlines are omitted.



(a) The basis vectors of $\mathbf{M}^6$.

(b) The basis vectors of $\mathbf{F}^6$.

Figure 2: The Plücker basis vectors for spatial motion vectors and spatial force vectors. Together they are the 12 basis vectors for the Plücker coordinate system.

4.2 Plücker Coordinate Transformations

Since each spatial vector is defined relative to some frame it can be useful to transform the spatial vector to be relative to another frame. This is done with a transformation matrix ${}^{B}\mathbf{X}_{A}$:

$$ {}^{B}\mathbf{v} = {}^{B}\mathbf{X}_{A}\,{}^{A}\mathbf{v} \tag{8} $$

Here A and B are frames. The transformation consists of a rotation and a translation.

If frame A and B have the same origin but frame B is rotated with the 3×3 rotation matrix ${}^{B}\mathbf{E}_{A} = \mathbf{E}$ with respect to frame A then the coordinate transformation of the spatial motion ${}^{A}\mathbf{m}$ is:

$$ {}^{B}\mathbf{m} = {}^{B}\mathbf{X}_{A}\,{}^{A}\mathbf{m} = \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{E} \end{bmatrix}_{6\times 6} {}^{A}\mathbf{m} \tag{9} $$

Rotation matrices have the property that $\mathbf{E}^{-1} = \mathbf{E}^{T}$. The rotational coordinate transform for forces will therefore be equivalent because of Equation 7.

If on the other hand frame A and B have the same orientation but are located at O and P respectively you need to do a translation (Equation 1) to go from frame A to frame B:

$$ {}^{B}\mathbf{m} = \begin{bmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\mathbf{m}_{P}} \end{bmatrix} = \begin{bmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\mathbf{m}_{O}} - \underline{\mathbf{r}} \times \underline{\boldsymbol{\omega}} \end{bmatrix} = {}^{B}\mathbf{X}_{A}\,{}^{A}\mathbf{m} = \begin{bmatrix} \mathbf{1}_{3\times 3} & \mathbf{0}_{3\times 3} \\ -\underline{\mathbf{r}}\times & \mathbf{1}_{3\times 3} \end{bmatrix}_{6\times 6} {}^{A}\mathbf{m} \tag{10} $$

Here $\underline{\mathbf{r}} = \vec{OP}$ and $\underline{\mathbf{r}}\times$ is a 3×3 skew symmetric matrix that computes the cross product of $\underline{\mathbf{r}} \times \underline{\mathbf{a}}$ when multiplying with some $\underline{\mathbf{a}}$:

$$ \underline{\mathbf{r}}\times = \begin{bmatrix} 0 & -\underline{r}_{z} & \underline{r}_{y} \\ \underline{r}_{z} & 0 & -\underline{r}_{x} \\ -\underline{r}_{y} & \underline{r}_{x} & 0 \end{bmatrix} \tag{11} $$

If you combine the translation and rotation you get the general Plücker coordinate transformation matrix for spatial motions:

$$ {}^{B}\mathbf{X}_{A} = \begin{bmatrix} \mathbf{E} & 0 \\ 0 & \mathbf{E} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\underline{\mathbf{r}}\times & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{E} & 0 \\ -\mathbf{E}\underline{\mathbf{r}}\times & \mathbf{E} \end{bmatrix} \tag{12} $$

Below you can see how the transformation matrix changes depending on whether or not is applied to a force or if it is transforming from B to A:

$$ \begin{aligned} {}^{B}\mathbf{X}^{*}_{A} &= \begin{bmatrix} \mathbf{E} & 0 \\ 0 & \mathbf{E} \end{bmatrix} \begin{bmatrix} 1 & -\underline{\mathbf{r}}\times \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{E} & -\mathbf{E}\underline{\mathbf{r}}\times \\ 0 & \mathbf{E} \end{bmatrix} \\ {}^{A}\mathbf{X}_{B} &= \begin{bmatrix} 1 & 0 \\ \underline{\mathbf{r}}\times & 1 \end{bmatrix} \begin{bmatrix} \mathbf{E}^{T} & 0 \\ 0 & \mathbf{E}^{T} \end{bmatrix} = \begin{bmatrix} \mathbf{E}^{T} & 0 \\ \underline{\mathbf{r}} \times \mathbf{E}^{T} & \mathbf{E}^{T} \end{bmatrix} \\ {}^{A}\mathbf{X}^{*}_{B} &= \begin{bmatrix} 1 & \underline{\mathbf{r}}\times \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{E}^{T} & 0 \\ 0 & \mathbf{E}^{T} \end{bmatrix} = \begin{bmatrix} \mathbf{E}^{T} & \underline{\mathbf{r}} \times \mathbf{E}^{T} \\ 0 & \mathbf{E}^{T} \end{bmatrix} \end{aligned} \tag{13} $$

There are two other properties of Plücker coordinate transforms that are worth mentioning. The first is that you get the identity matrix if you:

$$I_{6\times 6} = {}^{A}\mathbf{X}_{B}\,{}^{B}\mathbf{X}_{A} = ({}^{B}\mathbf{X}_{A}^{-1})\,{}^{B}\mathbf{X}_{A} = {}^{B}\mathbf{X}_{A}\,{}^{A}\mathbf{X}_{B} = ({}^{A}\mathbf{X}_{B}^{-1})\,{}^{A}\mathbf{X}_{B} \tag{14}$$

The inverse of a Plücker coordinate transform is therefore easy to compute since you can easily get it from Equation 12 and Equation 13.

The second property is that you can chain together Plücker coordinate transforms to get to a desired frame. If you have frames: $\{1, 2, 3, \ldots, n\}$ you can transform from frame 1 to n:

$${}^{n}\mathbf{X}_{1} = {}^{n}\mathbf{X}_{n-1}\,{}^{n-1}\mathbf{X}_{n-2}\ldots{}^{3}\mathbf{X}_{2}\,{}^{2}\mathbf{X}_{1} \tag{15}$$

4.3 Acceleration

The spatial acceleration of a rigid body is defined as the time derivative of the rigid body's spatial velocity. If the spatial velocity of a rigid body is described as ${}^{O}\mathbf{v}$ in the cartesian frame with origin O then the spatial acceleration is defined by:

$${}^{O}\mathbf{a} = \frac{d}{dt}({}^{O}\mathbf{v}) = \frac{d}{dt}\begin{bmatrix}\underline{\boldsymbol{\omega}}\\ \underline{\mathbf{v}}_{O}\end{bmatrix} = \begin{bmatrix}\underline{\dot{\boldsymbol{\omega}}}\\ \underline{\dot{\mathbf{v}}}_{O}\end{bmatrix} = \begin{bmatrix}\underline{\dot{\boldsymbol{\omega}}}\\ \underline{\ddot{\mathbf{r}}} - \underline{\boldsymbol{\omega}} \times \underline{\dot{\mathbf{r}}}\end{bmatrix} \tag{16}$$

Here $\underline{\dot{\mathbf{r}}}$ is the velocity of a body-fixed particle that coincides with O at time t and $\underline{\ddot{\mathbf{r}}}$ is the acceleration of that particle. $\mathbf{v}_O$ can be thought of as the velocity of the stream of body-fixed particles that pass through O. $\underline{\dot{\mathbf{v}}_O}$ can be thought of as the rate of change of that stream.

In this way the spatial acceleration describes the acceleration of a rigid body and not just the acceleration of a point inside the rigid body.

4.4 Inertia

The inertia matrix of a rigid body is a 6×6 matrix which maps the rigid body's spatial velocity to its momentum, $\mathbf{h}$, where $\mathbf{h} \in \mathbf{F}^6$.

$$\mathbf{h} = \mathbf{I}\mathbf{v} \tag{17}$$

In this way it describes the force required to move a body given its inertia with a certain velocity. The inertia of a body in Plücker coordinates are a combination of its mass m, the vector to the body's center $\underline{\mathbf{c}}$ and its rotational inertia about its center $\mathbf{I}_c$ (a 3×3 matrix):

$$\mathbf{I} = \begin{bmatrix}\mathbf{I}_c + m\underline{\mathbf{c}} \times \underline{\mathbf{c}}\times^{T} & m\underline{\mathbf{c}}\times\\ m\underline{\mathbf{c}}\times^{T} & mI_{3\times 3}\end{bmatrix} \tag{18}$$

The Plücker transformation of a spatial inertia is defined as:

$${}^{B}\mathbf{I} = {}^{B}\mathbf{X}_{A}^{*}\mathbf{I}_{A}\,{}^{A}\mathbf{X}_{B} \tag{19}$$

Inertias also have the property that for N bodies that are rigidly connected then:

$$\mathbf{I}_{total} = \sum_{i=1}^{N} \mathbf{I}_i \tag{20}$$

The composite rigid body inertia of the N bodies is thus defined by $\mathbf{I}_{total}$.

You can also get the kinetic energy of a rigid body using its inertia and velocity:

$$E = \frac{1}{2}\mathbf{v} \cdot \mathbf{Iv} \tag{21}$$

4.5 Equation of Motion

The total force acting on a rigid body can be described by the rate of change of its momentum in coordinate free form:

$$\mathbf{f} = \frac{d}{dt}(\mathbf{Iv}) = \mathbf{Ia} + \mathbf{v} \times^* \mathbf{Iv} \tag{22}$$

Here $\mathbf{v}\times^*$ is the spatial cross product for forces. The spatial cross product is like a cross product in 3 dimensions but now for spatial vectors. The spatial cross product for a motion vector can be derived by taking the derivative of the basis vectors of $\mathbf{M}^6$ (Equation 3). A similar matrix to the one defined for 3d cross products (Equation 11) can be defined for the cross product of motion vectors:

$$\mathbf{v}\times = \begin{bmatrix} \underline{\boldsymbol{\omega}}\times & \mathbf{0}_{3\times3} \\ \underline{\mathbf{v}}_O\times & \underline{\boldsymbol{\omega}}\times \end{bmatrix}_{6\times6} \tag{23}$$

The spatial cross product of forces can be derived by taking the derivative of the basis vectors of $\mathbf{F}^6$ (Equation 5) and is related to Equation 23 by:

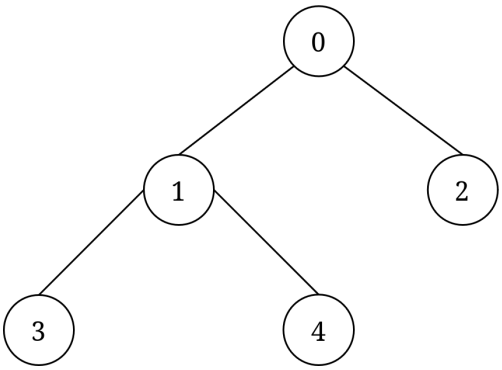
$$\mathbf{v}\times^* = -\mathbf{v}\times^T \tag{24}$$

Equation 22 is the equation which some of the dynamics algorithm take their basis from. E.g. the inverse dynamics algorithm the Recursive Newton Euler Algorithm. That is it computes the accelerations and velocities of the connected bodies such that the total force acting on a body can be found with Equation 22. This will be explained in more detail in section 5.

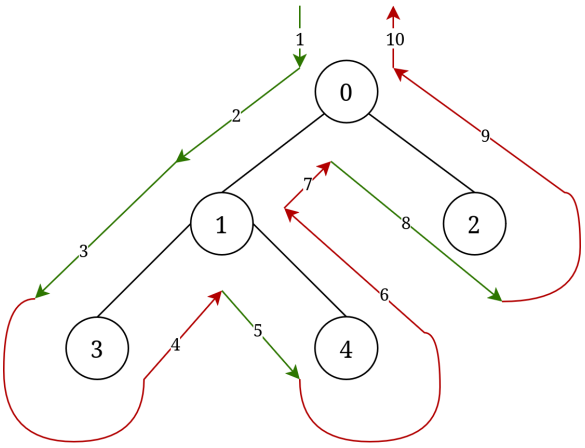
4.6 Vector Trees

The vector tree [3] (vtree) is a data parallel data structure that allows for parallel computations on tree structures using the scan function. The data structure consists of three arrays: The left and right parenthesis arrays (lp and rp) which are based on the Euler tour and contains the information of the connectivity of the tree. It also consists of a data array which contains the data of each node in the tree.

An example of the vtree data structure can be seen in Figure 3. Here you can see a tree where the nodes have their id as the data contained in the nodes. Figure 3b shows the Euler tour through this tree and also how it relates to the left and right parenthesis arrays shown in Figure 3c. As you



(a) Showing a vtree with each node having its index as its data.



(b) Showing the Euler tour of the vtree. The green arrows indicate that a node is visited for the first time. The red arrows indicate that the node is left for the last time.

1	2	8	3	5

lp

10	7	9	4	6

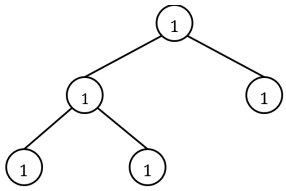
rp

0	1	2	3	4

data

(c) The contents of the vtree. That is: The lp array, the rp array and the data array.

Figure 3: This figure shows the vector tree data structure and the Euler tour that the left and right parenthesis arrays are made from.




```
def rootfix 'a [n] (op: a -> a -> a)
      (inv: a -> a) (ne: a)
      ({lp, rp, d}: t a [n]) : [n]a =
  let I = replicate (2 * n) ne
  let L = scatter I lp d
  let R = scatter L rp (map inv d)
  let S = exscan op ne R
  in map (\i -> S[i]) lp
```

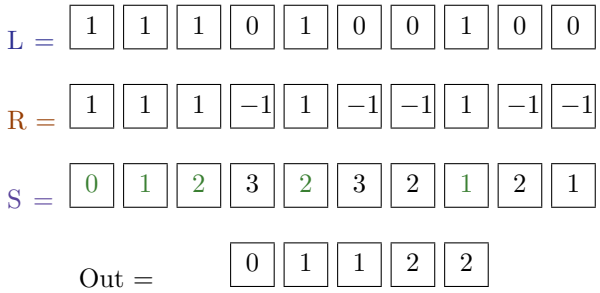


Figure 4: The implementation of rootfix and a walkthrough of how a rootfix operation can compute the depth of each node in a tree. Here the tree from Figure 3's data array is filled with 1s. The example shows how `rootfix (+) (\x -> -1 * x) 0 tree` would compute the depths where `tree` is the vtree shown in this figure.

can see the left parenthesis array contains the index of the step in the Euler tour when a node is first visited. And the right parenthesis array contains the index of the step in the Euler tour when a node is left for the last time.

There are two relevant operations that are useful in relation to dynamics and kinematics of rigid body systems. One is the rootfix operator which takes an associative binary operator, $\oplus$, and for each node computes the result of applying the operator on all of its ancestors. The other is the leaffix operator which also takes an associative binary operator and for each node returns the result of applying the operator to all the nodes in the subtree under it. Both of these operations also make use of an operator that takes an entry of the data array and computes the inverse in relation to the data and the $\oplus$ operator. This is because you want to erase the effects of applying the $\oplus$ operator to unrelated nodes. An example is if $\oplus$ is addition and the data array consist of integers then the inverse of adding an integer a to a sum is adding $-a$ to that sum.

The two operations are defined in the futhark code in Figure 4 and Figure 5. Here you can see that the rootfix operation is computed by populating an array of the size of the Euler tour with the data such that the indices from the lp array contains the data entries and the indices from the rp array contains the inverse of the data. The exclusive scan then walks the Euler tour. When a node is visited the data entry is applied to the accumulator. When a node is left for the last time its inverse data entry is applied to the accumulator to cancel the result of applying the data entry to the accumulator. Finally the lp entries are returned from the result of the exclusive scan. You can see how the depth of each node in a tree can be calculated with a rootfix operation in Figure 4.

The leaffix works by doing an exclusive scan on an array with the data entries at the indices of the lp array. Then the values at the indices of lp and rp are extracted to separate arrays where the values at the lp entries are inverted. Then you map over these arrays with $\oplus$ such that the inverse of the accumulated value when a node i is first visited is used to cancel out the applications to nodes that were visited prior to node i. The result is that for each node, i, the $\oplus$ is applied to all nodes in

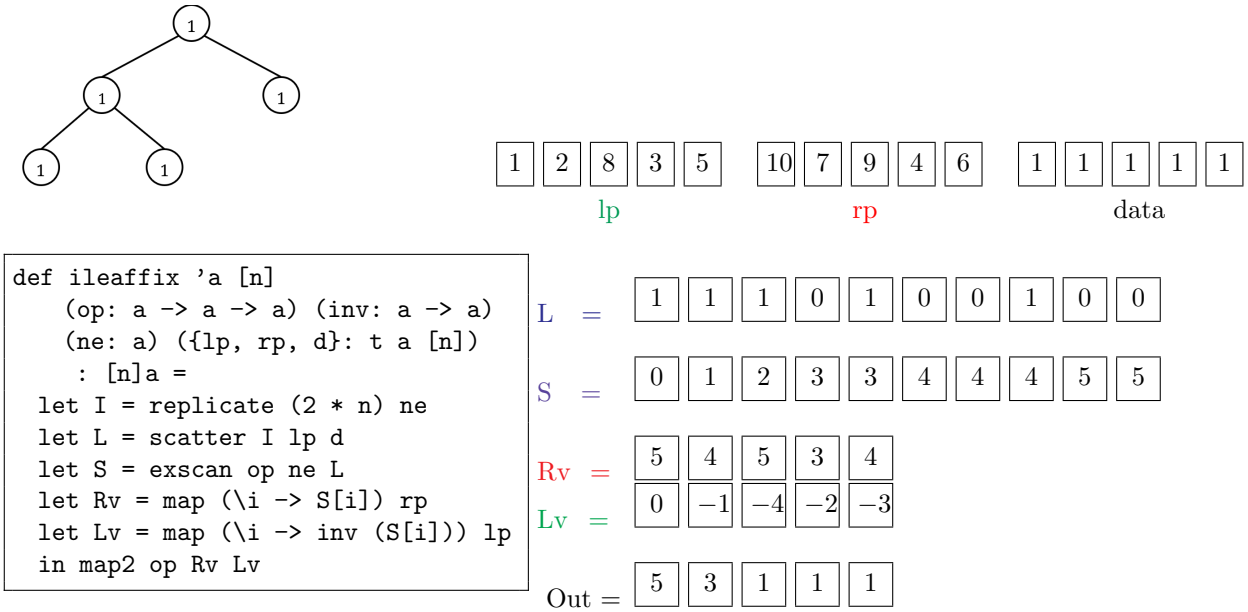


Figure 5: The implementation of ileaffix and a walkthrough of how a ileaffix operation can compute for each node the number of nodes in the subtree rooted at node i. Here the call: `illeaffix (+) (\x -> -1 * x) 0 tree` is being made.

the subtree under and including node i. You can see how you can compute for each node how many nodes that are in the subtree rooted at node i in Figure 5.

Both operations have a work complexity of $O(N \cdot W(\oplus))$ since the only operate over an array that is $2N$ long and a scan is used. The span of both operation is $O(\log N \cdot W(\oplus))$ also because of the scan.

The vtree library used in this thesis' implementations are taken from the library in [15].

4.7 The Model For Rigid-Body Systems

The model describes a rigid-body system that consists of rigid bodies and the joints between the rigid bodies which determines how the bodies move in relation to each other. The model is one of the inputs given to the dynamics algorithms. In this thesis for the sake of simplicity only models with a root that is fixed in place are considered without loss of generality. To describe the rigid-body system you need:

- Arrays that describe the connectivity of the system. This is typically a parent array, λ, which for each body points to its parent.
- The joint types, *joints*, of the joints that connect the parent of a body to the body. (That is one joint for each body.)
- The Plücker transformation from the parent's body to the place where the joint of the body is situated in the parent's body, $Xtree$. This means that each body i's joint is situated at one point in the parent and at one point in body i. This is showcased in Figure 7.

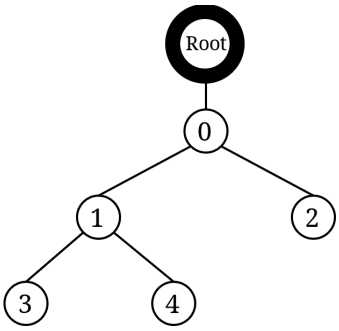


Figure 6: An example of a kinematic tree. The root has an ID of -1. The other nodes (the bodies) have an ID of 0 to $N-1$. You can see the ID of each node being the data inside it (except the root). The root is not actually a body but just a fixed point in space that is connected with a joint to body 0.

- The inertia matrix of each body, Is, which is in body i's coordinates: $Is(i) = {}^{i}\mathbf{I}$.

Each of these will be represented as arrays of size N where N is the number of bodies in the system.

The rigid-body system's connectivity can be represented by a kinematic tree which is a tree data structure of vertices representing the bodies and undirected edges representing the joints between the bodies. An example of such a tree can be seen in Figure 6. Here you can also see the data inside each node being the ID of that node. In general the child of a body will have an ID that is higher than its parent since some dynamics algorithms rely on this relation. This thesis will not consider closed kinematic chains which are rigid body models where there are loops in the connectivity.

The notation for indexing into these arrays will be done with e.g. $\lambda(i)$ which is the ith body's parent or e.g. $Is(i)$ being the ith body's inertia matrix.

4.7.1 Transformations and $Xtree$

Since $Xtree$ is needed one might suspect that at least N different frames are present in the model corresponding to the N different bodies. But there are in fact some additional frames that relate to the joints. This is because of the fact that body i can move in relation to its parent which means that the frame inside body i will have either rotated, translated or both from where it originally was depending on how body i's joint have moved. $Xtree(i)$ only describes the static transformation to body i's initial position from the parent's frame. There is therefore need for another transformation from this initial position to the current position which is dictated by the ith joint and how it has moved. There are therefore $2N+1$ frames in the model. The "plus one" is the root's frame.

Because of this the model is initially posed in such a way that the frame located in body i's parent coincides with the frame inside body i. This means that if none of the bodies have moved then $Xtree(i)$ is the complete coordinate transform from body $\lambda(i)$ to body i: $Xtree(i) = {}^{J_i}\mathbf{X}_{\lambda(i)} = {}^{i}\mathbf{X}_{\lambda(i)}$.

Figure 7 shows the different frames relevant for the Plücker transformation from body $\lambda(i)$ to body i (${}^{i}\mathbf{X}_{\lambda(i)}$). (Here Fs are used as notation for frames). Here joint i has been exploded such that you can see the frames better. To see why there are three frames you need to imagine how the ith joint connects body i with its parent. The connection makes it so the joint is situated at some point inside body $\lambda(i)$ and at some point inside body i. At the initial position these two points coincide.

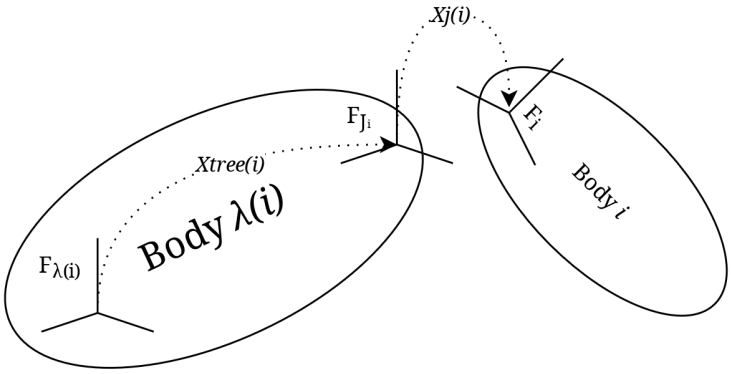


Figure 7: This figure shows how the ith joint is situated inside body $\lambda(i)$ and also inside body i. The Fs are frames. E.g. F_i is the ith body's frame. It also shows the Plücker transformations necessary to go from body $\lambda(i)$'s coordinates to body i's coordinates (the stippled arrows, $Xtree(i)$ and $Xj(i)$). Therefore ${}^{i}\mathbf{X}_{\lambda(i)} = Xj(i)Xtree(i)$.

The place the joint is situated inside body $\lambda(i)$ (the origin of $\mathrm{F}_{\mathrm{J}_i}$) does not change in relation to body $\lambda(i)$'s frame ($\mathrm{F}_{\lambda(i)}$). $Xtree(i)$ is therefore the transformation ${}^{\mathbf{J}_i}\mathbf{X}_{\lambda(i)}$. However the place the joint is situated inside body i does change in relation to body $\lambda(i)$. You therefore need to compute this relative transformation from the place the joint is situated inside body $\lambda(i)$, $\mathrm{F}_{\mathbf{J}_i}$, to the place it is situated inside body i which is F_i.

This calculation depends on the joint type and the positional change. For this thesis only joints with 1-DOF (1 degree of freedom) will be used. These joint types are:

- Revolute joints (joints which rotates about a given axis).
- Prismatic joints (joints which linearly transform along a given axis).
- Helical joints (joints which move in a screw motion given a pitch).

Many complicated joints are also able to be emulated by chaining together revolute and prismatic joints together with massless bodies. In Listing 1 you can see how you compute both the Plücker transformation ${}^{i}\mathbf{X}_{\mathbf{J}_i}$ and the vector subspace that the joint enforces (i.e. the way body i will move when its joint is moved). In general an n_i-DOF joint will have a subspace $\mathbf{S}_i$ of size $6 \times n_i$. Since only 1-DOF joints are used the subspace will be a 6×1 vector. The positional changes, velocities and accelerations ($\mathbf{q}_i$, $\dot{\mathbf{q}}_i$ and $\ddot{\mathbf{q}}_i$) of these joints are in general a $n_i \times 1$ vector. You are therefore able to simplify $\mathbf{q}_i$, $\dot{\mathbf{q}}_i$ and $\ddot{\mathbf{q}}_i$ to scalars.

Now ${}^{i}\mathbf{X}_{\lambda(i)}$ can be calculated:

$$ {}^{i}\mathbf{X}_{\lambda(i)} = {}^{i}\mathbf{X}_{\mathbf{J}_i}\,{}^{\mathbf{J}_i}\mathbf{X}_{\lambda(i)} = Xup \tag{25} $$

Xup is the name used for ${}^{i}\mathbf{X}_{\lambda(i)}$ in the code in the dynamics algorithm implementations.

4.7.2 Connectivity

The connectivity of the kinematic tree is often represented with a parent array, λ, where $\lambda(i)$ is the parent of the body with ID i.

Listing 1: jcalc: Computing the transformation to body i's coordinates and the vector subspace that the joint enforces. Here #Rx, #Ry and #Rz are revolute joints. #Px, #Py and #Pz are prismatic joints. And #helical is a helical joint. rotx, roty and rotz are functions that return a Plücker transformation matrix which only does transformation about the x, y and z axes respectively. xlt is a function that returns a Plücker transformation which translates from the current frame given a 3D vector.

```
1  def jcalc (jtyp : jointT) (q : f64) : ([6][6]f64, [6]f64) =
2    match jtyp
3        case #Rx -> (rotx q, [1f64,0,0,0,0,0])
4        case #Ry -> (roty q, [0,1f64,0,0,0,0])
5        case #Rz -> (rotz q, [0,0,1f64,0,0,0])
6        case #Px -> (xlt [q,0,0], [0,0,0,1f64,0,0])
7        case #Py -> (xlt [0,q,0], [0,0,0,0,1f64,0])
8        case #Pz -> (xlt [0,0,q], [0,0,0,0,0,1f64])
9        case #helical pitch ->
10         (rotz q `matrix_multiplication` (xlt [0, 0, q * pitch]), [0,0,1,0,0,pitch])
```

Some other connectivity arrays that can be useful are:

- μ which contains the children of each body.
- κ which for $\kappa(i)$ contains the path from the root to body i i.e. the IDs of the bodies on the path from the root to body i.
- v which for $v(i)$ contains all the IDs of the bodies in the subtree rooted at body i.

These arrays are useful when describing the algorithms in the dynamics chapters but not all are necessary for the actual implementation. Usually the algorithms just make use of the parent array, λ. However, in the data parallel implementations of this thesis the vector tree will describe the connectivity (i.e. the Euler tour). The vtree library used in this thesis contains a handy `mk_parent` function which takes a parent array and creates the left and right parenthesis arrays that were described in subsection 4.6.

A detail that is worth keeping in mind is how the parent array given to `mk_parent` needs one of the nodes to be the root. If you look at Figure 6 the root is not actually a body. This means that if you want a kinematic tree where more than one node's parent is the root you will need to insert an extra "root node" in the vector tree.

This thesis' implementation of the Composite-Rigid-Body Algorithm can not only settle for the vtree and needs the connectivity described by κ. One way to compute the κ array using a parent array and a vtree is shown in Listing 2.

This computes for each body i the IDs on the path from the root to body i not including body i's ID, i. The representation is a flat array to save space. It therefore also includes an array which keeps track of which body the paths entry belongs to. κ is static and can therefore be computed once and for all. This means that you do not have to compute this when computing the dynamics algorithms.

Listing 2: Computing $\kappa(i)$. Here `T.depth` is a function from the vtree library which computes the depth of each node (in the same way as was shown in Figure 4). `replicated_iota` takes the depths and creates a flat array containing iotas for each node's depth.

```
def paths [n] [k] (parent : [n]i64) (vtree : vtree) : ([k]i64, [k]i64) =
  let depths = T.depth vtree
  let tree_depth = reduce i64.max 0i64 depths
  let p_ii1 = replicated_iota depths

  let (_, paths) = map2 (\p d ->
       loop (j, path) = (p, replicate tree_depth (-1) with [0] = 0) for k < d do
              let path[k] = j
              let j = parent[j]
              in (j, path)
       ) parent depths
       |> unzip
  in (p_ii1,
    flatten paths
       |> filter (> -1i64))
```

5 Inverse Dynamics: The Recursive Newton Euler Algorithm

Inverse dynamics solves the problem of finding the forces that caused a given acceleration in a model of a multibody rigid body system. This is useful in e.g. motion control systems, mechanical design and as a component in the Composite Rigid-Body algorithm (forward dynamics). The inverse dynamics function is often represented as:

$$\boldsymbol{\tau} = \mathbf{ID}(model, \mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) \tag{26}$$

Here $\mathbf{q}, \dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ are each N motion vectors ($N \times 6$) that describe the positions, velocities and accelerations of the N rigid bodies in the model, respectively. These input variables are all expressed in body coordinate frame. That is, $\mathbf{q_i}$ describes the position of body i relative to body i's coordinates. Since the input is given in body coordinates the spatial vectors will in general be assumed to be in body coordinates unless specified otherwise. This means that e.g. the ith velocity ${}^{(body\ i)}\mathbf{v}_i$ (which is relative to body i's coordinate system as the superscript suggests) will be written as $\mathbf{v}_i$ instead regardless of whether or not coordinate free form is used. The *model* input is the one defined in subsection 4.7. $\boldsymbol{\tau}$ is the force vector (a vector of generalized force variables) which is also of size $N \times 6$. $\boldsymbol{\tau}$ is also the output of the algorithm.

The external forces applied to the system ($\mathbf{f}^x$) can also be provided as an extra optional input. This can be used to model e.g. force fields or physical contact applied on a body in the system. These forces are assumed to be given in root coordinates as opposed to the other inputs.

One way to compute the inverse dynamics is with the Recursive Newton Euler Algorithm (RNEA) which has $O(N)$ time complexity.

5.1 The Recursive Newton Euler Algorithm

RNEA has three main steps:

1. Compute the velocities and accelerations of the rigid bodies.
2. Compute the forces acting on each rigid body from its velocity and acceleration.
3. Compute the forces that are transmitted through the joints.

The following equations in coordinate free form give an overview of how the above needs to be calculated:

$$\mathbf{v_i} = \mathbf{v}_{\lambda(i)} + \mathbf{s_i}\dot{q}_i \tag{27}$$

$$\mathbf{a_i} = \mathbf{a}_{\lambda(i)} + \overset{o}{\mathbf{s}}_i\dot{q}_i + \mathbf{s}_i\ddot{q}_i + \mathbf{v}_i \times \mathbf{s}_i\dot{q}_i \tag{28}$$

$$\mathbf{f}_i^B = \mathbf{I}_i\mathbf{a}_i + \mathbf{v}_i \times^* \mathbf{I}_i\mathbf{v}_i \tag{29}$$

$$\mathbf{f}_i^J = \mathbf{f}_i^B - \mathbf{f}_i^x + \sum_{i \in \mu(i)} \mathbf{f}_i^J \tag{30}$$

$$\boldsymbol{\tau} = \mathbf{s}_i^T\mathbf{f}_i^J \tag{31}$$

In equations 27 to 31 $\lambda(i)$ refers to the parent of body i and $\mu(i)$ refers to the children of body i. To get a physical intuition for why the velocity and acceleration of the $\lambda(i)$ affects body i you

can try and move your arm and see how your hand (which is attached to your arm through a joint) moves with it.

The equations in body coordinates are only slightly more complicated. $\mathbf{v_i}$ (Equation 27) becomes:

$$\mathbf{v_i} = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{v}_{\lambda(\mathbf{i})} + \mathbf{s_i}\dot{q}_i \quad (\mathbf{v_0} = 0) \tag{32}$$

$\mathbf{a_i}$ is the derivative of $\mathbf{v_i}$ and is therefore:

$$\mathbf{a_i} = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{a}_{\lambda(\mathbf{i})} + \overset{o}{\mathbf{s}}_i\dot{q}_i + \mathbf{s_i}\ddot{q}_i + \mathbf{v_i} \times \mathbf{s_i}\dot{q}_i \quad (\mathbf{a_0} = -a_g) \tag{33}$$

Here $\overset{o}{\mathbf{s}}_i$ is the apparent derivative of $\mathbf{s_i}$ in body coordinates. However since the joint types that are used by the model (as described in subsection 4.7) have $\overset{o}{\mathbf{s}}_i = 0$ the acceleration equation can be simplified to:

$$\mathbf{a_i} = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{a}_{\lambda(\mathbf{i})} + \mathbf{s_i}\ddot{q}_i + \mathbf{v_i} \times \mathbf{s_i}\dot{q}_i \quad (\mathbf{a_0} = -a_g) \tag{34}$$

In Equation 34 you can see that the base case has been set to be $-a_g$ which is a gravity constant. $-a_g$ is considered to be part of the input *model* and is an efficient way of computing the effects of a uniform gravitational field's effect on the system. The field's effect is then propagated throughout the system through the recurrence. The alternative would be to provide the effects of gravity on each body with the help of an external force ($\mathbf{f}^x$) for each body.

Equation 29 is given by the equations of motion and does not change when when computing the result in body coordinates. This also applies to Equation 31 since $\mathbf{s}_i$ and $\mathbf{f}_i^J$ are already in body coordinates.

The joint forces (Equation 30) are related to the body forces in the following way:

$$\mathbf{f}_i^B = \mathbf{f}_i^J + \mathbf{f}_i^x - \sum_{j \in \mu(i)} \mathbf{f}_j^J \tag{35}$$

This is also evident from Figure 8. $\mathbf{f}_i^J$ is then defined by manipulating this equation. In body coordinates it becomes:

$$\mathbf{f_i^J} = \mathbf{f_i^B} - {}^{i}\mathbf{X}_0^*\mathbf{f_i^x} + \sum_{j \in \mu(i)} {}^{i}\mathbf{X}_j^*\mathbf{f_j^J} \tag{36}$$

Now all the definitions necessary to implement RNEA have been presented.

5.2 Implementing the Recursive Newton Euler Algorithm

The futhark code for the implementation can be seen in Listing 3, Listing 4 and Listing 5. To make the code more readable '*' has been used to represent matrix matrix multiplication, matrix vector multiplication, the scalar product of motion and force and scalar vector multiplication depending on the input to '*'. '+' is vector vector addition.

Another thing to remember from subsection 4.7 is because of how only 1-DoF joint types are used in the *model* it is possible to represent the input position, velocity and acceleration ($\mathbf{q}, \dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$) as an array of scalars instead of an array of motion vectors. To get the vector representation for each joint position, velocity or acceleration you can map over the joints and use the vector subspace

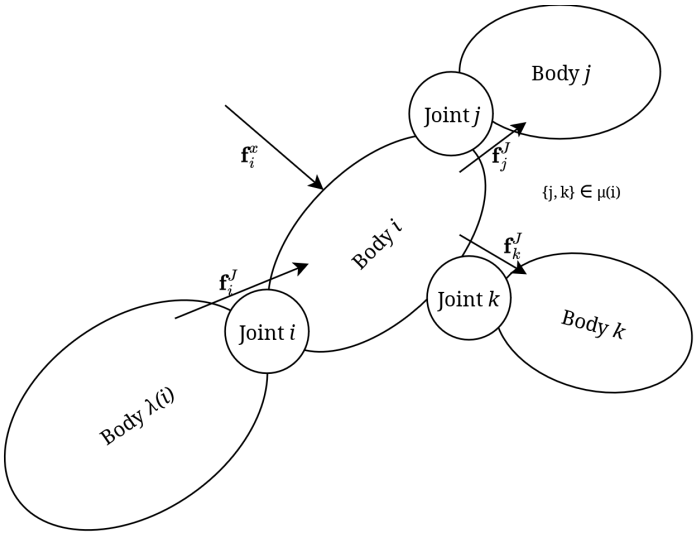


Figure 8: RNEA: The figure depicts the forces acting on body i. Here $\mathbf{f}_i^J$, $\mathbf{f}_j^J$ and $\mathbf{f}_k^J$ are the joint forces of their respective bodies. Because of Newton's third law the force exerted on body i must be: $\mathbf{f}_i^B = \mathbf{f}_i^J + \mathbf{f}_i^x - \mathbf{f}_j^J - \mathbf{f}_k^J$

returned by *jcalc* in Listing 1 and multiply it with either $\mathbf{q}, \dot{\mathbf{q}}$ or $\ddot{\mathbf{q}}$. This is also the reason why the $\boldsymbol{\tau}$ that is returned in Listing 3 is an array of scalars.

The first of the main steps of RNEA is computed by `vs_and_as` in Listing 4. The second step is done on line 9-10 of Listing 3. The final step is computed by `joint_forces` in Listing 5.

The input to the RNEA implementation in Listing 3 is as follows: The first four input parameters correspond to the *model* in Equation 26 where the fourth input is the gravity which is propagated through the bodies when computing the accelerations. The next three inputs (q, qd and qdd) are the position, velocity and acceleration. These are equivalent to Equation 26 (except the fact that they are arrays of scalars instead of vectors as previously mentioned). The final two arguments are the left and right parenthesis arrays used by the vtrees.

5.2.1 Step 1: Computing velocities and accelerations

In Listing 4 you can see how the computation of the velocities and accelerations are implemented.

To compute the velocities and accelerations you need to first compute the transformation matrices from $\lambda(i)$ to i's coordinates. This can be done using *jcalc* and the *model*'s *Xtree* as described in Equation 25 (line 5 and 7). You can also compute the isolated relative velocities and accelerations of the joints (line 6 and 20).

These computations are all easy to compute in a data parallel manner since they are independent of the other bodies in the *model*. However since Equation 32 and Equation 34 are recurrences another approach needs to be used. This is where the vector tree data structure is helpful. This is because the rigid bodies in the *model* form a kinematic tree which can be represented by the vector tree.

First the velocities need to computed. Since $\mathbf{s_i}\dot{q}_i$ has already been calculated in line 6 Equation 32 can be simplified to:

Listing 3: Structure of the data parallel RNEA implementation.

```
def rnea [n] (joint_types : [n]jointT) (Is : [n][6][6]f64) (Xtree : [n][6][6]f64)
             (gravity : [6]f64) (q : [n]f64) (qd : [n]f64) (qdd : [n]f64)
             (lp : [n]i64) (rp : [n]i64) : [n]f64 =
  -- Step 1: Compute the velocities and accelerations
  let (Xup, S, vs, as, transformation_tree) =
    vs_and_as joint_types Is Xtree gravity q qd qdd lp rp

  -- Step 2: Compute the bodyforces from the velocities and accelerations
  let fBs = tabulate n
      (\i -> (Is[i] * as[i]) + (((crf vs[i]) * Is[i]) * vs[i]))

  -- Step 3: Compute the joint forces
  let fJs = joint_forces Xup fBs transformation_tree lp rp
  in map2 (*) S fJs -- Computing tau
```

$$\mathbf{v}_i = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{v}_{\lambda(i)} + \mathbf{vJ}_i \quad (\mathbf{v_0} = 0) \tag{37}$$

This is a first-order recurrence which should be computed with a rootfix operation. To use the rootfix you need to define:

1. An associative operator.
2. The neutral element of the operator.
3. A function that creates an inverse of the input data.

Blelloch describes in [2] how one can compute a first order recurrence with a scan using an operator $\bullet$ over an ordered set of pairs which in this instance have the form $c_i = [a_i, b_i]$. The recurrence itself has the form:

$$x_i = \begin{cases} b_0 & i = 0 \\ (x_{i-1} \otimes a_i) \oplus b_i & 0 < i < n \end{cases} \tag{38}$$

The $\bullet$ binary associative operator does the following:

$$c_i \bullet c_j = [c_{i,a} \odot c_{j,a}, (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}] \tag{39}$$

Here $\oplus$ is associative, $\otimes$ is semiassociative such that there is an associative operator $\odot$ with the properties: $(a \otimes b) \otimes c = a \otimes (b \odot c)$ and $\otimes$ distributes over $\oplus$.

In the case of calculating the velocities it is clear that $c_i = [{}^{i}\mathbf{X}_{\lambda(i)}, \mathbf{vJ}_i]$ when comparing equations 37 and 38. Now you can define another ordered set that will be used as the accumulator in the scan that is used by the vtree: $s_i = [y_i, \mathbf{v}_i]$. Here y_i is:

$$y_i = \begin{cases} {}^{0}\mathbf{X}_{\lambda(0)}, & \text{if } i = 0 \\ y_{i-1} \odot {}^{i}\mathbf{X}_{\lambda(i)}, & 0 < i < n \end{cases} \tag{40}$$

Listing 4: Step 1 of the RNEA implementation.

```
def vs_and_as [n] (joint_types : [n]jointT) (Is : [n][6][6]f64)
            (Xtree : [n][6][6]f64) (gravity : [6]f64)
            (q : [n]f64) (qd : [n]f64) (qdd : [n]f64) (lp : [n]i64) (rp : [n]i64)
            : ([n][6][6]f64, [n][6]f64, [n][6]f64, [n][6]f64, [n][6][6]f64) =
  let (XJ, S) = unzip <| map2 (jcalc) joint_types q
  let vJ = map2 (*) S qd
  let Xup = map2 (*) XJ Xtree

  let inv_op (ci : ([6][6]f64, [6]f64)) : ([6][6]f64, [6]f64) =
    let inv_cia = X_inv ci.0
    in (inv_cia, (-1) * (inv_cia * ci.1))

  let op (si : ([6][6]f64,[6]f64)) (ci : ([6][6]f64,[6]f64)) : ([6][6]f64,[6]f64)
    = (ci.0 * si.0, (ci.0 * si.1) + ci.1)

  let vtree_vs = T.lprp <| mkt lp rp (zip Xup vJ)
  let (transformation_tree, vs) =
    T.irootfix op inv_op (identity 6, replicate 6 0f64) vtree_vs |> unzip

  let S_qdd = tabulate n (\i -> qdd[i] * S[i])
  let v_cross_S_qd = tabulate n (\i -> if i == 0 then Xup[0] * (-1 * gravity)
                                       else (crm vs[i]) * vJ[i])
  let as_tmp = map2 (+) S_qdd v_cross_S_qd

  let vtree_as = T.lprp <| mkt lp rp (zip Xup as_tmp)
  let as = T.irootfix op inv_op (identity 6, replicate 6 0f64) vtree_as
           |> map (.1)

  in (Xup, S, vs, as, transformation_tree)
```

And $\mathbf{v}_i$ is the one from Equation 37. Here $\lambda(0) = root$. The ${}^{0}\mathbf{X}_{\lambda(0)}$ term is therefore the transformation from the roots coordinates to body 0's coordinates.

Currently the $\oplus, \otimes$ and $\odot$ are not know but using the $\bullet$ operator on s_i and c_j will give an indication as to what they should be:

$$s_i \bullet c_j = [y_i \odot {}^{j}\mathbf{X}_{\lambda(j)}, (\mathbf{v}_i \otimes {}^{j}\mathbf{X}_{\lambda(j)}) \oplus \mathbf{vJ}_j] \tag{41}$$

Because of how rootfix works with the Euler tour it is assumed that $i = \lambda(j)$. To make the second element of the pair be equal to Equation 37 it is obvious that $\oplus$ must be vector vector addition. Vector vector addition is also associative. $\otimes$ must also be a matrix vector multiplication but since the transformation matrix needs to be multiplied with the velocity the operator needs to be a "reverse" matrix vector multiplication which takes a vector $\mathbf{v}$ and a matrix $\mathbf{X}$ and computes $\mathbf{v} \otimes \mathbf{X} = \mathbf{X} * \mathbf{v}$. Matrix vector multiplication does distribute over $\oplus$ but it is not associative. However, it does have the semiassociative property since if $\odot$ is matrix matrix multiplication then:

$$\mathbf{X}_1 *_v (\mathbf{X}_2 *_v \mathbf{v}) = (\mathbf{X}_1 *_M \mathbf{X}_2) *_v \mathbf{v} \tag{42}$$

Here $*_v$ is matrix vector multiplication and $*_M$ is matrix matrix multiplication. If you look at the first element of the pair in Equation 41. You will again notice how the matrix matrix multiplication is applied in the "wrong" order. The result should be $y_i \odot {}^{j}\mathbf{X}_{\lambda(j)} = {}^{j}\mathbf{X}_{\lambda(j)} y_i$. Therefore $\odot$ is a "reverse" matrix matrix multiplication that takes a matrix $\mathbf{A}$ and $\mathbf{B}$ and returns $\mathbf{BA}$. Now all the operators have been found and since they have the required properties the $\bullet$ operator can be used. It is defined in the line 13 - 14 in Listing 4.

The neutral element is a pair consisting of a 6×6 identity matrix, $I_{6\times6}$, and a 6×1 vector filled with zeroes, $\mathbf{v}_{zeroes}$:

$$s_i \bullet ne = [y_i \odot I_{6\times6}, (\mathbf{v}_i \otimes I_{6\times6}) \oplus \mathbf{v}_{zeroes}] = [y_i, \mathbf{v}_i] = s_i \tag{43}$$

Now all that is left before the velocities can be computed with a rootfix operation is to define the inverse operator. To get an idea for how the inverse operator should look like one can imagine process of computing a leaf during the scan of the rootfix operator. Just before the leaf is processed you have an accumulated value s_i. The next two values are the leaf and the leaf's inverted data: $c_{i+1} = c_{leaf} = [{}^{i+1}\mathbf{X}_{\lambda(i+1)}, \mathbf{vJ}_{i+1}]$ and $c_{i+2} = c_{leaf}^{-1}$.

To see how one should transform the leaf to be the inverse one can look at what the scan's computation will be on these three values. First the leaf is applied to s_i:

$$s_{i+1} = s_i \bullet c_{i+1} = [y_i \odot {}^{i+1}\mathbf{X}_{\lambda(i+1)}, \mathbf{v}_i \otimes {}^{i}\mathbf{X}_{\lambda(i+1)} \oplus \mathbf{vJ}_{i+1}] \tag{44}$$

First the left side of the pair is considered where the first element of s_{i+1} is: $y_i \odot {}^{i+1}\mathbf{X}_{\lambda(i+1)}$. To remove the ${}^{i+1}\mathbf{X}_{\lambda(i+1)}$ term one can multiply the inverse of this transformation matrix. Since the matrix is a Plücker transformation matrix the inverse matrix can be easily obtained be transforming ${}^{i+1}\mathbf{X}_{\lambda(i+1)}$ to ${}^{\lambda(i+1)}\mathbf{X}_{i+1}$.

Continuing the scan's computation the next element that it comes across is the inverse of c_{leaf}'s data. This should result in:

$$s_{i+1} \bullet c_{leaf}^{-1} = s_{i+1} \bullet c_{i+2} = s_{i+2} = s_i \tag{45}$$

Now that the first element of c_{i+2} has been found one can see how Equation 45 looks with

$$c_{i+2} = [{}^{\lambda(i+1)}\mathbf{X}_{i+1}, x], \tag{46}$$

where x is the unknown value of the second element of c_{i+2}:

$$\begin{aligned} s_{i+2} &= s_{i+1} \bullet c_{leaf}^{-1} = s_{i+1} \bullet c_{i+2} && (47)\\ &= [(y_i \odot {}^{i+1}\mathbf{X}_{\lambda(i+1)}) \odot {}^{\lambda(i+1)}\mathbf{X}_{i+1}, (({\mathbf{v}}_i \otimes {}^{i+1}\mathbf{X}_{\lambda(i+1)}) \oplus \mathbf{vJ}_{i+1}) \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1} \oplus x] && (48)\\ &= [y_i \odot ({}^{i+1}\mathbf{X}_{\lambda(i+1)} \odot {}^{\lambda(i+1)}\mathbf{X}_{i+1}), ({\mathbf{v}}_i \oplus (\mathbf{vJ}_{i+1} \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1})) \oplus x] && (49)\\ &= [y_i, {\mathbf{v}}_i \oplus (\mathbf{vJ}_{i+1} \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1}) \oplus x] && (50) \end{aligned}$$

From this it is easy to solve for x such that the second element of $s_{i+2} = \mathbf{v}_i$: $x = -(\mathbf{vJ}_{i+1} \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1})$

$$\begin{aligned} &= [y_i, {\mathbf{v}}_i + (\mathbf{vJ}_{i+1} \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1}) \oplus -(\mathbf{vJ}_{i+1} \otimes {}^{\lambda(i+1)}\mathbf{X}_{i+1})] && (51)\\ &= [y_i, {\mathbf{v}}_i] && (52) \end{aligned}$$

The inverse operator therefore transforms $c_i = [{}^{i}\mathbf{X}_{\lambda(i)}, \mathbf{vJ}_i]$ to:

$$c_i^{-1} = [{}^{\lambda(i)}\mathbf{X}_i, -(\mathbf{vJ}_i \otimes {}^{\lambda(i)}\mathbf{X}_i)] \tag{53}$$

The inverse operator is defined on line 9-11 in Listing 4.

Finally the rootfix operation can be used to obtain the velocities (line 16 - 18). Here the *mkt* function gathers the left and right parenthesis arrays and the vtree's data into a struct such that the vtree can be made. The `transformation_tree` produced in line 17 (the y array from Equation 40) will be used in step 3 and explained later.

Luckily the accelerations can be computed in an identical manner after simplifying Equation 34 to:

$$\mathbf{a_i} = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{a}_{\lambda(\mathbf{i})} + \mathbf{aJ_i} \quad (\mathbf{a_0} = -a_g) \tag{54}$$

Where $\mathbf{aJ_i} = \mathbf{s_i}\ddot{q}_i + \mathbf{v_i} \times \mathbf{s_i}\dot{q}_i$. $\mathbf{aJ_i}$ is computed in line 20 - 23 and the accelerations are computed in line 25 - 27.

5.2.2 Step 2 & 3: Computing the forces

The body forces (step 2) can be computed with a single map now that the velocities and accelerations have been computed. This is done in line 8 - 9 of Listing 3.

The joint forces of Equation 36 are however a bit more complicated. If the external forces are ignored this recurrence is a higher-order recurrence. To solve this in a direct way with a leaffix operation and how Blelloch describes how higher-order recurrences can be solved will make the scan's operator and the data of the vtree large and cumbersome.

However, there is another way to go about the computation which makes use of the properties of Plücker coordinate transformations (Equation 15). The idea is to transform all forces to one coordinate system such that they can relate to one another and then transforming them back into body coordinates. The process would be as follows:

1. First you transform all the body forces to root coordinates.
2. Then you do a summing leaffix operation on the forces in root coordinates.

3. And finally you transform the sums back into body coordinates.

For $\mathbf{f}_i^J$ this can also be written as:

$$\mathbf{f}_i^J = {}^i\mathbf{X}_0(\sum_{j \in v(i)} {}^0\mathbf{X}_j \mathbf{f}_j^B) \tag{55}$$

This can also be thought of as reducing the problem the leaffix has to compute (Equation 36) to be in coordinate free form (Equation 30). To get a sense of why it works one can look at a small example to see what the joint force computation actually does.

Consider the open chain robot (a robot with an unbranching kinematic tree) with a kinematic tree: $A \to B \to C \to D$, where A, B, C and D are rigid bodies. Now see what the joint force computation results in for each of the bodies when computing Equation 36:

$$\mathbf{f}_D^J = \mathbf{f}_D^B \tag{56}$$
$$\mathbf{f}_C^J = \mathbf{f}_C^B + {}^C\mathbf{X}_D \mathbf{f}_D^B \tag{57}$$
$$\mathbf{f}_B^J = \mathbf{f}_B^B + {}^B\mathbf{X}_C \mathbf{f}_C^B + {}^B\mathbf{X}_C {}^C\mathbf{X}_D \mathbf{f}_D^B = \mathbf{f}_B^B + {}^B\mathbf{X}_C \mathbf{f}_C^B + {}^B\mathbf{X}_D \mathbf{f}_D^B \tag{58}$$
$$\mathbf{f}_A^J = \mathbf{f}_A^B + {}^A\mathbf{X}_B \mathbf{f}_B^B + {}^A\mathbf{X}_B {}^B\mathbf{X}_C \mathbf{f}_C^B + {}^A\mathbf{X}_B {}^B\mathbf{X}_D \mathbf{f}_D^B = \mathbf{f}_A^B + {}^A\mathbf{X}_B \mathbf{f}_B^B + {}^A\mathbf{X}_C \mathbf{f}_C^B + {}^A\mathbf{X}_D \mathbf{f}_D^B \tag{59}$$
$$\tag{60}$$

As one can see $\mathbf{f}_i$ is equal to transforming all the body forces of the bodies in the subtree under it to i's coordinate and then summing it. To see how the proposed idea is equivalent you can compute $\mathbf{f}_i^J$ again using Equation 55:

$$\mathbf{f}_D^J = {}^D\mathbf{X}_A({}^A\mathbf{X}_D \mathbf{f}_D^B) = \mathbf{f}_D^B \tag{61}$$
$$\mathbf{f}_C^J = {}^C\mathbf{X}_A({}^A\mathbf{X}_C \mathbf{f}_C^B + {}^A\mathbf{X}_D \mathbf{f}_D^B) = \mathbf{f}_C^B + {}^C\mathbf{X}_D \mathbf{f}_D^B \tag{62}$$
$$\mathbf{f}_B^J = {}^B\mathbf{X}_A({}^A\mathbf{X}_B \mathbf{f}_B^B + {}^A\mathbf{X}_C \mathbf{f}_C^B + {}^A\mathbf{X}_D \mathbf{f}_D^B) = \mathbf{f}_B^B + {}^B\mathbf{X}_C \mathbf{f}_C^B + {}^B\mathbf{X}_D \mathbf{f}_D^B \tag{63}$$
$$\mathbf{f}_A^J = \mathbf{f}_A^B + {}^A\mathbf{X}_B \mathbf{f}_B^B + {}^A\mathbf{X}_C \mathbf{f}_C^B + {}^A\mathbf{X}_D \mathbf{f}_D^B \tag{64}$$

As you can see this results in the same as applying the recurrence from Equation 36.

To compute the proposed method you first need the transformation to root coordinates. This can be done with a rootfix operator on the Xup transformations. This creates a "transformation tree" where each node is the transformation from root coordinates to body i's coordinate. The operation given to rootfix is therefore a matrix matrix multiplication which will result in node i's transformation being:

$$transformation\_tree_i = {}^i\mathbf{X}_{\mu(\dots)}(\dots({}^{\mu(\mu(0))}\mathbf{X}_{\mu(0)}({}^{\mu(0)}\mathbf{X}_0))) = {}^i\mathbf{X}_0 \tag{65}$$

Here it assumed that every child in this chain is $\in \kappa(i)$ where $\kappa(i)$ is the set of nodes on the path from the root to body i. As can be seen in Equation 65 it is important that the accumulated value is the one being multiplied by the child's transformation matrix. This $transformation\_tree$ just so happens to be the $\mathbf{y}$ array from Equation 40 (which was computed in line 16-18 of Listing 4). This means that it is already calculated and is a by product of computing the velocity recurrence using rootfix.

Listing 5: Computing the joint forces.

```
1  def joint_forces [n] (Xup : [n][6][6]f64) (fBs : [n][6]f64)
2                       (from_root_to_joint_M : [n][6][6]f64)
3                        (lp : [n]i64) (rp : [n]i64) : [n][6]f64 =
4    let to_root_F = map transpose from_root_to_joint_M
5    let from_root_F = map X_MtoF from_root_to_joint_M
6
7    let fBs_root = map2 (*) to_root_F fBs
8
9    let vtree_fjs_root = T.lprp <| mkt lp rp fBs_root
10   in T.ileaffix (+) ((-1) *) (replicate 6 0f64) vtree_fjs_root
11         |> map2 (*) from_root_F
```

Listing 6: A change to line 7 of the code in Listing 5 to account for external forces.

```
1  let fBs_root = map3 (\X_to_root f_bi f_xi -> X_to_root * f_bi + (-1 * f_xi))
2                      to_root_F fBs f_ext
```

The implementation of computing the joint forces (step 3) is shown in Listing 5.

Since the force transformation matrices and the inverse force transformation is related (as is shown in Equation 13) ${}^{i}\mathbf{X}_0$ is transformed with two maps to: $to\_root_i^F = {}^{0}\mathbf{X}_i^*$ and $to\_body_i^F = {}^{i}\mathbf{X}_0^*$. Then you transform all the body forces to root coordinates with a map.

Now a leaffix is used to sum up the relevant body forces for each node. This is done with vector vector addition which is associative. The inverse of adding a vector $\mathbf{v}_i$ to a sum is adding the negative of that vector to the sum: $-\mathbf{v}_i$. The result of this is then transformed back into body coordinates with a map. In this way the joint forces (Equation 36) are computed in line 3-11.

Once the joint forces are computed $\boldsymbol{\tau}$ can easily be computed with a map (line 11 of Listing 3).

All in all this method of computing the joint forces only does some maps, a rootfix and a leaffix which has work of $O(N)$ and span of $O(\log N)$ which does not alter the asymptotics of the algorithm.

5.2.3 External Forces

Since the convention for the external forces is that they are given as input in root coordinates and the fact that a transformation to body i's coordinates has already been found, it is easy to add the external forces. Normally you would need to compute the transformations of the external forces which is usually found with a recurrence. If the RNEA implementation should support having external forces as an input only line 7 of Listing 5 should be changed to what is shown in Listing 6 and `(f_ext : [n][6]f64)` should be given as an input.

5.2.4 Optimizations

As it turns out the scan operation used by the rootfix and leaffix operations is very expensive. Futhark's native scan implementation is designed for simple operators (e.g. addition) and does not perform well when given larger ones (e.g. the recurrence operator, $\bullet$, defined in line 13-14 of Listing 4 or the vector vector addition used to compute the joint forces.)

Listing 7: Pseudocode for the optimized implementation RNEA.

```
1  def rnea_optimized [n] (model) (q) (qd) (qdd) (lp) (rp) : [n]f64 =
2    -- Step 1: Compute the velocities and accelerations
3    let transformation_tree = rootfix with matrix matrix operator
4    let vJ_root = transfrom vJ to root
5    let vs_r = rootfix with vector vector addition
6    let vs = transform as_r to body coordinates
7
8    let as_tmp = transform as_tmp to root coordinates
9    let as_r = rootfix with vector vector addition
10   let as = transform as_r to body coordinates
11
12   -- Step 2: Compute the bodyforces from the velocities and accelerations
13   let fBs = compute body forces
14
15   -- Step 3: Compute the joint forces
16   let fJs = compute joint_forces fBs transformation_tree lp rp
17   in map2 (*) S fJs -- Computing tau
```

Therefore two other scan implementations are explored both of which have the same asymptotics as the native scan. They will be presented here and their performance will be explored in subsection 7.3. The futhark implementations of these scans can be found in the publicly available source code provided with this thesis. (Here)

The first scan variation is Blelloch's work efficient scan which is presented in [2]. It works by first doing an upsweep that computes the final entry of the resulting array along with some intermediate sums. This is computed with a loop of $\log N$ iterations (where N is the size of the input array). Then a downsweep uses the intermediate sums to compute the rest of the entries in the resulting array with another loop of $\log N$ iterations.

A downside of this implementation is that the input array needs to be padded such that its size is a power of 2.

The second scan variation is a blocked scan. It works by dividing the input array into blocks of a given size and then mapping a sequential scan over each block. Then the final entries of each block are combined with a scan over them to produce the sums that need to be carried out to the blocks. Finally a map is used to apply the sums to the entries inside the blocks.

The blocked scan's block size can have a big impact on performance. The best block size in terms of runtime is therefore explored in subsection 7.3.

A third scan that is only applicable to rootfix and leaffix operations that use vector vector addition is also explored. This "scan" is computed by mapping a scan over each of the 6 dimensions of a spatial vector. This can be done since vector vector additions operates on each dimension independently.

The performance of a rootfix on the recurrence operator, $\bullet$, from line 13-14 of Listing 4 was also compared to performance of a rootfix using vector vector addition in subsection 7.3. It was found that vector vector addition is over 6 times faster for larger inputs. This therefore inspired a change in the algorithm such that the accelerations and velocities are computed in a similar manner to how the joint forces are computed in Listing 5. The structure of the optimized implementation is shown

in Listing 7.

The new approach to computing the velocities includes first computing the "transformation tree" which was computed as a byproduct of computing the velocities in line 17-18 of Listing 4. This can be computed with a rootfix with a matrix matrix multiplication operator. Then the transformation tree was used to transform the `vJs` to root coordinates. This reduces the recurrence of the velocities Equation 37 to its coordinate free form (Equation 27). Now a rootfix with a vector vector operation can be used to compute the velocities in root coordinates. The velocities are then transformed back into body coordinates. This only produced a slight speedup of up to 27 percent for small inputs and around 6 percent for large inputs when compared to using the rootfix using the $\bullet$ operator.

The accelerations are computed in a similar way to the velocities by transforming them to root coordinates such that the acceleration equation (Equation 34) can be thought of as having been reduced to its coordinate free form (Equation 28).

One other thing that is worth mentioning is that it is possible to do many of the spatial operations in a less costly manner. This is done by using "optimized" data structures and optimized operations on these data structures to mimic the matrix matrix multiplication, matrix vector multiplications and vector vector additions of the regular 6d vectors and 6×6 matrices.

An example of this is when multiplying two Plücker coordinate transforms, $\mathbf{X}_1$ and $\mathbf{X}_2$, you multiply two 6×6 matrices which results in $6 \cdot 6 \cdot 6 = 216$ floating point multiplications and $6 \cdot 6 \cdot 5 = 180$ floating point additions. This can be reduced to 33 multiplications and 24 additions if you extract the rotational matrix $\mathbf{E}_{3\times3}$ from $\mathbf{X}_1$ and $\mathbf{X}_2$ and the translational vector $\underline{\mathbf{r}}$ (see how the transformations are defined: Equation 12). Using this you can represent the transformations with a tuple $\mathbf{X}_1 = (\mathbf{E}_1, \underline{\mathbf{r}}_1)$ and the product of two coordinate transforms can be represented as:

$$\mathbf{X}_1\mathbf{X}_2 = (\mathbf{E}_1\mathbf{E}_2, \quad \underline{\mathbf{r}}_2 + \mathbf{E}_2^T\underline{\mathbf{r}}_1) \tag{66}$$

An overview of such spatial operation optimizations can be found in table A.2 on page 244, table A.3 on page 245 and table A.4 on page 247 of [8].

Not all operations benefit in the same way as multiplying two Plücker coordinate transforms. And some such as adding motion vectors or force vectors are the same. However, the computational saving in multiplying two Plücker coordinate transforms is significant since this is the operator which computes transformation tree. The scan in the transformation tree's rootfix therefore has to do a lot less work. Since scans turned out to be a bottleneck for the implementation these optimized data structures and optimized operations produces a speedup of almost 6x on large inputs when compared to the same implementation that does not use these optimizations.

The effects the optimized data structures has on performance is explored in subsection 7.2.

5.3 Asymptotics of the Data Parallel RNEA Implementation

First step 1 is considered (Listing 4). This step includes some maps over the N bodies where some vectors and matrices of constant size are multiplied. (All the matrices that are operated on in the presented RNEA implementation are of size 6×6 and all vectors are of size 6×1. That is they are of constant size.) These therefore have work of $O(N)$ and a span of $O(1)$. The rootfix operations are done on arrays of size N. The $\bullet$ operator that is used consists of one matrix matrix multiplication, one matrix vector multiplication and one vector vector addition where the matrices and vectors are of constant size. The rootfix operations therefore have a work asymptotic of $O(N)$ and a span of $O(\log N)$.

Step 2 (Listing 3) is a map over the N bodies which does some matrix and vector multiplications. This has work of $O(N)$ and span of $O(1)$.

Step 3 (Listing 5) changes the "transformation tree" computed in step 1 with 2 maps. The changes take constant time. Finally there is a leaffix operation which uses a vector vector addition which takes constant time. The work is therefore $O(N)$ and the span is $O(\log N)$.

All in all the work of the data parallel implementation of RNEA using vtrees is $O(N)$ and the span is $O(\log N)$.

6 Forward Dynamics: The Composite-Rigid-Body Algorithm

The forward dynamics problem is the problem of finding the acceleration of the bodies in a *model* given the position and velocity of the bodies and the forces acting the bodies. It is presented as:

$$\ddot{\mathbf{q}} = \mathbf{FD}(model, \mathbf{q}, \dot{\mathbf{q}}, \mathbf{f}^x, \boldsymbol{\tau}) \tag{67}$$

One way to calculate this is using the Composite-Rigid-Body Algorithm (CRBA). The algorithm takes its roots in the equation of motion of a kinematic tree:

$$\boldsymbol{\tau} = \mathbf{H}\ddot{\mathbf{q}} + \mathbf{c}, \tag{68}$$

and can be split into three parts:

- Compute the joint-space bias force vector, $\mathbf{c}$ (where $\mathbf{c} = \mathbf{ID}(model, \mathbf{q}, \dot{\mathbf{q}}, \mathbf{0}, \mathbf{f}^x)$).
- Compute the joint-space inertia matrix, $\mathbf{H}$.
- Solve for $\ddot{\mathbf{q}}$ in: $\mathbf{H}\ddot{\mathbf{q}} = \boldsymbol{\tau} - \mathbf{c}$.

As in the inverse dynamics chapter $\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{f}^x$ and $\boldsymbol{\tau}$ are vectors of size $N \times 6$ (where N is the number of bodies in the *model*) and the *model* is the model for rigid-body systems described in subsection 4.7. The joint-space bias force vector, $\mathbf{c}$, is a $N \times 6$ vector and the joint-space inertia matrix, $\mathbf{H}$, is an $N \times N \times (6 \times 6)$ matrix of inertias.

If the *model* in question only uses joints with 1-DOF then the dimensions of the vectors can be simplified to $N \times 1$ and for $\mathbf{H}$ it can be simplified to $N \times N \times (1 \times 1)$.

As with RNEA $\mathbf{f}^x$ is an optional input.

The final step (solving for $\ddot{\mathbf{q}}$) will not be implemented for this thesis since it includes implementing a linear solver which is not the focus of the thesis. The asymptotics of the steps are $O(N)$ for computing $\mathbf{c}$ (since it is a call to RNEA), $O(N^2)$ for computing $\mathbf{H}$ and $O(Nd^2)$ for solving for $\ddot{\mathbf{q}}$ where d is the depth of the kinematic tree.

6.1 The Composite-Rigid-Body Algorithm: Computing H

A property of the joint-space inertia matrix, $\mathbf{H}$, is that the kinetic energy of the kinematic tree can be described with $\mathbf{H}$:

$$T = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \dot{\mathbf{q}}_i^T \mathbf{H}_{i,j} \dot{\mathbf{q}}_i \tag{69}$$

The kinetic energy of a kinematic tree is also the sum of its rigid body's kinetic energy (Equation 21):

$$T = \frac{1}{2} \sum_{k=1}^{N} \mathbf{v}_k^T \mathbf{I}_k \mathbf{v}_k \tag{70}$$

You can rewrite Equation 70 such that you can know what $\mathbf{H}_{i,j}$ should be.

First you substitute the velocity equation used when computing the inverse dynamics (Equation 27) in and get:

$$T = \frac{1}{2}\sum_{k=1}^{N}\sum_{i\in\kappa(k)}\sum_{j\in\kappa(k)}\dot{\mathbf{q}}_i^T\mathbf{S}_i^T\mathbf{I}_k\mathbf{S}_j\dot{\mathbf{q}}_j, \tag{71}$$

where $\kappa(i)$ is the bodies from the root to body i. This can be rewritten to:

$$T = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k\in v(i)\cap v(j)}\dot{\mathbf{q}}_i^T\mathbf{S}_i^T\mathbf{I}_k\mathbf{S}_j\dot{\mathbf{q}}_j, \tag{72}$$

where $v(i)$ is the subtree rooted at body i.

If you compare Equation 72 and Equation 69 you can see that the elements of $\mathbf{H}$ are:

$$\mathbf{H}_{i,j} = \sum_{k\in v(i)\cap v(j)}\mathbf{S}_i^T\mathbf{I}_k\mathbf{S}_j. \tag{73}$$

The physical reality of what the elements of $\mathbf{H}$ are is that $\mathbf{H}_{i,j}$ relates the acceleration at joint j to the force applied to joint i.

Now to get rid of the sum over the inertias a new inertia is defined:

$$\mathbf{I}_i^c = \mathbf{I}_i + \sum_{j\in\mu(i)}\mathbf{I}_j^c, \tag{74}$$

which is the inertia of the subtree under body i i.e. the composite rigid body of $v(i)$.

Now the elements of $\mathbf{H}$ can finally be described as:

$$\mathbf{H}_{i,j} = \begin{cases}\mathbf{S}_i^T\mathbf{I}_i^c\mathbf{S}_j & \text{if } i \in v(j)\\ \mathbf{S}_i^T\mathbf{I}_j^c\mathbf{S}_j & \text{if } j \in v(i)\\ \mathbf{0} & \text{otherwise}\end{cases} \tag{75}$$

You might also notice the way the connectivity affects the elements of $\mathbf{H}$ (due to the use of κ). To get a notion of how the $\mathbf{H}$ matrix will look you can look at Equation 75 and Figure 9. In the figure the two kinematic trees with different connectivities are depicted above their respective $\mathbf{H}$ matrix. The white colored spaces in the matrices are elements where $\mathbf{H}_{i,j} = \mathbf{0}$. The other spaces correspond to either case 1 or 2 of Equation 75. The nodes have been colored such that you can see how each node affects $\mathbf{H}$. From this it is evident that body i is responsible for row i and column i of $\mathbf{H}$.

To actually compute $\mathbf{H}$ you of course need to be write Equation 75 and Equation 74 in body coordinates. The composite rigid body inertias become:

$$\mathbf{I}_i^c = \mathbf{I}_i + \sum_{j\in\mu(i)}{}^i\mathbf{X}_j^*\mathbf{I}_j^{cj}\mathbf{X}_i \tag{76}$$

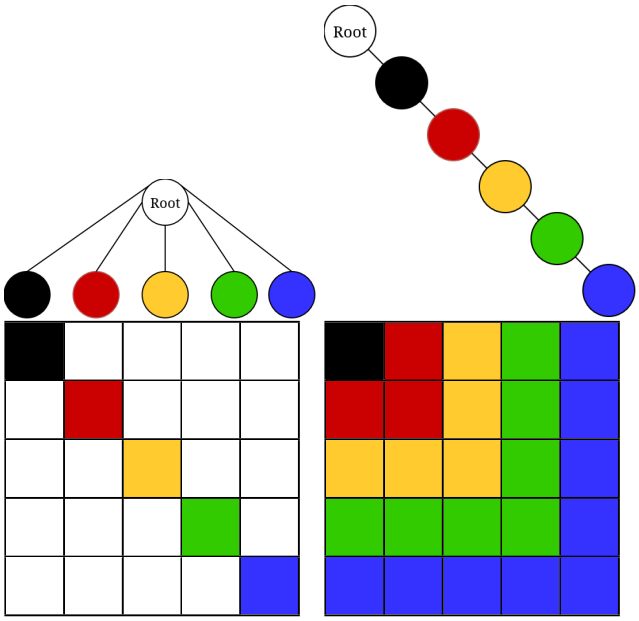


Figure 9: Shows the branch-induced sparsity of $\mathbf{H}$. Here the $\mathbf{H}$ matrix of an open chain rigid body system (right) is contrasted with the $\mathbf{H}$ matrix of a kinematic tree where the maximum depth of a node is 1 (left). The colors of in the table correspond to the nodes' colors. You can see how each node will fill out $\mathbf{H}$ depending on its connectivity.

The elements of $\mathbf{H}$ are:

$$\mathbf{H}_{ij} = \begin{cases} {}^{j}\mathbf{f}_i^T \mathbf{s}_j & \text{if } i \in v(j) \\ \mathbf{H}_{ji}^T & \text{if } j \in v(i) \\ 0 & \text{otherwise} \end{cases} \tag{77}$$

The ${}^{j}\mathbf{f}_i$ is $\mathbf{I}_i^c \mathbf{S}_i$ in body i's coordinates. ${}^{j}\mathbf{f}_i$ can be computed with:

$${}^{\lambda(j)}\mathbf{f}_i = {}^{\lambda(j)}\mathbf{X}_j^{*\,j}\mathbf{f}_i, \qquad ({}^{i}\mathbf{f}_i = {}^{i}\mathbf{I}_i^{c\,i}\mathbf{S}_i) \tag{78}$$

Now everything needed for computing the $\mathbf{H}$ matrix is in place.

6.2 Implementing the Composite-Rigid-Body Algorithm

This implementation will only focus on computing the $\mathbf{c}$ vector and $\mathbf{H}$ matrix. This is because the final step of CRBA is to solve Equation 68 which does not have anything to do with vector trees. The implementation can be seen in Listing 8, Listing 9 and Listing 10. '*' and '+' are again used as all-purpose multiplications and addition between matrices and vectors depending on the input as in subsection 5.2. Also because of how the *model* only uses 1-DOF joints many of the vectors and matrices can be represented as vectors and matrices of scalars. E.g. because of this the H matrix can be an $N \times N \times (1 \times 1)$ matrix of scalars instead of a matrix of inertias ($N \times N \times (6 \times 6)$).

The main steps of CRBA are outlined in Listing 8. Another thing to notice is that the algorithm takes the `paths` and `p_ii1` arrays as inputs. These are the paths from the root to each body i, κ, which was shown in subsection 4.7. These are used when computing $\mathbf{H}$.

Step 1 of CRBA is to compute the $\mathbf{c}$ vector. Since it is identical to computing rnea except that the terms which were multiplied with the accelerations `qdd` are omitted. Some of the intermediate computations of the rnea implementation from subsection 5.2 can be used when computing $\mathbf{H}$ and are therefore also returned. These are the "transformation trees" that contain the Plücker transformation matrices from root to body i and from body i to the root.

Step 2 which computes $\mathbf{H}$ along with the composite rigid body inertias is shown in Listing 9 and Listing 10 and is explained in subsubsection 6.2.1 and subsubsection 6.2.2 respectively.

Listing 8: Computing the H and C matrices. Similar to what is described in subsubsection 5.2.3 you can also give the external forces on the model as an input in step 1.

```
1  def crba_vtree [n] [nd] [ndd] (joint_types : [n]jointT) (Is : [n][6][6]f64)
2              (Xtree : [n][6][6]f64) (gravity : [6]f64)
3              (lp : [n]i64) (rp : [n]i64) (paths : [nd]i64) (p_ii1 : [ndd]i64)
4              (q : [n]f64) (qd : [n]f64)
5              : ([n]f64, [n][n]f64) =
6    -- Step 1: Compute the joint-space bias force: c = ID(model, q, qd, 0)
7    let (c, to_root_F, from_root_F, to_root_M, from_root_M)
8      = rnea joint_types Is Xtree gravity q qd 0 lp rp
9
10   -- Step 2: Compute H
11   let Ics = crb Is to_root_F from_root_F to_root_M from_root_M lp rp
12   let H = joint-space_inertia_matrix Ics S paths p_ii1 from_root_F to_root_F
13
14   in (c, H)
```

6.2.1 Step 2: Computing the Composite Rigid-Body Inertias

When looking at Equation 76 (which computes the composite rigid body inertias) it looks very much like the recursion that computed the joint forces (Equation 36). A similar approach can therefore be used. The only difference is that inertia matrices are transformed slightly differently as presented in Equation 19. And since the 'transformation trees' were already computed when doing step 1 these can be reused.

In Listing 9 you can see how the inertia matrices are mapped to root coordinates. Then they are added with a leaffix. As opposed to the joint forces which are vectors the inertias are matrices and the leaffix therefore uses matrix matrix addition instead of vector vector addition. The inverse of adding a matrix, $\mathbf{A}$, to a sum is adding $-\mathbf{A}$ to the same sum. Finally the composite rigid-body inertias are transformed back into body coordinates and returned in line 9.

6.2.2 Step 2: Computing the Joint-Space Inertia Matrix

What Equation 77 shows is that the ith row of the $\mathbf{H}$ matrix consists of the spatial force applied to the ith composite rigid bodies in link j's coordinates for every body j in the path between i and the root ($j \in \kappa(i) \setminus i$). (This might be a bit clearer when also looking at the branch-induced sparsity shown in Figure 9.) That is you need the path from the root to body i.

Listing 9: Computing the composite rigid body inertias.

```
1 def crb [n] (Is : [n][6][6]f64) (tr_F : [n][6][6]f64) (fr_F : [n][6][6]f64)
2             (tr_M : [n][6][6]f64) (fr_M : [n][6][6]f64)
3             (lp : [n]i64) (rp : [n]i64)
4             : [n][6][6]f64 =
5
6   let I_to_root = map3 (\bXa_F aXb_M I -> bXa_F * (I * aXb_M)) tr_F fr_M Is
7   let vtree_Ics_root = T.lprp <| mkt lp rp I_to_root
8   let Ics_root = T.ileaffix (+) ((-1) *) (replicate 6 <| replicate 6 0f64)
        ↪ vtree_Ics_root
9   in map3 (\bXa_F aXb_M rI -> bXa_F * (rI * aXb_M)) fr_F tr_M Ics_root
```

Listing 2 shows how these paths could be calculated into an irregular array, `paths`, along with another irregular array, `p_ii1`, that keeps track of which path element in `paths` correspond to which body. These arrays are given as inputs in line 2 of Listing 10.

Now computing the elements of $\mathbf{H}$ is easy. First the $\mathbf{H}$ matrix is initialized in line 5. Next the forces $\mathbf{f}_i$ are computed in body i's coordinates. The ith diagonal of $\mathbf{H}$ is also in i's coordinates. Theses can therefore be computed and then scattered about $\mathbf{H}$ without transforming them to another bodies coordinate system (line 8 and 9).

Now a map computes the lower triangular matrix of the $\mathbf{H}$ matrix (the first case in Equation 77). Here the forces computed in line 7 are transformed to body j's coordinates where $j \in \kappa(i)$ by first transforming them to root coordinates and then to body j's coordinates. Again the Plücker transformations from the "transformation tree" that was computed when computing $\mathbf{c}$ is used. Then the coefficients of $\mathbf{H}$ are computed in line 8. Theses are then scattered into $\mathbf{H}$ using the `paths` and `p_ii1` arrays (line 16). Since the individual elements of $\mathbf{H}$ are scalars the transpose of these are just the same scalar (case 2 of Equation 77). These can therefore also just be scattered into the upper triangular matrix of $\mathbf{H}$ (line 17).

With that the $\mathbf{H}$ matrix has been computed.

6.3 Asymptotics of the Composite-Rigid-Body Algorithm

The asymptotics of step 1 was explained in subsection 5.3 since it is just a call to RNEA. Here the work was $O(N)$ and the span was $O(\log N)$.

Computing the composite rigid-body inertias is another leaffix computation in the same vain as step 3 of RNEA. The only difference is that it is (6×6) matrices that are added. These matrix additions take constant time. The work of computing the composite rigid-body inertias is therefore $O(N)$ and the span is $O(\log N)$.

Computing $\mathbf{H}$'s work complexity is dominated by the creation of $\mathbf{H}$ (line 5 of Listing 10) which has size $N \times N$. Computing the entries of $\mathbf{H}$ are dependent on the connectivity of the kinematic tree as is show in Figure 9. The connectivity of the tree is also what determines the size of the κ array which is mapped over in line 11 to 14 in Listing 10 where the size of κ (or `paths` as it is called in the code) is $O(Nd)$ where d is the depth of the tree. In some cases $d = N$ (as is shown in Figure 9). However to encapsulate all types of trees it is more descriptive to have d as its own variable. Since the entries are computed with a map and some matrix matrix multiplications the work is $O(Nd)$ and the span is $O(1)$. The entries are then scattered into $\mathbf{H}$ which has work of $O(Nd)$ and span of $O(1)$.

Listing 10: Computing the joint-space inertia matrix, H, from the composite rigid body inertias.

```
def joint-space_inertia_matrix [n] [nd] (Ics : [n][6][6]f64) (S : [n][6]f64)
           (paths : [nd]i64) (p_ii1 : [nd]i64)
           (tr_F : [n][6][6]f64) (fr_F : [n][6][6]f64)
           : [n][n]f64 =
  let H = replicate n <| replicate n 0f64

  let fhs = map2 (*) Ics S
  let Hii = map2 (*) S fhs
  let H = scatter_2d H (zip (iota n) (iota n)) Hii

  let Hij = map2 (\i j ->
         let f = fr_F[j] * (tr_F[i] * fhs[i])
         in S[j] * f
     ) p_ii1 paths

  let H = scatter_2d H (zip p_ii1 paths) hij
  in scatter_2d H (zip paths p_ii1) hij
```

All in all the CRBA has a work complexity of $O(N^2 + Nd)$ because of the size of $\mathbf{H}$ and a span of $O(\log N)$ because of the scans used by the rootfix and leaffix operations.

7 Experiments

The performance of the data parallel implementations of the Recursive Newton Euler Algorithm and the Composite-Rigid-Body Algorithm were run on the Hendrix cluster [5] on an NVIDIA A100 40GB PCIe GPU with 40 gigabytes of memory. The CPU was an AMD EPYC 7413 24-Core Processor with 80 gigabytes of RAM. All benchmarks were run with Futhark version 0.26.2 and CUDA version 12.2.

Two sections are also dedicated to optimizations related to RNEA and CRBA. One explores the speedup achieved from using the optimized data structures. The other explores the performance of different scan implementations. Both of these optimizations have been previously presented in subsubsection 5.2.4. The results from these informed how the implementations of RNEA and CRBA could be modified such that they achieve better performance.

The data parallel RNEA and CRBA implementations are both measured against the runtime of CPU implementations that try to mimic the behavior of Featherstone's implementations of RNEA and CRBA in [11]. These CPU implementations were implemented in Futhark and run with the c backend. A short presentation of the CPU implementations can be found in Appendix A.

All benchmarks are run with the `futhark bench` tool and all tests were run with the `futhark test` tool.

Insights into the inputs to the algorithms are provided in the testing section.

7.1 Testing

The Recursive Newton Euler Algorithm and the Composite-Rigid-Body Algorithm were tested against Roy Featherstone's library of Matlab implementations which can be found in [11]. In Featherstone's library he uses a function which creates a rigid body *model*. This function has been replicated in Futhark. The *model* is provided a number N which dictates its number of bodies and a number bf which is the branching factor of the model and therefore dictates the connectivity. This determines how many children each body will have on average. A good way to think about it is that the branching factor determines the depth of the kinematic tree of the *model*:

$$depth\_of\_model = \begin{cases} 1 + \log_{bf} N & \text{if } bf > 1 \\ N & \text{if } bf = 1 \\ undefined & \text{if } bf < 1 \end{cases} \tag{79}$$

A branching factor of 1 is therefore a serial robot (open chain) that does not branch. A branching factor of 2 is a *model* where each body has 2 children. If the branching factor is a fraction e.g. 1.5 it means that each body has an average of 1.5 children. Each body in the *model* is a thin-walled cylinder with a radius of 0.05 meters, a mass of 1 kg and a length of 1 meter. All the joints in the model are revolute joints. This *model* is also the one that the implementations of RNEA and CRBA is benchmarked with.

For testing the *model* was run against Featherstone's RNEA with some random positions, velocities and accelerations. The input data that Featherstone's RNEA was run with and the torques, $\boldsymbol{\tau}$, that it returned with the given input data was then converted by some Matlab scripts to something Futhark can recognize. The data parallel implementation was then run with the exact same input and tested against the result with Futhark's C, multicore and CUDA backends. That is the $\boldsymbol{\tau}$ produced by the data parallel implementation was compared with the $\boldsymbol{\tau}$ that Featherstone's implementation produced. The accuracy of the floating point numbers that were compared was arbitrarily

chosen to be double-precision floating points at 4 decimals. Floats with more decimals have not been tested with and might also pass the tests. The input *models* that were tested with had sizes that ranged from $N = 4$ to $N = 100000$ and had varying branching factors. All tests passed.

The same approach was used when testing the CRBA implementation from subsection 6.2 except that smaller inputs (smaller Ns) were used because of the complexity of CRBA. Featherstone even has a function, $HandC$, which only computes the joint-space inertia matrix, $\mathbf{H}$, and the joint-space bias force vector, $\mathbf{c}$. All tests passed.

7.2 Optimized Data Structures and Operations

In Figure 10 the speedup of using the optimizations to the data structures of the Plücker Coordinate Transformations, spatial vectors and spatial inertias on the different implementations of RNEA and CRBA is presented. The implementation include both the CPU and the data parallel vtree implementations. They are compared to the same implementation using the spatial algebra presented in the theory chapter (section 4).

RNEA benefits the most from the use of these optimizations as can be seen in the figure. The vtree implementation has a speedup of around 3x for input models with $N \leq 100\,000$ and a around a 5 times speedup for $N \geq 1\,000\,000$. The speedup is calculated by dividing the runtime of the implementation not using the optimizations presented in subsubsection 5.2.4 with the runtime of the ones using the optimizations. The CPU implementation's performance is also improved and it has a speedup of between 1.33x and 1.9x.

As for CRBA it also receives a speedup but it is most substantial at lower input sizes of $N \leq 1000$. This is because of the lower overhead when using the optimizations since CRBA has almost identical performance for inputs with $N \leq 1000$ as is shown latter on (Figure 16). For larger Ns the speedup is around 1.5x. The CPU implementation does not benefit from the optimizations. When comparing the performance of CRBA it will therefore be compared to the "unoptimized" version of the CPU CRBA implementation since it is faster. Otherwise the rest of the benchmarks are run on the implementations that make use of the optimizations.

The large speedup seen in the vtree implementations are likely because it reduces the work of computing the transformation tree which consists of a bunch of matrix multiplications between Plücker Coordinate transformation matrices. The floating point operations are reduced to 33 multiplications and 24 additions from 216 multiplications and 180 additions when using the optimized operation. The scans of the rootfix operation therefore need to do a lot less work.

7.3 Scan Variations on rootfix and leaffix

The different scan variations that were mentioned in subsubsection 5.2.4 were benchmarked. The benchmark scenario involved doing a rootfix operation equivalent to the one computing the velocities in subsubsection 5.2.1. The inputs to the benchmark was the `vJ` and `Xup` computations from Listing 4 done on a *model* of N bodies, a branching factor of 2 and the vtree associated with this model. (As is explored later on: The branching factor does not have a large impact on the performance of RNEA. It is therefore static in this benchmark.) All scans use the optimized data structures. The results can be seen in Figure 11.

In the figure a blocked scan with a block size of 64 is used since it was the best performing block size of the ones that were benchmarked. These included block sizes of: 512, 256, 128, 64, 32 and 16.

The simulated scan in the figure refers to the process of computing the transformation tree with a rootfix that uses Plücker Coordinate multiplication using a blocked scan with a block size of 64. Then transforming `vJ` to root coordinates and performing a rootfix vector vector addition on these

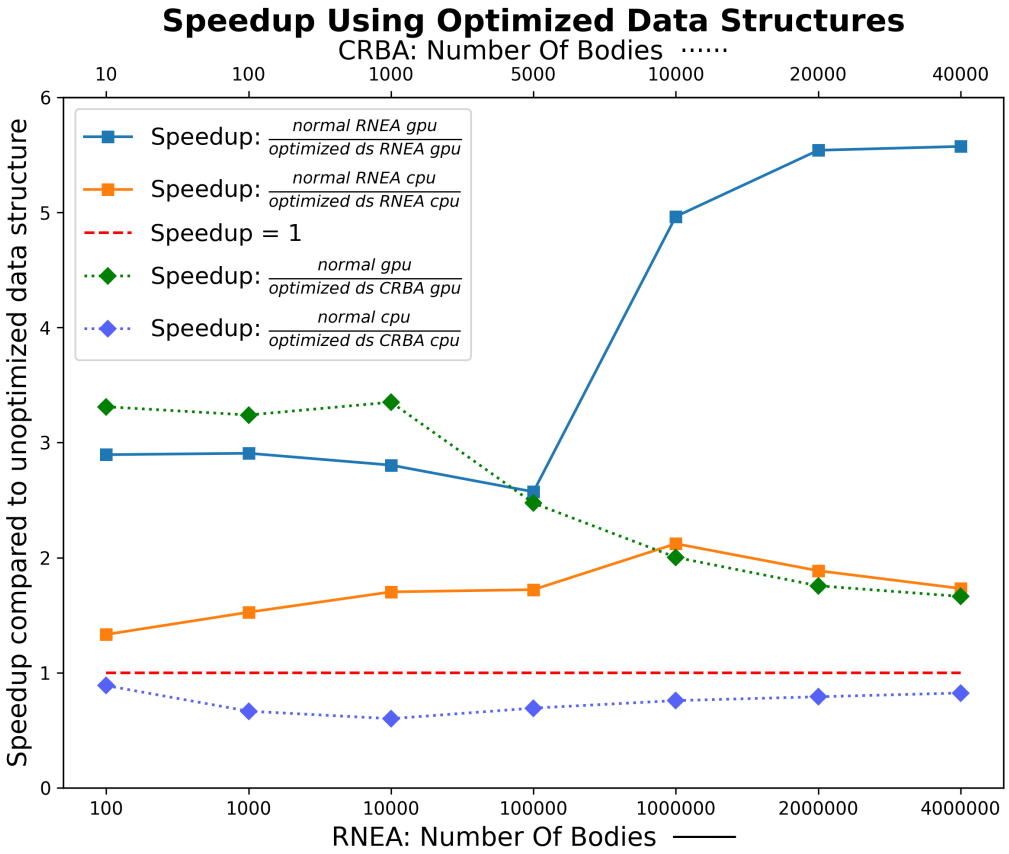


Figure 10: A comparison of the improvement that the optimized data structures and operations have on runtime for both the CPU and vtree (GPU) implementations. The y-axis shows the speedup in comparison with the algorithm using 6d vectors and 6×6 matrices and the algorithm using the optimized data structures. There are two x-axes which both display the number of bodies in the input *model*. The one on the bottom relates to RNEA. The one on the top relates to CRBA. The dotted lines shows the speedup of CRBA. The regular lines show the speedup of RNEA. All the benchmarks in this figure were run on *models* with a branching factor of 1.

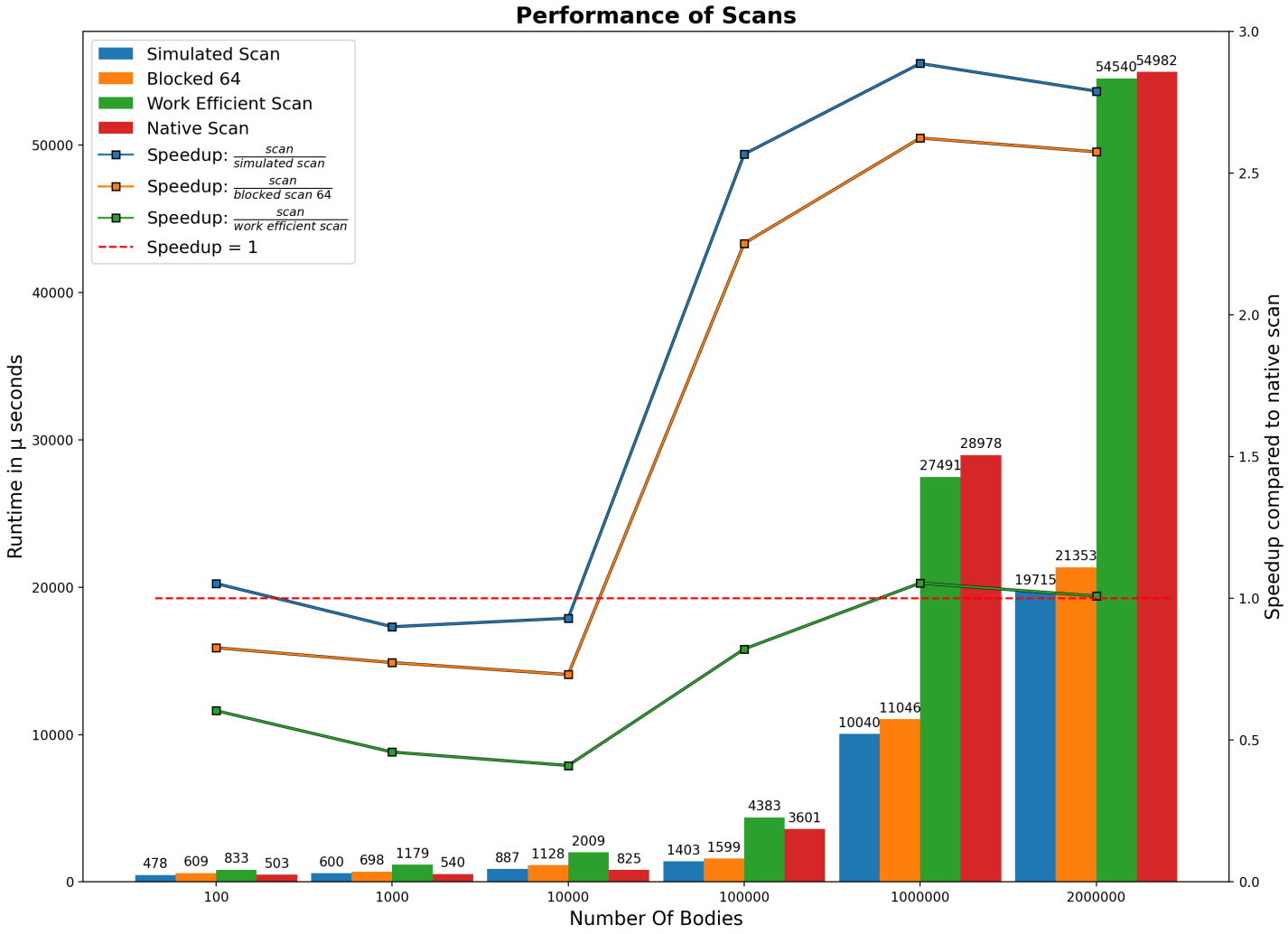


Figure 11: A comparison of the isolated runtime of the rootfix operation with different scan implementation that computes the velocities in RNEA (line 9-18 of Listing 4). Here the x-axis is the number of bodies in the *model* that is given as input. The y-axis on the left is the runtime in microseconds which relates to the bars in the figure. The values above the bars relate to the y-axis on the left. The y-axis on the right is the speedup when compared to the native scan which relates to the lines in the figure. Here four different scan implementations are compared: A simulated scan which computes the velocities using first a rootfix that computes the transformation tree, then transforms joint velocities to root coordinates and computes the velocities with a rootfix with vector vector addition. The blocked scan using a block size of 64, Blelloch's work efficient scan and Futhark's native scan implementation. These are all presented in subsubsection 5.2.4.

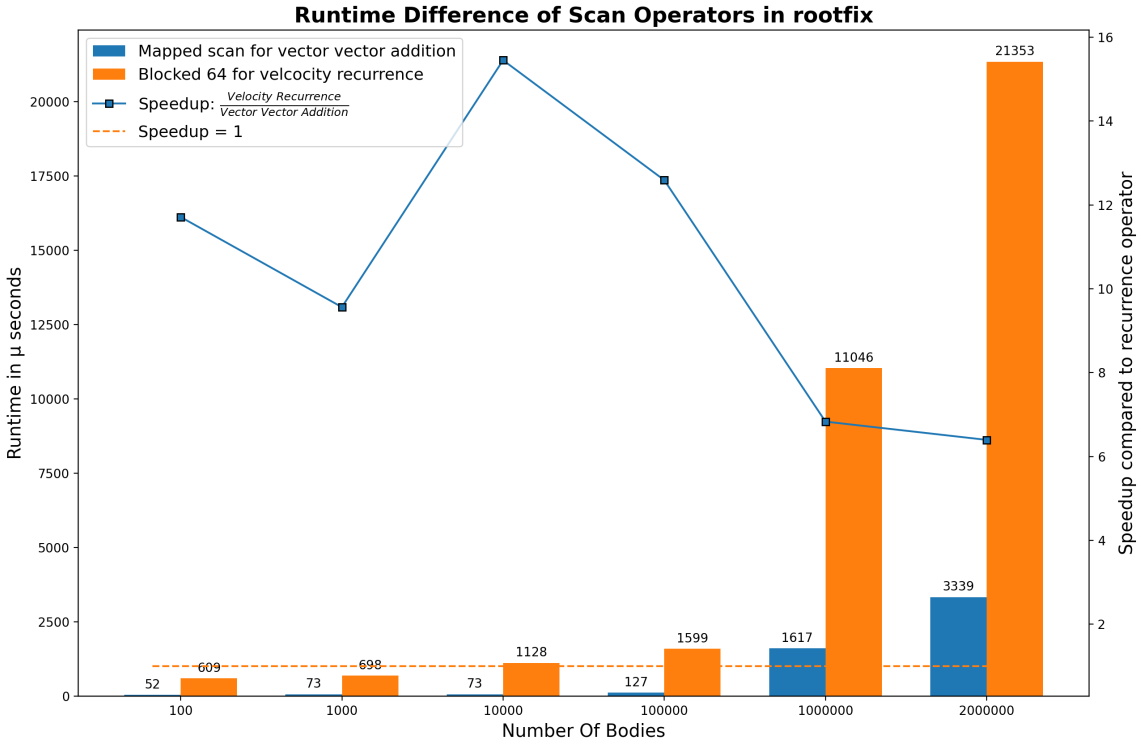


Figure 12: Comparing the runtime difference of a doing a rootfix where its operator is the recurrence defined in Equation 37 and a rootfix where its operator is vector vector addition. The axes are the same as Figure 11. The scan implementations used are the blocked scan with a block size of 64 for the recurrence operator and the mapped scan for the vector vector addition.

to compute the velocities. And finally the velocities are transformed back into body coordinates. It thus produces the same result as the other scans in this benchmark.

As you can see the blocked scan is about 2.5 times faster than the native scan for large inputs. The speedup was calculated by dividing the runtime of the native scan implementation with the blocked scan: $speedup = \frac{scan}{blocked\_scan\_64}$. Blelloch's work efficient scan was only a little faster on larger inputs when compated to the native scan. The simulated scan turned out to be the best performing solution except for inputs with $N = 1000$ and $N = 10\,000$. For larger inputs the speedup was close to 3x. The simulated scan was therefore chosen to be used when computing the recurrence shown in Equation 37 and the accompanying transformation tree.

Another thing to notice is that there is some overhead associated with the rootfix operation. E.g. an input *model* with $N = 100$ have almost the same performance as an input model with $N = 10\,000$ (see the numbers above the bars). This indicates that the data parallel RNEA and CRBA will not have good performance for low values of N.

It is also explored how the type of operator given to a rootfix or leaffix affects the runtime. What is relevant for RNEA is the recurrence operator, $\bullet$, associated with Equation 37 and Equation 34 and the vector vector addition used to compute the joint forces (Equation 55 and Listing 5 line 4 to 11). The comparison is shown in Figure 12. Here the rootfix using a blocked scan with block size 64 from Figure 11 and a rootfix which does vector vector addition using a mapped scan with the $+$

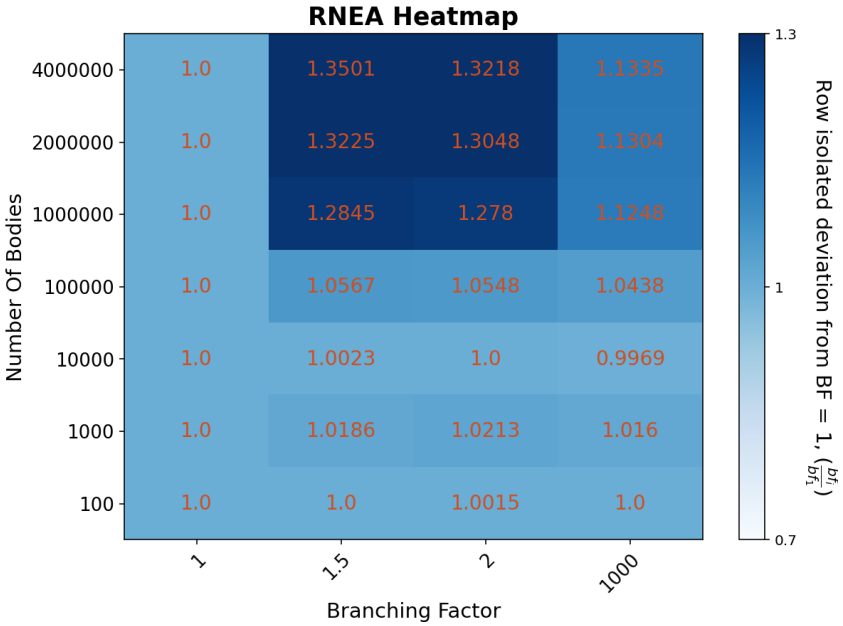


Figure 13: A heatmap of the performance of the data parallel RNEA implementation on *models* with the same number of bodies but different branching factors. Here each value in a row in the heatmap is the deviation from the performance of RNEA on the *model* with a branching factor of 1 (the first value in a row). This relates to the right y-axis. The left y-axis is the number of bodies in the input *model*. The x-axis is the branching factor of the input *model*.

operator over the 6 dimensions of the spatial vector are compared. This vector vector addition was chosen since it was the fastest according to a benchmark similar to the one presented in Figure 11 where it was done with vector vector addition instead.

Here you can see that a rootfix with vector vector addition is always faster than doing it with the • operator. It fluctuates from being around 6 times faster to around 16 times faster. This comparison inspired the change to the optimized version of RNEA (discussed in subsubsection 5.2.4): That is, instead of computing the accelerations with • they were transformed to root coordinates such that Equation 34 could be computed with a rootfix in a similar way to its coordinate free representation (vector vector addition) and then transformed back into body coordinates. The rootfix using vector vector addition with the mapped scan was therefore chosen to compute the accelerations and joint forces in RNEA.

7.4 The Performance of RNEA

First the impact of the connectivity of the input *model's* effect on RNEA's runtime is explored. Since the most interesting thing about the data parallel implementation is the use of vtrees it is important to know if the connectivity of the tree has an impact on performance. One of the benefits of the vtree data structure and the operations performed on it is that the connectivity of its tree should not impact the performance of the operations performed on it.

In Figure 13 you can see a row isolated heatmap that compares different branching factors of input *models* with the same number of bodies. Here you can see that there is little difference in

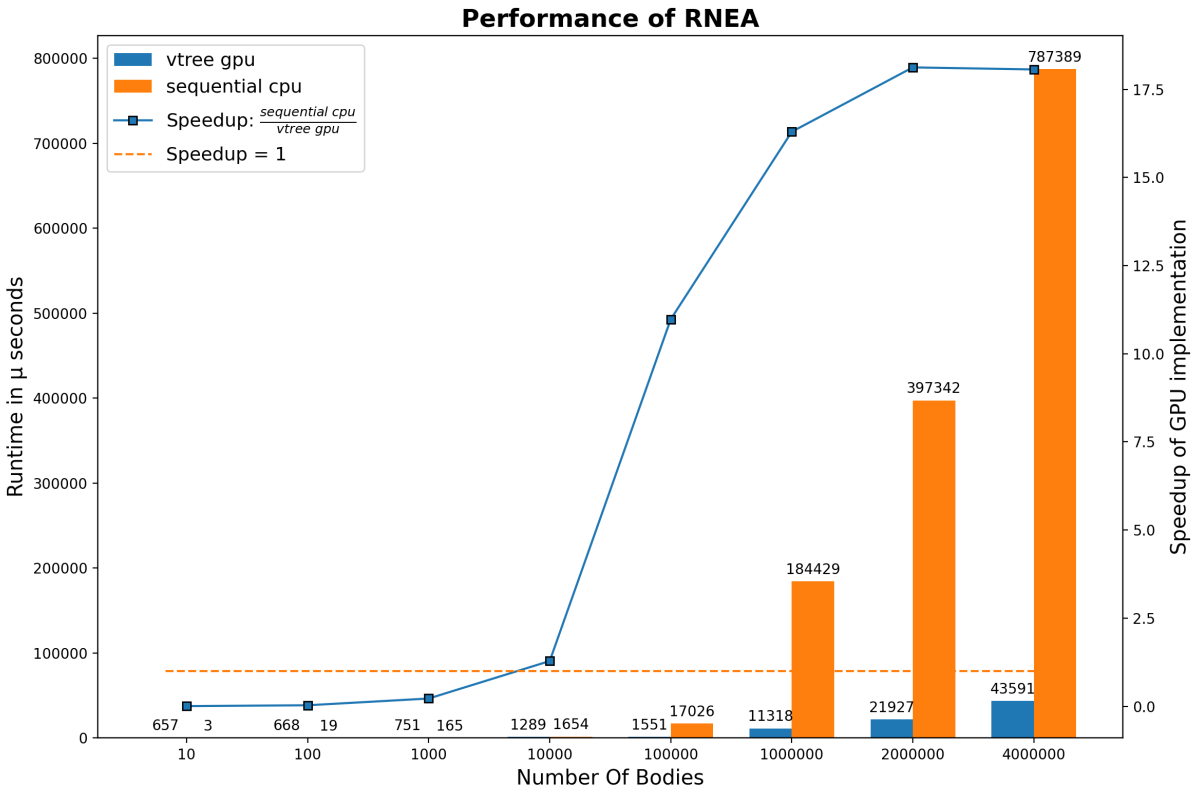


Figure 14: The performance of the data parallel RNEA and the CPU implementation of RNEA. The axes are similar to the ones in Figure 11. Both implementations are given the same input *model* with N bodies and a branching factor of 1. The branching factor is kept constant since Figure 13 showed that the connectivity does not have significant impact on the performance.

runtime between input *models* with differing branching factors in most cases. The largest deviation is when the number of bodies of the model, N, is above 1 million. Here there is a deviation where the input *models* with a branching factor of 1 is between 28 and 35 percent faster when compared to input *models* with a branching factor of 1.5 and 2. This deviation is consistent for the larger inputs. As an aside: This seems to be a property of the RNEA implementation using the optimized data structures and operations on spatial vectors. The regular implementation that uses 6d vectors and 6×6 matrices only has a maximum deviation of 6.2 percent.

To measure the performance of the data parallel implementation of RNEA it is compared to a CPU implementation inspired by [11] and implemented in Futhark. The CPU implementation computes the recurrences with the help of a parent array and was run using Futhark's c backend. The comparison can be seen in Figure 14.

For lower input sizes of $N \leq 1000$ the overhead of the data parallel implementation is too large and the CPU implementation is therefore faster. However, with $N \geq 10\,000$ the data parallel RNEA shows a speedup. For input sizes of 1 million to 4 million the speedup is around 16x to 18x.

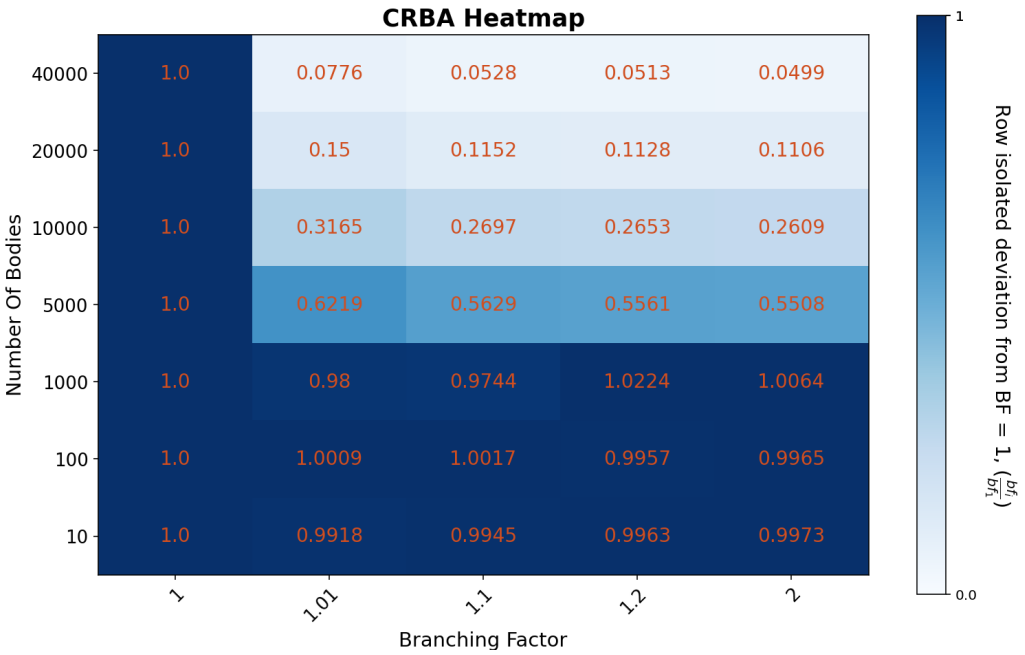


Figure 15: A heatmap with the same axes as Figure 13. CRBA is a lot more sensitive to the connectivity of the input *model* as can be seen by how much the performance deviates.

7.5 The Performance of Computing H with CRBA

As with RNEA a row isolated heatmap has been made for CRBA in Figure 15. Because of how **H**'s sparsity is linked to the connectivity of the input *model* the performance varies a lot depending on the input *model*'s branching factor. Even with a low branching factor of 1.01 CRBA is a lot faster compared to when the branching factor is 1. The largest difference is seen on an input with $N = 40\,000$. Here the input *model* with a branching factor of 2 is over 20 times faster compared to the input model with an unbranching tree. The deviation increases as N increases. If one looks at the rows with $N \geq 5000$ and a branching factor different from 1 the deviation doubles as the input doubles in each column.

In Figure 16 you can see the performance of the data parallel CRBA compared to a CPU CRBA implementation inspired by the *HandC* function in [11]. The CPU implementation computes the **H** matrix using the parent array to traverse the path from each body to the root to compute the entries of **H**. The CPU implementation does not use the optimized data structures as discussed in subsection 7.2 since it made it slower. The largest amount of the implementations' work is done when computing the entries of **H**. The data parallel CRBA also does not use any costly scan operations when computing **H**. This might be one reason for why the speedup is large when compared to the CPU implementation.

In the figure the performance of four different branching factors is shown. The one in the upper left corner is run with a *model* that has a branching factor of 1. This is the benchmark where both the CPU and data parallel implementations need to do the most work. It is also here where the largest speedup is seen when comparing the two implementations. Already at 1000 bodies there is a substantial speedup of 16.5x. For the input *model* with $N = 40\,000$ the data parallel vtree

implementation is around 163 times faster. The *model* with $N = 40\,000$ and a branching factor of 1.01 also produced a speedup of around 160x. The two other graphs do not have quite as large of a speedup even though their speedups are still substantial for larger input sizes. The curves of their speedups on increasing Ns are also steeper which indicates that their speedup might increase with inputs of $N > 40\,000$.

None of the benchmarks indicate when the speedup will level off since the speedup is always increasing as N and the complexity of computing $\mathbf{H}$ is increasing in this benchmark. Another thing one will notice is that there is still a lot of overhead on smaller input sizes below $N = 1000$ for the data parallel implementation. The data parallel CRBA implementation uses the same expensive scans that the data parallel RNEA implementation uses when computing $\mathbf{c}$ and the composite rigid-body inertias. This might be the cause for the large overhead.

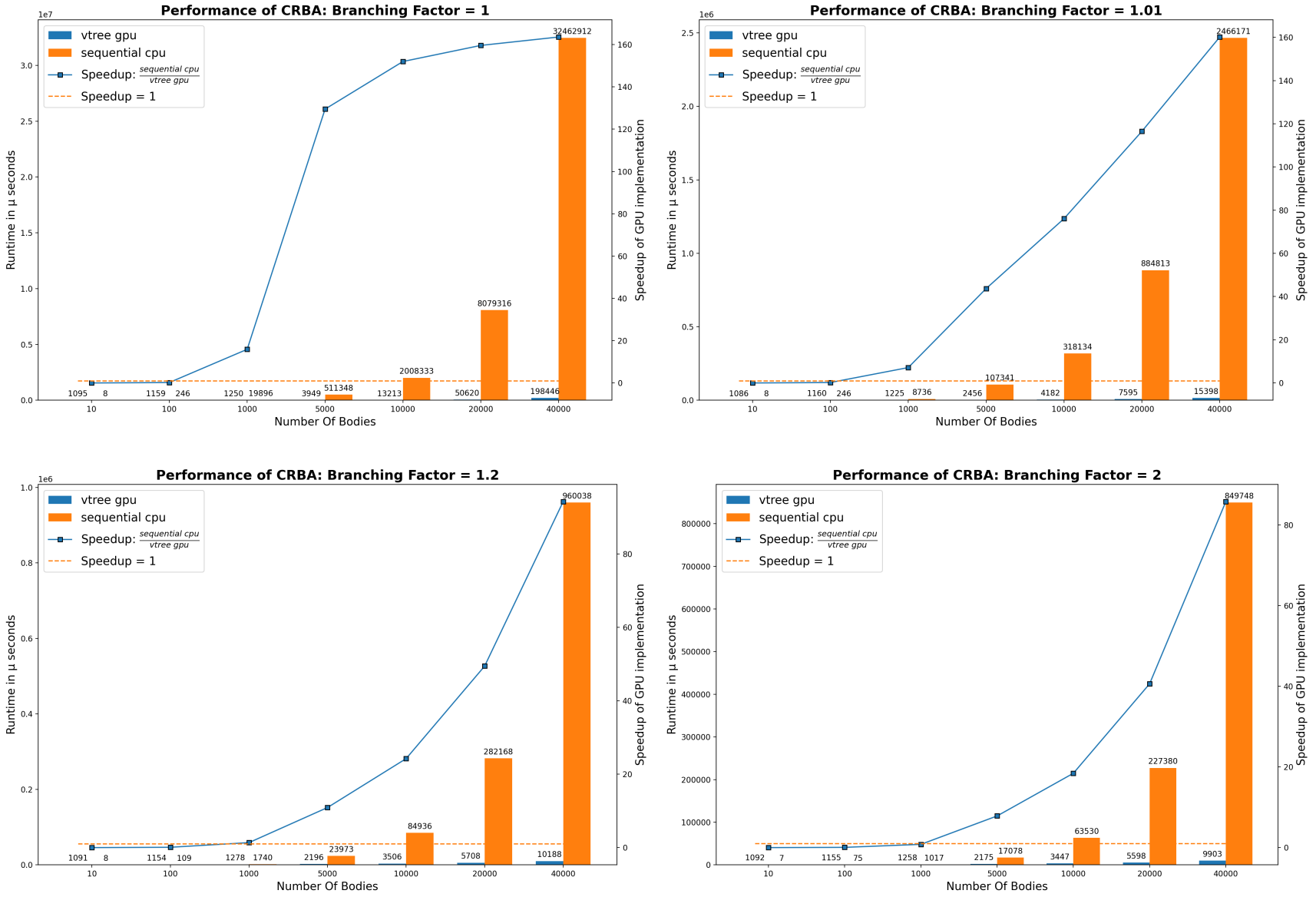


Figure 16: The performance of the data parallel implementation of CRBA when compared to a CPU implementation of CRBA. Since the performance of CRBA varies depending on the connectivity of the input *model* 4 different charts with different branching factor values in the input *model* are presented. The axes of each of the charts are the same as in Figure 11.

8 Conclusion

In this thesis the Recursive Newton Euler Algorithm and the computation of the **c** vector and the **H** matrix for the Composite-Rigid-Body Algorithm were implemented using vtrees to compute the recurrences found in these algorithms. Each implementation was tested and their performance using various optimizations was measured against CPU implementations. The optimizations included using different implementations of scan for the vtree's rootfix and leaffix operations and using optimized data structures for the spatial vector algebra. The maximum speedup measured of the Recursive Newton Euler Algorithm is 18x as shown in Figure 14. The maximum speedup measured for the Composite-Rigid-Body algorithm was 163x as shown in Figure 16.

Another algorithm which uses recurrences to compute the forward dynamics of rigid body systems is the Articulated-Body Algorithm which has $O(N)$ time complexity. These recurrences resembles the ones used by the algorithms that were implemented in this thesis. Future work could explore the potential this algorithm has to be computed using vtrees.

9 Related Work

This chapter presents previous work on parallel computation of open chains in subsection 9.1 and a divide and conquer approach to computing rigid body dynamics in subsection 9.2.

9.1 Scans Computing Dynamics for Open Chain Robots

In open chain robots the tree structure of the robot (also called the kinematic chain) is structured such that the bodies are connected in a sequence. That is body i's parent is body $i-1$ and its one child is body $i+1$. This unbranching tree structure can also be represented with a simple array.

In [16] they use this fact to implement the Recursive Newton Euler Algorithm (RNEA) and some forward dynamics algorithms. As previously mentioned RNEA computes the velocities of the bodies by propagating them from body i to its children. Since each body only has one child the velocity of body i will propagate to all the remaining bodies in the sequence of the open chain. In this way the forward propagation of the velocities works similarly to a scan and its accumulator. The accelerations can be computed in a similar fashion while the forces of the joints are propagated backwards also using a scan. That is the forces applied to body i are propagated backwards to its parent (body $i-1$) to compute the forces at the joints of the bodies. In this way three scans can compute the forces applied to the joints of the bodies in the open chain which can then be used to compute their torques $\boldsymbol{\tau}$.

The rootfix and leaffix operations on vtrees can be thought of as a way of unfolding the branching structure of a tree into an array on which a scan operation can be used. In this way any tree structure can be computed and not just open chains (unbranching trees). The RNEA implementation in subsection 5.2's rootfix and leaffix operators are therefore comparable to the scans used in [16].

9.2 Divide and Conquer

The Articulated-Body Algorithm is a forwards dynamics algorithm with time complexity $O(N)$. It takes its basis in the equation of motion for articulated bodies:

$$\mathbf{f}_i = \mathbf{I}_i^A \mathbf{a}_i + \mathbf{p}_i^A \tag{80}$$

This describes the relation between the force, $\mathbf{f}_i$, applied to body i and the resulting acceleration, $\mathbf{a}_i$, given the inertia, $\mathbf{I}_i^A$, of the articulated body of body i and the bias force, $\mathbf{p}_i^A$, of the articulated body of body i. The articulated body of body i can be thought of as a body that takes into account the dynamics of all the bodies in the subtree rooted at body i. The main problem that the Articulated-Body Algorithm solves is to find the articulated body inertias and bias forces using recursion such that the accelerations can be found using Equation 80.

This algorithm has been parallelized using a divide and conquer scheme in [6] and [7]. Here a separate assembly tree is created from the kinematic tree that describes the rigid body system. This assembly tree describes how the different parts of the tree should be computed. The assembly tree achieves a depth of $O(\log n)$, however it does so by splitting some of the bodies who have more than 1 child into k bodies where k is the number of its children. This does not increase the number of bodies in the system to more than $O(N)$ where N is the number of bodies in the kinematic tree. The assembly tree is then used to compute the articulated bodies of the nodes by assembling the nodes as described by the tree. The algorithm's work complexity is $O(N)$ and its span is $O(\log n)$.

The Articulated-Body Algorithm is of interest in relation to vtrees since the algorithm can be described with recursions similar to the ones that describe (RNEA) and the Composite-Rigid-Body

Algorithm (CRBA). It also has a time complexity of $O(N)$ in comparison the CRBA which solves the same problem and has a time complexity of $O(N^3)$.[2] However, the recursion that computes the articulated bodies is more complex compared to the ones encountered in RNEA and CRBA which might prove difficult to implement using vtrees.

[2]The amount of work CRBA has to do is reliant on the depth of its kinematic tree. Featherstone argues in chapter 6.5 of [8] that a more accurate description of its time complexity is $O(Nd^2)$ where d is the depth of the kinematic tree.

References

[1] V. S. Aslanov. *Rigid Body Dynamics for Space Applications*. Butterworth-Heinemann, 2017.

[2] G. E. Blelloch. Prefix sums and their applications. 1990.

[3] G. E. Blelloch. Vector models for data-parallel computing. In *Vector Models for Data-Parallel Computing*, 1990.

[4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[5] diku dk. Hendrix cluster. `https://diku-dk.github.io/wiki/slurm-cluster`. Retrieved 13-05-26.

[6] R. Featherstone. A divide-and-conquer articulated-body algorithm for parallel o(log(n)) calculation of rigid-body dynamics. part 1: Basic algorithm. *The International Journal of Robotics Research*, 18(9):867–875, 1999.

[7] R. Featherstone. A divide-and-conquer articulated-body algorithm for parallel o(log(n)) calculation of rigid-body dynamics. part 2: Trees, loops, and accuracy. *The International Journal of Robotics Research*, 18(9):876–892, 1999.

[8] R. Featherstone. *Rigid Body Dynamics Algorithms*, volume vol. 49. 01 2007.

[9] R. Featherstone. A beginner's guide to 6-d vectors (part 1). *IEEE Robotics Automation Magazine*, 17(3):83–94, 2010.

[10] R. Featherstone. A beginner's guide to 6-d vectors (part 2) [tutorial]. *IEEE Robotics Automation Magazine*, 17(4):88–99, 2010.

[11] R. Featherstone. Spatial vector and rigid-body dynamics software. `https://royfeatherstone.org/spatial/v2/`, 2015. Retrieved 13-05-26.

[12] G. Golluccio, G. Gillini, A. Marino, and G. Antonelli. Robot dynamics identification, 2019.

[13] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea. Compiling generalized histograms for gpu. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[14] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 53–67, New York, NY, USA, 2019. Association for Computing Machinery.

[15] W. H. D. Martin Elsman, Troels Henriksen. vtree. `https://github.com/diku-dk/vtree`, 2025. Commit: 2e9e5ef96252566313088e7c3f23f7aaf87a0786.

[16] J. P. Yajue Yang, Yuanqing Wu. Parallel dynamics computation using prefix sum operations. *IEEE Robotics and Automation Letters*, 2(3):1296–1303, 2016.

A CPU Implementations of RNEA & CRBA

This appendix presents the CPU implementations that the data parallel implementations were compared against.

A.1 RNEA

The sequential RNEA implementation is shown in Listing 11. Here "*" is again used as an all purpose operator which indicates multiplications of different kinds depending on the input. The implementation resembles the data parallel one but now loops are used to compute the recurrences. Here in loop iteration i the parent array is used to lookup the parent of i to know how to compute the recurrence. In this way the sequential implementation resembles the recurrence equations of section 5 which use the value of the parent.

Body i's joint force is computed with the values of its children. When the joint forces are computed body i of loop iteration i adds its body force sum to its parent. In this way the recurrence of the joint forces are computed.

One thing you might notice is that there is no need for a "transformation tree" and that you do not transform the velocities, accelerations and body forces to root coordinates and then back to body coordinates. The CPU implementation therefore has less it needs to compute. However, the asymptotic time complexity of both implementations is still the same: $O(N)$.

A.2 CRBA

The sequential implementation of computing the joint-space bias force, $\mathbf{c}$, and the joint-space inertia matrix, $\mathbf{H}$, is shown in Listing 12.

The first step is just a call to RNEA as in the data parallel implementation. Step 2 then computes the composite rigid bodies with a loop which resembles the recurrence equation.

As with RNEA there is no need to do as many coordinate transforms as in the data parallel version. However when $\mathbf{H}$ needs to be computed the data parallel implementation and the sequential implementation needs to do the same amount of computations. The sequential implementation goes about it differently to the data parallel one. Instead of the κ array it again uses the parent array to compute paths to the root with a loop. The data parallel implementation transforms all the forces to root coordinates and when they need to be multiplied with the vector subspace of the joint (`S[i]`) they are transformed to body coordinates. The sequential implementation instead transforms the forces from the current body's coordinates to its parents and inserts it into the $\mathbf{H}$ matrix.

Listing 11: Sequential implementation of RNEA using loops to compute the recurrences.

```
def rnea_seq [n] (p : [n]i64) (joint_types : [n]jointT)
           (Is : [n][6][6]f64) (Xtree : [n][6][6]f64)
           (gravity : [6]f64) (q : [n]f64) (qd : [n]f64) (qdd : [n]f64)
           : [n]f64 =
  let (XJ, S) = unzip <| map2 (jcalc) joint_types q
  let vJ = map2 (*) qd S
  let Xup = map2 (*) XJ Xtree

  let n_vectors = replicate n (replicate 6 0f64)
  let (vs, as) = loop (vs', as') = (copy n_vectors, copy n_vectors) for i < n do
    if i == 0 then
        let vs' = vs' with [i] = vJ[i]
        let as' = as' with [i] = Xup[i] * ((-1) * gravity) +
                                 qdd[i] * S[i]
        in (vs', as')
    else
        let parent = p[i]
        let vs' = vs' with [i] = Xup[i] * vs'[parent] + vJ[i]
        let as' = as' with [i] = Xup[i] * as'[parent] + qdd[i] * S[i] +
                                (crm vs'[i]) * vJ[i]
        in (vs', as')

  let fBs = tabulate n
         (\i -> Is[i] * (as[i] * (((crf vs[i]) * Is[i]) * vs[i])))

  let (tau, _) = loop (tau', fs') = (replicate n 0f64, fBs) for i < n do
    let idx = n - (i+1)
    let p' = p[idx]
    let tau' = tau' with [idx] = S[idx] * fs'[idx]
    in
      if idx > 0 then
        let fs'' = fs' with [p'] = fs'[p'] + (transpose Xup[idx]) * fs'[idx]
        in (tau', fs'')
      else (tau', fs')
  in tau
```

Listing 12: Sequential implementation of CRBA using loops to compute the recurrences.

```
def crba_seq [n] (p : [n]i64) (joint_types : [n]jointT)
            (Is : [n][6][6]f64) (Xtree : [n][6][6]f64)
            (gravity : [6]f64) (q : [n]f64) (qd : [n]f64)
            : ([n]f64, [n][n]f64) =
  -- Step 1: Compute the joint-space bias force
  let c = rnea_seq p joint_types Is Xtree gravity q qd 0

  -- Step 2: Compute the joint-space inertia matrix
  let Ics = loop IC = (copy Is) for i < n do
    let idx = n - (i+1)
    let parent = p[idx]
    in
        if idx > 0 then
          let tmp = IC[parent] + (transpose Xup[idx]) * (IC[idx] * Xup[idx])
          in IC with [parent] = copy tmp
        else IC

  let H'' = loop H = (replicate n <| replicate n 0f64) for i < n do
    let fh = Ics[i] * S[i]
    let H = H with [i,i] = S[i] * fh
    let (H'',_,_) = loop (h, j, f) = (H, i, fh) while p[j] >= 0 do
        let fh' = (transpose Xup[j]) * f
        let j = p[j]
        let h' = h with [i,j] = S[j] * fh'
        let h'' = h' with [j,i] = (copy h'[i,j])
        in (h'', j, fh')
    in H''
  in (c, H'')
```