IEEE Trans. on Robotics and Automation, Vol. 9, No. 6, December, 1993, pp. 858-863.

Vectorization of Robot Inverse Dynamics on a Pipelined Vector Processor[1]

Harry H. Cheng
Integration Engineering Laboratory
Department of Mechanical & Aeronautical Engineering
and Graduate Group in Computer Science
University of California
Davis, CA 95616
Email: hhcheng@ucdavis.edu
World Wide Web: http://iel.ucdavis.edu

Krishna C. Gupta
Department of Mechanical Engineering
University of Illinois at Chicago
Chicago, IL 60680
Email: *k.c.gupta@uic.edu*

Abstract

This paper presents a computational scheme on a pipelined vector processor for the robot inverse dynamics. Vectorization of the inverse dynamics computation is achieved while the dynamics formulation still remains within the general linear recursive framework. Parametric studies are conducted on a commercially available pipelined vector processor. The speedup 1.82 is achieved through vectorization for the inverse dynamics computation of a general six-degrees-of-freedom robot manipulator.

1 Introduction

The determination of the driving actuator forces or torques of a robot manipulator to realize the specified end-effector motion is called the robot inverse dynamics, which is required in many robot control systems [13]. The computation frequency of the robot inverse dynamics higher than 60 Hz, about ten times the mechanical resonant frequency of most industrial manipulators, is needed in order to achieve fast convergence of control algorithms. It has been shown that, for general dynamics formulations which are manipulator configuration independent, the recursive Newton-Euler formulation is one of the most efficient ones [14]. To control a redundant robot, which has more than six degrees of freedom, more computational power is demanded. It is unlikely that the computational complexity can be

[1] An earlier version of this paper was presented at *The IEEE International Conference on Robotics and Automation*, Sacramento, CA, April 9-11, 1991. This work was supported in part by the National Science Foundation as a Research Initiation Award under Grant DMII-9309207, and it utilized the Cray X-MP/48 at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

further significantly reduced on a scalar uniprocessor because the order of the computational complexity for the recursive formulation already varies linearly with the number of joints.

However, parallel processing can provide potential gains in computational speed. Parallel processing allows multiple parts of the program to be executed simultaneously, thereby reducing the total execution time. Luh and Lin [15] described a procedure for scheduling the parallel computation of the inverse dynamics. One processor is assigned to each manipulator link. By only considering the floating-point operations of the algorithm and ignoring the overhead associated with data communication and synchronization, they reported a computational speedup of 2.64 for Stanford arm. The parallel execution procedure obtained through a modified branch-and-bound search algorithm is manipulator configuration dependent. Along this line, Kasahara and Narita [8] and Chen et al. [1] also considered the scheduling algorithms for the parallel execution of the inverse dynamics. Lathrop [11] proposed two parallel algorithms for the computation of the inverse dynamics. The execution time required for one of the algorithms is linear in the number of joints and for the other it is logarithmic. By assigning a group of processors to each joint in the recursive formulation, the massive parallel processing of the floating-point operations in the algorithm was performed. The systolic pipeline architecture was also exploited. The number of processors required for the implementation of the logarithmic algorithm in a systolic pipeline architecture is 637 for a six degrees of freedom robot manipulator. Data communication and synchronization for these algorithms would be very costly because the large number of processors are involved, especially for systolic algorithms which require implicit synchronization between processes producing data and processes consuming data [10]. The pipeline overhead of a systolic algorithm may also result in the failure of discrete robot control systems. The parallel algorithm for the robot inverse dynamics developed by Lee and Chang [12] is based upon the recursive doubling algorithm [9] and uses the Newton-Euler equations expressed in the base coordinate system. The systolic pipeline architecture has also been investigated. It requires 530 modular processors to implement the parallel algorithm for a six-degrees-of-freedom robot manipulator.

However, parallel processing can be efficient only when the cost for scheduling and synchronizing parallel activities is relatively small. In contrast to scalar and vector processings which are relatively mature, parallel processing is still in an embryonic state. Computations that exhibit coarse-grain parallelism can be efficiently parallelized while fine-grain parallelism may result in slowing down, rather than accelerating, the execution of the program. Unlike many other computationally intensive problems for which the run times on a currently available fastest supercomputer can be measured in hours or even days; computations involved in the robot inverse dynamics are relatively small. For example, there are only 852 floating-point multiplications and 738 additions for the inverse dynamics computation of a general six-degrees-of-freedom robot manipulator. If these relatively small amount of floating-point operations are broken into many tiny pieces in fine-grain massive parallel fashion, the program will likely slow down rather than speed up because of the excessive overhead for data communication and synchronization among processors.

Besides parallelization, vectorization can also enhance the performance of the algorithm. Unlike scalar processing which executes a single pair of data per machine instruction, vector processing performs the same operation on many pairs of data in one machine instruction. Pipelined vector processors have been commercially available since the early of 1970's. It is possible that a well-designed vector processor with vector operations such as vector chaining will ensure that the computational speed of an algorithm in vector mode can be faster than that in scalar mode even for the vector length of two [2]. However, despite the great success of vector processing in the computer market for the past decade, applications of vector pipelined processors in robotics have not been fully explored yet. The real-time robot dynamic simulation on a vector/ parallel supercomputer has been investigated [16]. In this paper, an numerical algorithm of the robot inverse dynamics for vector processing is developed. The vector algorithm is targeted for execution on a pipelined vector processor. For more information about vector processing, one can refer to [2], [17]. It is vector pipelining and vector chaining that distinguish vector processing from conventional scalar processing. Some general guidelines for successful vectorization on vector processors from both vector pipelining and vector chaining points of views can be found in [2].

2 Recursive Formulation of Inverse Dynamics

To vectorize and parallelize the computation of the robot inverse dynamics, as efficient an algorithm as possible must be used. Although it is generally acknowledged that the fastest serial scalar algorithm does not necessarily make the most efficient vector and parallel algorithm [17], it is important that as little time complexity as possible is added to an algorithm which is chosen to be vectorized and parallelized. It has been shown that the recursive formulation based on Newton-Euler equations, which is computationally equivalent to the recursive Lagrangian formulation is one of the most efficient algorithms for computation of the robot inverse dynamics. In this formulation, a recursive computation of the joint angular velocities and accelerations starts from the base to the end-effector. The linear accelerations of mass centers of each link are also computed at each step. These linear accelerations, and angular velocities and accelerations are then used to calculate the inertia forces and inertia moments acting on each link through Newton-Euler dynamic equations. But, the joint reaction forces and moments are calculated recursively from the end-effector to the base. To find the joint actuator driving forces or torques, the D'Alembert principle is used to obtain the dynamic equilibrium equations for each link. Formulations used for vectorization in this paper are based upon Newton-Euler equations and the zero reference position method. The recursive formulations of a general robot inverse dynamics are included in Appendix.

It should be pointed out that the choice of the zero reference position of a robot manipulator in the formulation is arbitrary as was discussed in [7]. If zero position of the robot arm described by the Denavit-Hartenberg notation is used as the zero reference position, then, the results obtained by the formulation given in Appendix will be identical to ones obtained by the D-H notation. Therefore, the vectorization scheme presented in this paper is independent of the notation used to describe the manipulator.

3 Vectorization of Inverse Dynamics

The recursive formulation of a general robot inverse dynamics in Appendix is most efficient for scalar processing. However, the recursive equations cannot be correctly vectorized on a pipelined vector processor because of data dependence. A data dependence exists when the order in which statements are executed is important to the results of the program. The

Table 1: Vector operations and vector length N for vectorization of the inverse dynamics computation.

Loop	N	Vector Operations	Eq. No.
1	3	$\mathbf{R}_{B,T}$	16
2	3	$\mathbf{R}_{B,T}\mathbf{R}_{B,Z}^T$	17
3	3	$-\mathbf{R}_{Z,T}^T\mathbf{f}_e$; $\quad -\mathbf{R}_{Z,T}^T\mathbf{n}_e$	18
4	n	$\sin q_k$; $\quad \cos q_k$; $\quad R_{ij}(q_k, \mathbf{u}_{k0}), i, j = 1, 2, 3$; $\quad \dot{q}_k\mathbf{u}_{k0}$	1, 2
5	$(n), 3$	$\boldsymbol{\omega}_k^* \Longleftarrow \boldsymbol{\omega}_{k-1}^*$	2
6	n	$\ddot{q}_k\mathbf{u}_{k0} + \boldsymbol{\omega}_{k-1}^* \times (\dot{q}_k\mathbf{u}_{k0})$	3
7	$(n), 3$	$\boldsymbol{\alpha}_k^* \Longleftarrow \boldsymbol{\alpha}_{k-1}^*$	3
8	n	$\boldsymbol{\omega}_k^* \cdot \boldsymbol{\omega}_k^*$	6
9	$n-1$	$\boldsymbol{\omega}_k^* \cdot \mathbf{d}_k^*$; $\quad (\boldsymbol{\omega}_k^* \cdot \mathbf{d}_k^*)\boldsymbol{\omega}_k^* - (\boldsymbol{\omega}_k^* \cdot \boldsymbol{\omega}_k^*)\mathbf{d}_k^* + \boldsymbol{\alpha}_k^* \times \mathbf{d}_k^*$	6
10	n	$\boldsymbol{\omega}_k^* \cdot \mathbf{c}_k^*$; $\quad (\boldsymbol{\omega}_k^* \cdot \mathbf{c}_k^*)\boldsymbol{\omega}_k^* - (\boldsymbol{\omega}_k^* \cdot \boldsymbol{\omega}_k^*)\mathbf{c}_k^* + \boldsymbol{\alpha}_k^* \times \mathbf{c}_k^*$	8
11	$(n), 3$	$\mathbf{a}_{G_k}^* \Longleftarrow \mathbf{a}_{G_{k-1}}^*$	6, 7, 8
12	n	$-m_k\mathbf{a}_{G_k}^*$	9
13	$(n), 3$	$\mathbf{f}_k^* \Longleftarrow \mathbf{f}_{k+1}^*$	11
14	n	$\mathbf{b}_k^* \times \mathbf{f}_{k+1}^* - \mathbf{c}_k^* \times \mathbf{F}_k^* + \boldsymbol{\omega}_k^* \times (\mathbf{M}_{k0}\boldsymbol{\omega}_k^*) + \mathbf{M}_{k0}\boldsymbol{\alpha}_k^*$	10, 12
15	$(n), 3$	$\mathbf{n}_k^* \Longleftarrow \mathbf{n}_{k+1}^*$	12
16	n	$\mathbf{n}_k^* \cdot \mathbf{u}_{k0}$	13

following Fortran vector loop, which is equivalent to the operations of the recursive formulations in Appendix, can be used to demonstrate the data dependence of the recursive structure.

```
          Do  100  k = 1, N
100       X(k + 1) = X(k) + R * Z(k)
```

In scalar execution mode, X(2) equal to X(1) + R*Z(1) is stored before it is fetched to calculate X(3) = X(2) + R*Z(2). If this loop was vectorized, all of the N elements of X and Z on the right-hand side would be fetched. All the elements would be added together and then stored in $X(2)$ through $X(N+1)$. This would result in the data element $X(2)$ being fetched before it was stored. The wrong value of $X(2)$ would be used for the computation. Therefore, the recursive computational scheme of the robot inverse dynamics for scalar processors cannot be directly used for vector processors.

The data dependence of the recurrence is the main obstacle which inhibits vectorization. The efficient recursive formulations in Appendix have to be restructured so that the vector facilities can be utilized to enhance the performance of the algorithm. The key to successful vectorization of an algorithm on vector processors is to have sufficiently large values of vector lengths. For this purpose, the number of bodies n can, in general, be chosen as the vector length for vectorization. The computational scheme on a vector processor is given in Table 1. For clarity, the listed vector operations are for manipulators with all revolute joints. Due to vector pipelining and chaining, the loop setup, load, and store operation can cost the same CPU time as the floating-point operations. Therefore, the simple statements are combined into a complicated statement whenever possible. There are totally 16 vector loops. The loops 1, 2, and 3, with the vector length 3, are independent of the number of joints. The results obtained in loop 8 are used in both loops 9 and 10. The loops 5, 7, 11, 13, and 15 with the vector length 3 for vectorization involve recursive computations which are inherent from the recursive structure of the original algorithm. All other loops have the vectorization length of n ($k = 1$ to n), except for loop 9 with vector length $(n-1)$. The most computationally intensive loops 4, 6, 9, 10, and 14 are successfully vectorized. The equation numbers of the formulation in Appendix, corresponding to the vector operations in the third column of Table 1, are given in the fourth column for the cross reference.

Parametric studies have been conducted to test the performance of the vector algorithm on a pipelined vector processor. The algorithm was programmed in the Fortran language and run on a Cray X-MP which has both scalar and vector facilities. It should be mentioned that machines, such as the Cray X-MP/48 and IBM 3090/600VF, are general-purpose computers which are not best suitable for applications in robotics. Moreover, using a supercomputer to control a robot manipulator is not realistic from the cost effective point of view. But, while vector and parallel processings have reached the minicomputer market, these capabilities for personal computers are emerging slowly. Advances in the design and fabrication of VLSI circuits have already made it feasible to build vector and parallel machines using off-the-shelf VLSI products. For example, the SKYbolt Application Accelerator is an add-in high performance application accelerator which incorporates the Intel i860 RISC CPU [18]. This accelerator provides application programs with both scalar and vector processing capabilities in a VME board level. The SuperCard SC-2XL/VME introduced by CSP, Inc. even has dual Intel i860 vector processors on one VME board. It should be pointed out that the Alliant FX/8 minisupercomputer has eight Intel i860 CPUs which are used for scalar, vector, and parallel processings. Therefore, it is quite possible that, due to the rapid advance of the technology, the power and features of today's supercomputers will be enjoyed by tomorrow's personal or even lap-top computers. Nevertheless, the machine architectures of supercomputers, which represent the current state-of-the-art of vector and parallel computing technologies, may be tailored to the specific characteristics of the robot inverse dynamics. The special-purpose hardware can be designed based upon the simulation results on a general-purpose computer. The selection of the CPU of Cray X-MP to test the vectorization algorithm presented in this paper is mainly due to our vector processor accessing availability and its well-known superior vector processing capabilities. The performance evaluation of this supercomputer from vector pipelining and vector chaining points of views can be found in reference [2] which also contains the method used to measure the CPU times presented in this paper. The Cray X-MP has a very small vector break-even length which is defined as the minimum vector length for the vector mode of a vector processor surpassing the equivalent scalar mode [2]. The testing results of the above developed vector algorithm on a Cray X-MP are discussed as follows. Given the joint values $\mathbf{q}$, rates $\dot{\mathbf{q}}$, and accelerations $\ddot{\mathbf{q}}$ along with the external force $\mathbf{f}_e$ and moment $\mathbf{n}_e$ acting on the end-effector, the speed for computing a set of joint actuator forces or torques are given in Table 2. The number of joints n ranges from 5 to 24. The value of H_1 is the computational frequency in hertz for a robot with its first joint being a prismatic one when the program is run in scalar mode; but, the program is developed for general robots with any number of revolute or prismatic joints. The value H_2 is the computational frequency in vector mode. The only difference in

Table 2: Computational frequency (hertz) for the inverse dynamics of fully general robots and general revolute robots in both scalar and vector modes, and the corresponding speedups with different numbers of joints.

n	R-joints and one P-joint			R-joints only		
	Scalar H_1(hz.)	Vector H_2(hz.)	Speedup $S_v = H_2/H_1$	Scalar H_3(hz.)	Vector H_4(hz.)	Speedup $S_v = H_4/H_3$
5	7201	10332	1.43	8409	14493	1.72
6	6082	9320	1.53	7090	12890	1.82
7	5263	8459	1.61	6143	11529	1.88
8	4655	7756	1.67	5413	10456	1.93
9	4146	7152	1.73	4840	9533	1.97
10	3762	6656	1.77	4367	8753	2.01
11	3442	6215	1.81	3958	8098	2.05
12	3166	5867	1.85	3641	7586	2.08
14	2740	5218	1.91	3135	6662	2.13
16	2402	4666	1.94	2751	5975	2.17
20	1934	3905	2.02	2189	4874	2.23
24	1618	3328	2.06	1848	4156	2.25

running the computer program to obtain the values of H_1 and H_2 is the execution mode.

The data communication and synchronization are the major bottlenecks for vector and parallel processings, they may even take more CPU time than floating-point operations. The vector startup overhead is critical for computations of short vectors. Counting the number of arithmetic operations, which is the conventional method for evaluating the computational complexity of an algorithm, works well for scalar machines, but it may not be valid for vector and parallel processors. To measure the improvement due to the application of vector facilities, an operational measure S_v, called the speedup ratio, should be used. The speedup ratio S_v is defined as

$$S_v = \frac{\text{CPU time in scalar mode}}{\text{CPU time in vector mode}}$$

The speedups S_v for the inverse dynamics computation of general robots with different numbers of joints are given in the fourth column of Table 2. It can be seen that the larger the value n is, the more is the speed gains that can be achieved. The algorithm for the inverse dynamics computation of general robots involves many branch operations which detect whether the type of a joint is revolute or prismatic. These branch operations do not have a significant effect on the computation speed when the program runs in scalar mode. In other words, the computation speed H_1 will not change significantly when the types of joints are changed. However, these branch operations in the program hinder the performance of the algorithm in vector mode substantially. For example, using the same program, the computation speed in vector mode for a six-degrees-of-freedom robot with all revolute joints is 11,235 Hz (not shown in the Table 2) while the computation speed for a robot with five revolute joints and one prismatic joint is 9320 Hz. The computational speed of a robot with all revolute joints is increased about 22.55%.

It should be pointed out that the locations of prismatic joints in the kinematic chain of a robot may affect the vectorization and speed of the code, depending on the implementation of the algorithm. As an example, an n degrees-of-freedom robot manipulator with one prismatic joint which is located at the ith position , one may vectorize the robot inverse dynamics by breaking the vector length n into two separate vectors with lengths $(i-1)$ and $(n-i)$. However, this is generally unacceptable from vectorization point of view because this kind of separation will introduce an extra vector pipelining overhead and result in two short vectors. One way to solve this problem is to treat all joints as revolute joints in the first vector loop with the vector length n, whenever it is possible, under the assumption that the number of prismatic joints of a robot is less than that of revolute joints. Then, after the calculation of the first vector loop, the entry corresponding to the prismatic joint will be replaced in the second vector loop also with the vector length n, which involves a branch operation to detect if a joint is revolute or prismatic. Although this will introduce some unnecessary computations for prismatic joint i in the first loop, the computational speed gained due to vectorization for long vectors will compensate extra calculations in the first loop. This ideas about vectorization concerning prismatic joints can be best illustrated by vectorization of computations of the rotation matrices $\mathbf{R}(q_k, \mathbf{u}_{k0})$ of equation (1) in Appendix. The computation of $\mathbf{R}(q_k, \mathbf{u}_{k0})$ can be vectorized by $\mathbf{R}(q_k, \mathbf{u}_{k0}) = \mathbf{R}(\theta_k, \mathbf{u}_{k0})$ in the first loop with the vector length n for $k = 1, ..., n$. The rotation matrix $\mathbf{R}(q_i, \mathbf{u}_{i0}) = I$ for prismatic joint i can then be obtained in the second vector loop with branch operations which detect whether the ith joint is a prismatic joint. However, since we only vectorize the inner most vector loop, the branch operations in many cases can be placed in outer loops if multiple vector loops are involved. Our software for the inverse dynamics computation of general robots are so programmed, which can be found in [3]. Therefore, in general, the more the number of prismatic joints a robot has, the slower is the computational speed in vector mode. An ad hoc computer program, which takes the geometric configuration of a specific robot into consideration can run faster than a general robot program. For example, the prismatic joint of the robot with simulation results reported in Table 2 is located in joint 1. The computation speeds presented in Table 2 are based on the program written for general robots. For this particular case, the computation of $\mathbf{R}(q_k, \mathbf{u}_{k0})$ can be vectorized by $\mathbf{R}(q_k, \mathbf{u}_{k0}) = \mathbf{R}(\theta_k, \mathbf{u}_{k0})$ for a vector loop with vector length $k = 2, ..., n$ plus scalar processing for $\mathbf{R}(q_1, \mathbf{u}_{10}) = I$. The computation speed of a special program for this kind of robot with their first joint being a prismatic joint will certainly be faster than H_2 in Table 2 which is based on a program for general robots. However, unlike scalar processing, due to the vector pipelining and vector chaining, the difference of the CPU times for vector operations with vector lengths $(n-1)$ and n is insignificant. But, if the prismatic joint is not located at the first or last joint of a robot arm, the separation of vector n may not gain speed as is discussed before.

Since the branch operations for prismatic joints in a vector loop have significant effect on the computation speed, a separate program which does not have branch operations for the computation of the inverse dynamics for general robot manipulators with all revolute joints was developed. The computational frequency in both scalar and vector modes, and corresponding speedups for robots with different numbers of joints are also given in Table 2 for the cross comparison. In general, the speed H_4 in vector mode is about 1.38 times faster than the speed H_2; the speed H_3 in scalar mode is about 1.17 times faster than the speed

Figure 1: CPU time (in seconds) for the inverse dynamics of general revolute robots in both scalar and vector modes corresponding to different numbers of joints.

H_1. Therefore, the robot with all revolute joints should be favored from the computational speed point of view if pipelined vector processors are used. Of course, the actual selection of joint types depends on many other factors such as the repeatability of motion, rigidity, load-carrying capacity, and dexterity of a robot manipulator. It may be interesting to compare the computation speeds of the robot inverse dynamics on both supercomputer and personal computer although this is not the purpose of this paper. Since the same inverse dynamics program of a general six-degrees-of-freedom robot manipulator can be executed at the rate of about 150 Hz on an IBM PS/2 [3], [5], the computation speed of the inverse dynamics on a Cray X-MP is about 85 times faster than that on an IBM PS/2.

Figure 1 shows the CPU times versus the number of joints for computing a set of joint actuator torques when the relevant joint and end-effector information for a robot with all revolute joints is known. It is obvious that the CPU times in both scalar mode and vector mode are almost linearly dependent upon the number of joints. However, the slopes of two nearly linear dependences are different. Due to vector pipelining and chaining, the program in vector mode takes relatively less CPU time than that in scalar mode when the number of joints increases. It should be pointed out that, as the number of joints increases, the percentage of the CPU time for the computation of the recursive loops 5, 7, 11, 13, and 15 in Table 1 will also increase because the length for the vectorization of these loops remains a small value 3. These recursive equations are the bottleneck for the execution of the algorithm on pipelined vector processors.

4 Conclusions

Due to data dependence, the basic recursive structure of the Newton-Euler formulation of the robot inverse dynamics hinders vectorization of the algorithm. However, without any increase in the computational complexity, the developed vector algorithm, which still remains within the linear recursive framework, can significantly enhance the computational speed on pipelined vector processors. Although the computational frequency of the inverse dynamics on scalar processors is almost independent of the type of manipulator joints, the mixture of both the revolute and prismatic joints in a robot manipulator will substantially degrade the performance of the algorithm on vector processors. Robot manipulators with all revolute joints will have a better performance solely from the vector pipelining and vector chaining points of views.

It has been shown that the speedup S_v of vector processing over scalar processing is 1.82 for a general six-degrees-of-freedom robot manipulator. Although the CPU time required for the robot inverse dynamics computation is linear with respect to the number of joints in both scalar and vector processings, the growth rate of the CPU time in vector processing is significantly smaller than that in scalar processing. In general, the more is the number of joints, the higher the speedup. Therefore, this feature of vector processing can be exploited for computations involved in the control of redundant manipulators. The ideas of vectorization presented in this paper are general and they are applicable to other areas of robotics.

The vector break-even length of most vector processors currently impelmented in the VME board level is larger than the number of joints of most industrial robots, which inhibits its successful vectorization for applications in robotics. However, as the VLSI circuits technology advances, it is expected that the inexpensive general-purpose vector processors with better vector performance and a small vector break-even length will be available and their applications in robotics will be feasible in the near future.

5 Appendix

If the joint values $\mathbf{q}$, rates $\dot{\mathbf{q}}$, and accelerations $\ddot{\mathbf{q}}$ are known, the modified inertia force $\mathbf{F}_k^*$ and moment $\mathbf{N}_k^*$ can be computed recursively from $k = 1$ to n (the ground is defined as body 0 and the number of joints is n) using the zero reference position data as follows [3], [5]:

$$\mathbf{R}(q_k, \mathbf{u}_{k0}) = \begin{cases} \mathbf{R}(\theta_k, \mathbf{u}_{k0}) & \text{if joint } k \text{ is revolute, } k\text{—}R \\ \mathbf{I} & \text{if joint } k \text{ is prismatic, } k\text{—}P \end{cases} \tag{1}$$

$$\boldsymbol{\omega}_k^* = \begin{cases} \mathbf{R}^T(q_k, \mathbf{u}_{k0})\boldsymbol{\omega}_{k-1}^* + \dot{q}_k \mathbf{u}_{k0} & k\text{—}R \\ \mathbf{R}^T(q_k, \mathbf{u}_{k0})\boldsymbol{\omega}_{k-1}^* = \boldsymbol{\omega}_{k-1}^* & k\text{—}P \end{cases} \tag{2}$$

$$\boldsymbol{\alpha}_k^* = \begin{cases} \mathbf{R}^T(q_k, \mathbf{u}_{k0})(\boldsymbol{\alpha}_{k-1}^* + \ddot{q}_k \mathbf{u}_{k0} + \boldsymbol{\omega}_{k-1}^* \times \dot{q}_k \mathbf{u}_{k0}) & k\text{—}R \\ \mathbf{R}^T(q_k, \mathbf{u}_{k0})\boldsymbol{\alpha}_{k-1}^* = \boldsymbol{\alpha}_{k-1}^* & k\text{—}P \end{cases} \tag{3}$$

$$\mathbf{c}_k^* = \begin{cases} \mathbf{c}_{k0} & k\text{—}R \\ \mathbf{c}_{k0} & k\text{—}P\text{, the collar is on the } k\text{th link} \\ \mathbf{c}_{k0} + q_k \mathbf{u}_{k0} & k\text{—}P\text{, the collar is on the } (k-1)\text{th link} \end{cases} \tag{4}$$

$$\mathbf{d}_k^* = \begin{cases} \mathbf{d}_{k0} & (k+1)\text{—}R \\ \mathbf{d}_{k0} & (k+1)\text{—}P\text{, the collar is on the } k\text{th link} \\ \mathbf{d}_{k0} + q_{k+1}\mathbf{u}_{k+1,0} & (k+1)\text{—}P\text{, the collar is on the } (k+1)\text{th link} \end{cases} \tag{5}$$

$$\begin{aligned} \mathbf{a}^*_{Q_k^{k-1}} &= \mathbf{a}^*_{G_{k-1}} + \boldsymbol{\omega}^*_{k-1} \times (\boldsymbol{\omega}^*_{k-1} \times \mathbf{d}^*_{k-1}) + \boldsymbol{\alpha}^*_{k-1} \times \mathbf{d}^*_{k-1} \\ &= \mathbf{a}^*_{G_{k-1}} + (\boldsymbol{\omega}^*_{k-1} \cdot \mathbf{d}^*_{k-1})\boldsymbol{\omega}^*_{k-1} - (\boldsymbol{\omega}^*_{k-1} \cdot \boldsymbol{\omega}^*_{k-1})\mathbf{d}^*_{k-1} + \boldsymbol{\alpha}^*_{k-1} \times \mathbf{d}^*_{k-1} \end{aligned} \tag{6}$$

$$\mathbf{a}^*_{Q_k^k} = \begin{cases} \mathbf{R}^T(q_k, \mathbf{u}_{k0})\mathbf{a}^*_{Q_k^{k-1}} & k\text{—}R \\ \mathbf{R}^T(q_k, \mathbf{u}_{k0})(\mathbf{a}^*_{Q_k^{k-1}} + \ddot{q}_k\mathbf{u}_{k0} + 2\boldsymbol{\omega}^*_{k-1} \times \dot{q}_k\mathbf{u}_{k0}) & \\ \qquad = \mathbf{a}^*_{Q_k^{k-1}} + \ddot{q}_k\mathbf{u}_{k0} + 2\boldsymbol{\omega}^*_{k-1} \times \dot{q}_k\mathbf{u}_{k0} & k\text{—}P \end{cases} \tag{7}$$

$$\begin{aligned} \mathbf{a}^*_{G_k} &= \mathbf{a}^*_{Q_k^k} + \boldsymbol{\omega}^*_k \times (\boldsymbol{\omega}^*_k \times \mathbf{c}^*_k) + \boldsymbol{\alpha}^*_k \times \mathbf{c}^*_k \\ &= \mathbf{a}^*_{Q_k^k} + (\boldsymbol{\omega}^*_k \cdot \mathbf{c}^*_k)\boldsymbol{\omega}^*_k - (\boldsymbol{\omega}^*_k \cdot \boldsymbol{\omega}^*_k)\mathbf{c}^*_k + \boldsymbol{\alpha}^*_k \times \mathbf{c}^*_k \end{aligned} \tag{8}$$

$$\mathbf{F}^*_k = -m_k\mathbf{a}^*_{G_k} \tag{9}$$

$$\begin{aligned} \mathbf{N}^*_k &= -\boldsymbol{\omega}^*_k \times (\mathbf{M}_{k0}\boldsymbol{\omega}^*_k) - \mathbf{M}_{k0}\boldsymbol{\alpha}^*_k \\ &= -\boldsymbol{\Omega}^*_k\mathbf{M}_{k0}\boldsymbol{\omega}^*_k - \mathbf{M}_{k0}\boldsymbol{\alpha}^*_k \end{aligned} \tag{10}$$

The joint actuator forces or torques τ_k can then be calculated recursively from $k = n$ to 1 as follows

$$\mathbf{f}^*_k = \mathbf{R}(q_k, \mathbf{u}_{k0})(\mathbf{f}^*_{k+1} - \mathbf{F}^*_k) \tag{11}$$

$$\mathbf{n}^*_k = \mathbf{R}(q_k, \mathbf{u}_{k0})(\mathbf{n}^*_{k+1} + \mathbf{b}^*_k \times \mathbf{f}^*_{k+1} - \mathbf{c}^*_k \times \mathbf{F}^*_k - \mathbf{N}^*_k) \tag{12}$$

$$\tau_k = \begin{cases} \mathbf{n}^*_k \cdot \mathbf{u}_{k0} & k\text{—}R \\ \mathbf{f}^*_k \cdot \mathbf{u}_{k0} & k\text{—}P \end{cases} \tag{13}$$

To increase the computational efficiency, the starting angular velocity $\boldsymbol{\omega}^*_0$ and accelerations $\boldsymbol{\alpha}^*_0$, $\mathbf{a}^*_{G_0}(= \mathbf{a}_{G_0})$ of the ground for the recursive computation are set to

$$\boldsymbol{\omega}^*_0 = \boldsymbol{\alpha}^*_0 = \mathbf{0}, \qquad \mathbf{a}^*_{G_0} = (0, \quad 0, \quad 9.81\,\mathrm{m/s^2})^T \tag{14}$$

The external force $\mathbf{f}_e$ and moment $\mathbf{n}_e$ acting on the end-effector are usually expressed at the target position in the base coordinate system. To incorporate these into the recursive structure, define $\mathbf{f}_{n+1} = -\mathbf{f}_e$ and $\mathbf{n}_{n+1} = -\mathbf{n}_e$. The external force and moment can be formulated in the zero reference position as follows:

$$\mathbf{R}_{B,Z} = \mathbf{R}_{BCP,ZRP} = [\mathbf{u}_{to} \times \mathbf{u}_{a0} \;\; \mathbf{u}_{t0} \;\; \mathbf{u}_{a0}] \tag{15}$$

$$\mathbf{R}_{B,T} = \mathbf{R}_{BCP,TP} = [\mathbf{u}_t \times \mathbf{u}_a \;\; \mathbf{u}_t \;\; \mathbf{u}_a] \tag{16}$$

$$\mathbf{R}_{Z,T} = \mathbf{R}_{ZRP,TP} = \mathbf{R}_{B,T}\mathbf{R}^T_{B,Z} \tag{17}$$

$$\mathbf{f}^*_{n+1} = -\mathbf{R}^T_{Z,T}\mathbf{f}_e, \qquad \mathbf{n}^*_{n+1} = -\mathbf{R}^T_{Z,T}\mathbf{n}_e \tag{18}$$

6 References

1. Chen, C. L., Lee, C. S. G., and Hou, E. S. H.: Efficient Scheduling Algorithms for Robot Inverse Dynamics Computation on a Multiprocessor System, *Proc. IEEE Intl. Conf. Robotics and Automation*, Philadelphia, Pennsylvania, Vol. 2, 1988, pp. 1146-1151,

2. Cheng, H. H.: Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP, *(IEEE) Computer*, Vol. 22, No. 9, September, 1989a, pp. 31-46.

3. Cheng, H. H.: *Computational Aspects of Robot Manipulation*, PhD Dissertation, University of Illinois at Chicago, Chicago, 1989b.

4. Cheng, H. H. and Gupta, K. C.: Vectorization of Robot Dynamics on a Pipelined Vector Processor, *IEEE International Conf. on Robotics and Automation*, Sacramento, CA, April 9-11, 1991, Vol 1, pp. 96-101.

5. Cheng, H. and Gupta, K. C.: An Efficient Manipulator Dynamics Formulation Based upon Newton-Euler Equations and the ZRP Method, *Flexible Mechanisms, Dynamics, and Analysis*, ed. by G. Kinzel, et al., *1992 ASME 22nd Biennial Mechanisms Conference*, Scottsdale, AZ, September. 13-16, DE-Vol. 47, pp. 81-87.

6. CSP, Inc.: *SuperCard*, Billerica, MT, 1991.

7. Gupta, K. C: A Note on Position Analysis of Manipulator, *Proceedings of 7th Applied Mechanism Conference*, Kansas City, 1981; also *Mechanism and Machine Theory*, Vol. 19, No. 1, pp. 5-8.

8. Kasahara, H. and Narita, S.: Parallel Processing of Robot-Arm Control Computation on a Multiprocessor System, *IEEE J. of Robotics and Automation*, Vol. RA-1, No. 2, June, 1985, pp. 104-113

9. Kogge, P. M. and Stone, H. S.: A Parallel Algorithm for Efficient Solution of a General Class of Recurrence Equations, *IEEE Trans. Computer*, Vol. C-22, NO. 8, August 1973, pp. 789-793.

10. Kung, H. T.: Why Systolic Architecture?, *(IEEE) Computer*, Jan. 1982, pp. 37-46.

11. Lathrop, L. H.: Parallelism in Manipulator Dynamics, *Intl. J. Robotics Res.*, Vol. 4, No. 2, Summer 1985, pp. 80-102.

12. Lee, C. S. G. and Chang, P. R.: Efficient Parallel Algorithm for Robot Inverse Dynamics Computation, *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-16, No. 4, July 1986, pp. 532-542.

13. Luh, J. Y. S., Walker, M. W., and Paul, R. P.: Resolved Acceleration Control of Mechanical Manipulators, *IEEE Trans. Automatic Control*, 25(3), 1980a, pp. 468-474.

14. Luh, J. Y. S., Walker, M. W., and Paul, R. P.: On-Line Computational Scheme for Mechanical Manipulators, *J. Dynamics, Measurement, Control*, Vol. 102, 1980b, pp. 69-76.

IEEE Trans. on Robotics and Automation, Vol. 9, No. 6, December, 1993, pp. 858-863.

15. Luh, J. Y. S. and Lin, C. S.: Scheduling of Parallel Computation for a Computer Controlled Mechanical Manipulator, *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-12, No. 2, Mar. 1982, pp. 214-234.

16. McMillan, S., Orin, D. E., and Sadayappan, P.: Real-Time Robot Dynamic Simulation on a Vector/Parallel Supercomputer, *Proc. of IEEE Int. Conf. on Robotics and Automation*, Sacramento, CA, pp. 1836-1841, April, 1991.

17. Schönauer, W.: *Scientific Computing on Vector Computers*, New York, North-Holland, 1987.

18. SKY Computers, Inc.: *The SKYbolt Application Accelerator*, Fairport, NY, 1991.