LIST OF REFERENCES

1. Roberts, L.G. "Homogeneous Matrix Representation and Manipulation of N-dimensional Constructs." Document No. MS1045, Lincoln Laboratory, Massachusetts Institute of Technology, 1965.
2. Pieper, D.L. "The kinematics of Manipulators under Computer Control." AIM-72, Stanford, Calif. : Stanford University Artificial Intelligence Laboratory, 1968.
3. Paul, R.P.; Stevenson, C.N. "Kinematics of Robot Wrists." Robotic Research 21 (Spring 1983): 31-38.
4. Renaud, M. "Geometric and kinematic Models of a Robot Manipulator: Calculation of the Jacobian Matrix and Its Inverse." Proc. 11th ISIR Tokyo, Japan, 1981.
5. Lee, C.S.G.; Chung, M.J. "An Adaptive Control Strategy for Computer Based Manipulators." Proc. of the 21st IEEE Conference on Decision and Control, Dec. 1982: 95-100.
6. Goldenberg, A.A.; Apkarian, J.A. and Smith, H.W. "An Approach to Adaptive Control of Robot Manipulators Using the Computed Torque Technique." ASME Transactions on Dynamic System, Measurement, and Control, Mar. 1989, Vol. 111/1.
7. Bitmead, R.R.; Gevers, M. and Wertz, V. "Adaptive Optimal Control: the Thinking Man's GPC." Prentice Hall, Australia, 1990.
8. Lehtomaki, N.A.; Sandell Jr, N.R. and Athans, M. "Robustness Results in Linear Quadratic Gaussian Based Multivariable Control Design." IEEE Transactions on Automatic Control, Vol. AC-26: 75-92, 1981.
9. de Souza, C.E.; Gevers, M. and Goodwin, C.G. "Riccati Equations in Optimal Filtering of Nonstabilizable Systems Having Singular State Transition Matrices." IEEE Transaction on Automatic Control, Vol. AC-31: 831-838, 1986.
10. Riedle, B.D. and Kokotovic, P.V. "Integral Manifolds of Slow Adaptation." IEEE Transactions on Automatic Control, AC-31: 316-323, 1986.

11. Riedle, B.D. and Kokotovic, P.V. "Stability Bounds for Slow Adaptation: an Integral Manifold Approach." Proc. 2nd IFAC Workshop on Adaptive System in Control and Signal Processing, Lund, Sweden, 1986.

APPENDIX

APPENDIX A

Theoretical Background

In this section we shall present the basic mathematical issues of the linear quadratic regulator (LQR) and the RLS identifier to have sufficient theoretical background for guiding our experimental evaluation on adaptive control of a manipulator. The experimental evaluation as described in details in Chapter 4 is to justify by actual hardware experiments the applicability of a certainty equivalence control law designed to be used with a robot manipulator. This certainty equivalence control law involves cooperation of the LQ regulator and the recursive least square (RLS) identifier augmented with feedforward compensation from nominal model calculation to suppress large nonlinear dynamics of a manipulator whose calculated model is also derived from another RLS identifier. Details of the control law formulation are presented in Section 3.2. This section aims to introduce the notation, lay dawn mathematical foundation and reveal some important facts needed to be understood before we can go on to the detail analysis of dynamic interplay between the LQR and the RLS identifier which is the crucial discussion to applicability of the certainly equivalence control law.

A.1. The Finite Horizon Regulator

Consider a linear system with state-space representation,

$$\mathbf{x}_{k+1} = \mathbf{F}\mathbf{x}_k + \mathbf{G}\mathbf{u}_k \qquad \text{(a.1)}$$

where $\mathbf{x}_k$ is the n-dimensional state vector, $\mathbf{u}_k$ is the m-dimensional vector of the control input, $\mathbf{F}$ and G are the system and the input matrix respectively. A finite horizon optimal control problem

can be formulated by assuming that we wish to find the control sequence $\mathbf{u}_k$ that will minimize the quadratic performance index of the form

$$J(N,\mathbf{x}_k) = \mathbf{x}_{k+N}^T \mathbf{P}_0 \mathbf{x}_{k+N} + \sum_{j=0}^{N-1} \left\{ \mathbf{x}_{k+j}^T \mathbf{Q} \mathbf{x}_{k+j} + \mathbf{u}_{k+j}^T \mathbf{R} \mathbf{u}_{k+j} \right\} \tag{a.2}$$

where N denotes the number of steps that the index looks ahead for data of the system states. $\mathbf{P}_0, \mathbf{Q}, \mathbf{R}$ are presumed to be non-negative, symmetric matrices. The solution of this LQR problem may be directly given in closed loop form as described by iterating the following Riccati Difference Equation (RDE) from the matrix $\mathbf{P}_0$ as the initial condition.

$$\mathbf{P}_{j+1} = \left(\mathbf{F} + \mathbf{G}\mathbf{K}_j\right)^T \mathbf{P}_j \left(\mathbf{F} + \mathbf{G}\mathbf{K}_j\right) + \mathbf{Q} + \mathbf{K}_j^T \mathbf{R} \mathbf{K}_j \tag{a.3}$$

where $\mathbf{K}$ is the state feedback gain whose description is

$$\mathbf{K}_j = -\left(\mathbf{G}^T \mathbf{P}_j \mathbf{G} + \mathbf{R}\right)^{-1} \mathbf{G}^T \mathbf{P}_j \mathbf{F} \tag{a.4}$$

The control sequences that minimize the above performance index can be derived backward in time from

$\mathbf{u}_{k+N-1}$ to $\mathbf{u}_k$ as

$$\mathbf{u}_{k+N-j} = \mathbf{K}_{j-1} \mathbf{x}_{k+N-j}, \; j = 1,2,\ldots,N \tag{a.5}$$

From the above, it follows that the evaluated optimal cost is quadratic in $\mathbf{x}_k$ that is

$$J\left(N,\mathbf{x}_k\right)^* = \mathbf{x}_k^T \mathbf{P}_N \mathbf{x}_k \tag{a.6}$$

where $\mathbf{P}_N$ is the end matrix solution of the RDE (a.3).

A.2. The Infinite Horizon Regulator

The finite horizon LQR problem presented above yields a time-varying control law, although the controlled plant and the weighting matrices, Q and R are time-invariant. However if we pose the quadratic performance index evaluated up to infinite state measurements, i.e., allow N to approach infinity, we will see that the dependence on the number of steps, N is naturally diminished and under appropriate conditions, the solution of the RDE should converge to a unique, constant matrix $\mathbf{P}_\infty$ as j iterated to infinity. Therefor, a stationary LQR should be obtained by implementing a constant state feedback gain K derived from the constant solution $\mathbf{P}_\infty$ to obtain the control input $\mathbf{u}_k$. This is indeed the case for that we can pose

$$J(\mathbf{x}_k) = \lim_{N\to\infty} \frac{1}{N} J(N, \mathbf{x}_k), \tag{a.7}$$

for an infinite horizon LQR and we obtain $\mathbf{P}_\infty$ governed by the Algebraic Riccati Equation (ARE)

$$\mathbf{P}_\infty = (\mathbf{F}+\mathbf{GK})^T \mathbf{P}_\infty (\mathbf{F}+\mathbf{GK}) + \mathbf{Q} + \mathbf{K}^T\mathbf{RK} \tag{a.8}$$

where

$$\mathbf{K} = -\left(\mathbf{G}^T\mathbf{P}_\infty\mathbf{G}+\mathbf{R}\right)^{-1}\mathbf{G}^T\mathbf{P}_\infty\mathbf{F}, \tag{a.9}$$

and the corresponding stationary control law is

$$\mathbf{u}_k = \mathbf{K}\mathbf{x}_k \tag{a.10}$$

Conformably with the finite horizon case, the evaluated optimal cost can be expressed as

$$J(\mathbf{x}_k)^* = \mathbf{x}_k^T \mathbf{P}_\infty \mathbf{x}_k \qquad \text{(a.11)}$$

A.3. Asymptotic Stability

Our one-step-ahead horizon regulator is a finite horizon LQR with N = 1 whose stability is not guaranteed by its optimal designs. The finite horizon LQ feedback control strategy does not guarantee closed loop asymptotic stability. Conversely, the closed loop asymptotic stability result can be derived from the stationary infinite horizon strategy. Indeed, the fundamental LQR asymptotic stability result relies upon the properties of the Algebraic Riccati Equation (ARE) (a.8) obtained from the infinite horizon LQR problem. According to this fact, we will state the following stability theorem as it represents fundamental stability result underlying all stability and robustness analysis of the one-step-ahead controller in the sequel.

Theorem A.1

Consider the ARE associated with an infinite horizon LQR problem

$$\mathbf{P} = (\mathbf{F} + \mathbf{GK})^T \mathbf{P}(\mathbf{F} + \mathbf{GK}) + \mathbf{Q} + \mathbf{K}^T \mathbf{R}\mathbf{K} \qquad \text{(a.12)}$$

where

$$\mathbf{K} = -\left(\mathbf{G}^T \mathbf{P}\mathbf{G} + \mathbf{R}\right)^{-1} \mathbf{G}^T \mathbf{P}\mathbf{F} \qquad \text{(a.13)}$$

with $\mathbf{Q} \geq \mathbf{0}$ *and* $\mathbf{R} > \mathbf{0}$

(a). $\mathbf{P}$ *exists iff there are no modes of* $[\mathbf{F},\mathbf{G}]$ *and* $\left[\mathbf{F},\mathbf{Q}^{1/2}\right]$ *that are simultaneously*

- Observable

- *Uncontrollable*

- *Unstable*

(b). If $\mathbf{P}$ *exists, then* $\mathbf{P} > \mathbf{0}$ *iff* $[\mathbf{F}, \mathbf{Q}^{1/2}]$ *is an observable pair.*

(c). If $[\mathbf{F}, \mathbf{G}]$ *is stabilizable and* $[\mathbf{F}, \mathbf{Q}^{1/2}]$ *detectable, then* $(\mathbf{F} + \mathbf{GK})$ *is asymptotically stable.*

Proof :

(a). From the optimal cost for an infinite horizon LQR, we can pose it as

$$J(\mathbf{x}_k) = \sum_{j=0}^{\infty} \left\{ \mathbf{x}_{k+j}^T \mathbf{Q} \mathbf{x}_{k+j} + \mathbf{u}_{k+j}^T \mathbf{R} \mathbf{u}_{k+j} \right\} \tag{a.14}$$

The quadratic term in $\mathbf{x}_k$ can be thought of as the product of the measurement output $\mathbf{y}_k$ from the output equation as follows,

$$\mathbf{y}_k = \mathbf{Q}^{1/2} \mathbf{x}_k . \tag{a.15}$$

Referring to the plant state space description (a.1), $\mathbf{Q}^{1/2}$ denotes the *symmetric square root of Q*, and using (a.14), the optimal cost is then

$$J(\mathbf{x}_k) = \sum_{j=0}^{\infty} \left\{ \mathbf{y}_{k+j}^T \mathbf{y}_{k+j} + \mathbf{u}_{k+j}^T \mathbf{R} \mathbf{u}_{k+j} \right\} = \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k \tag{a.16}$$

Suppose that $\mathbf{P}$ exists but there are modes that are simultaneously observable, uncontrollable and unstable. Then,

$$J(\mathbf{x}_k) \; < \infty \tag{a.17}$$

and some unstable mode is observable in $\mathbf{y}$ and is also uncontrollable. Hence, for any control, $\mathbf{y}_{k+j}^T\mathbf{y}_{k+j}$ will diverse to infinity. But this contradicts with $J(\mathbf{x}_k) < \infty$. This proves that $\mathbf{P}$ exits implies no observable, unstable uncontrollable modes. Now we suppose the converse that there are no modes that are observable, unstable uncontrollable, but this time $\mathbf{P}$ does not exists. Then we get

$$J(\mathbf{x}_k) = \infty, \text{ for any finite } \mathbf{x}_k \qquad \text{(a.18)}$$

Since all unstable modes are uncontrollable by assumption. By definition of being uncontrollable, there exists some control such that any states can be forced to zero. The control obviously yields that $J(\mathbf{x}_k)$ must converge to limited value or $J(\mathbf{x}_k) < \infty$. And this contradicts the assumption of $J(\mathbf{x}_k) = \infty$. Hence, conclusion is that the absence of observable, unstable uncontrollable modes implies $\mathbf{P}$ exists. This completes the proof (a).

(b). Suppose that $P > 0$, but $\left[\mathbf{F}, \mathbf{Q}^{1/2}\right]$ is not observable. Then,

$$J(\mathbf{x}_k) = \sum_{j=0}^{\infty}\left\{\mathbf{y}_{k+j}^T\mathbf{y}_{k+j} + \mathbf{u}_{k+j}^T\mathbf{R}\mathbf{u}_{k+j}\right\} = \mathbf{x}_k^T\mathbf{P}\mathbf{x}_k \ > \infty \quad , \qquad \text{(a.19)}$$

and for some $\mathbf{x}_k \neq 0$ where $\left[\mathbf{F}, \mathbf{Q}^{1/2}\right]$ is not observable, we get

$$\sum_{j=0}^{\infty}\left\{\mathbf{y}_{k+j}^T\mathbf{y}_{k+j}\right\} = 0 \qquad \text{(a.20)}$$

Thus,

$$J(\mathbf{x}_k) = \sum_{j=0}^{\infty}\left\{\mathbf{u}_{k+j}^T\mathbf{R}\mathbf{u}_{k+j}\right\} \quad , \qquad \text{(a.21)}$$

which implies that the optimum cost $J(\mathbf{x}_k) = 0$ by choosing $\mathbf{u}_{k+j} = 0$, for $j = 0$ to ∞, to minimize $J(\mathbf{x}_k)$. This result contradicts the assumption $J(\mathbf{x}_k) > \infty$ above. This proves that $P > 0$ implies the pair $[\mathbf{F}, \mathbf{Q}^{1/2}]$ is observable. Next , we proof the converse-Suppose $[\mathbf{F}, \mathbf{Q}^{1/2}]$ is observable, but $P = 0$. We get, for some $\mathbf{x}_k \neq 0$,

$$J(\mathbf{x}_k) = \sum_{j=0}^{\infty} \left\{ \mathbf{y}_{k+j}^T \mathbf{y}_{k+j} + \mathbf{u}_{k+j}^T \mathbf{R} \mathbf{u}_{k+j} \right\} = \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k = 0 \quad \text{(a.22)}$$

$[\mathbf{F}, \mathbf{Q}^{1/2}]$ being observable implies that $\mathbf{x}_k \neq 0 \rightarrow \mathbf{y}_k \neq 0$ for $j = 0$ to ∞ Since $\mathbf{y}_{k+j} \neq 0$, then $J(\mathbf{x}_k) = \sum_{j=0}^{\infty} \left\{ \mathbf{y}_{k+j}^T \mathbf{y}_{k+j} + \mathbf{u}_{k+j}^T \mathbf{R} \mathbf{u}_{k+j} \right\} > 0$, which contradicts the assumption $P = 0$. This proves that $[\mathbf{F}, \mathbf{Q}^{1/2}]$ is observable implies $P > 0$ .Thus, part (b). is proved.

(c). Consider the closed loop system

$$\mathbf{x}_{k+1} = (\mathbf{F} + \mathbf{G}\mathbf{K})\mathbf{x}_k, \quad \text{(a.23)}$$

we choose the evaluated optimal cost as a Lyapunov function candidate for the above system.

$$V(\mathbf{x}_k) = \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k. \quad \text{(a.24)}$$

If we assume the pair $[\mathbf{F}, \mathbf{Q}^{1/2}]$ is observable, from the proof of part (c)., then

$$V(\mathbf{x}_k) = \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k > \mathbf{0}, \text{ for any } \mathbf{x}_k \neq 0,$$

and

$$V(\mathbf{x}_k) = \mathbf{x}_k^T \mathbf{P} \mathbf{x}_k = \mathbf{0}, \text{ only if when } \mathbf{x}_k = 0.$$

Thus, $V(\mathbf{x}_k)$ is positive definite. Next, we evaluate $\Delta V(\mathbf{x}_k) = V(\mathbf{x}_{k+1}) - V(\mathbf{x}_k)$ and examine whether $\Delta V(\mathbf{x}_k)$ is negative definite.

$$\Delta V(\mathbf{x}_k) = \mathbf{x}_{k+1}^T \mathbf{P}\mathbf{x}_{k+1} - \mathbf{x}_k^T \mathbf{P}\mathbf{x}_k$$

$$= \mathbf{x}_k^T \left[(\mathbf{F} + \mathbf{GK})^T \mathbf{P}(\mathbf{F} + \mathbf{GK}) - \mathbf{P} \right] \mathbf{x}_k \qquad \text{(a.25)}$$

Using the ARE (a.12) with the above,

$$\Delta V(\mathbf{x}_k) = -\mathbf{x}_k^T \left[\mathbf{Q} + \mathbf{K}^T \mathbf{R}\mathbf{K} \right] \mathbf{x}_k \qquad \text{(a.26)}$$

Since $\left[\mathbf{Q} + \mathbf{K}^T \mathbf{R}\mathbf{K} \right]$ is positive definite, then $\Delta V(\mathbf{x}_k)$ is negative definite. Therefore $V(\mathbf{x}_k)$ is indeed a Lyapunov function of the system (a.23). This implies that the system (a.23) is asymptotically stable. We observe that the optimal control is only influenced through $\mathbf{P}$ by the modes observable in $\mathbf{y}$. If there exists modes that are unobservable, these modes must be asymptotically stable by itself without any effort of the control. This is the definition of *detectability*. Thereby the existence of a necessary assumption that the $\left[\mathbf{F}, \mathbf{Q}^{1/2} \right]$ is detectable is insisted. The condition of $[\mathbf{F}, \mathbf{G}]$ being stabilizable is introduced to support the existence of $\mathbf{P}$ solution, hence stabilizing $\mathbf{K}$ as evident in the proof of part (a). Thus, part (c) is proved

At this point we go on further with the asymptotic stability proof of the one-step-ahead horizon regulator. The proof will be performed in the same way as Theorem A.1 that the Lyapunov's second method is applied. This is because that this way of proving will introduce, as evident later, the Fake Riccati Equation (FARE) that reveals close connection between the finite and infinite horizon LQR in the issue of stability and robustness. The connection through the FARE will transform the finite horizon LQR problem into the infinite horizon LQR problem in stability robustness sense.

Before we go on to state the stability theorem of the one-step-ahead horizon LQR, we will recast the form of the performance index presented in Section 3.1.3 into the general expression of the performance index for a finite horizon LQR. Refer to Section 3.1.3, we formulated a one-step-ahead optimization control problem for state regulation of the linearized perturbed model of a manipulator by posing the performance index as

$$J_k = \mathbf{x}_{k+1}^T \mathbf{Q}^* \mathbf{x}_{k+1} + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k \qquad \text{(a.27)}$$

where J_k is the optimal cost evaluated at $\mathbf{x}_k$. $\mathbf{Q}^*$ and $\mathbf{R}$ represent the weighting matrices presumed to be non negative definite and positive definite respectively. * superscript of $\mathbf{Q}$ distinguishes the weighting in Section 3.1.3 from the one in the general performance index expression for the finite horizon problem described above. Equation (a.27) is equivalent to

$$J(1,\mathbf{x}_k) = \mathbf{x}_{k+1}^T \mathbf{P}_0 \mathbf{x}_{k+1} + \mathbf{x}_k^T \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k, \qquad \text{(a.28)}$$

and

$$\mathbf{Q} = (\mathbf{F} + \mathbf{GK})^T (\mathbf{P}_0 - \mathbf{Q}^*)(\mathbf{F} + \mathbf{GK}). \qquad \text{(a.29)}$$

Since the weighting matrices $\mathbf{Q}, \mathbf{Q}^*$ and $\mathbf{P}_0$ all are arbitrary, the solution of this problem is invariant to the form of the performance index, (a.27) or (a.28). It is now appropriate to state the stability theorem for the one-step-ahead horizon LQR.

Theorem A.2

Consider the RDE associated the one-step-ahead horizon LQR,

$$\mathbf{P}_1 = (\mathbf{F}+\mathbf{GK})^T \mathbf{P}_0 (\mathbf{F}+\mathbf{GK}) + \mathbf{Q} + \mathbf{K}^T\mathbf{RK} \tag{a.30}$$

where

$$\mathbf{K} = -\left(\mathbf{G}^T\mathbf{P}_0\mathbf{G} + \mathbf{R}\right)^{-1}\mathbf{G}^T\mathbf{P}_0\mathbf{F} \tag{a.31}$$

and consider the Fake Riccati Equation (FARE) associated with the RDE above

$$\mathbf{P}_1 = (\mathbf{F}+\mathbf{GK})^T \mathbf{P}_1 (\mathbf{F}+\mathbf{GK}) + \overline{\mathbf{Q}} + \mathbf{K}^T\mathbf{RK} \tag{a.32}$$

where

$$\overline{\mathbf{Q}} = \mathbf{Q} - (\mathbf{P}_1 - \mathbf{P}_0) \tag{a.33}$$

Provided

- $[\mathbf{F},\mathbf{G}]$ *is stabilizable,*
- $\left[\mathbf{F},\mathbf{Q}^{1/2}\right]$ *is detectable,*
- $\mathbf{P}_0 \geq \mathbf{P}_1$

Then

- $(\mathbf{F}+\mathbf{GK})$ *is asymptotically stable.*

Proof:

The FARE is introduced to rewrite $\mathbf{P}_1$ as a solution of the ARE with $\mathbf{Q}$ replaced by $\overline{\mathbf{Q}}$, but the same $\mathbf{R}$. Indeed, the FARE is just another rewriting of the RDE in terms of the new weighting $\overline{\mathbf{Q}}$ whose definition is as (a.33). By this means, we can then proceed to draw conclusion of asymptotic stability by exploiting results in the proof of Theorem A.1. Suppose that we are given, $\left[\mathbf{F}, \overline{\mathbf{Q}}^{1/2}\right]$ being observable, and by using the result in the proof of part (a) of Theorem A.1, we obtain $\mathbf{P}_1$ being positive definite. Hence, we can use the optimal cost of the one-step-ahead horizon LQR as a Lyapunov function candidate. That is there exists

$$V(\mathbf{x}_k) = J(1, \mathbf{x}_k) = \mathbf{x}_k^T \mathbf{P}_1 \mathbf{x}_k > 0, \text{ for any } \mathbf{x}_k \neq 0. \qquad \text{(a.34)}$$

$V(\mathbf{x}_k)$ is a positive definite function. To follow the Lyapunov's second method, we pose the forward difference of $V(\mathbf{x}_k)$.

$$\begin{aligned} \Delta V(\mathbf{x}_k) &= V(\mathbf{x}_{k+1}) - V(\mathbf{x}_k) \\ &= \mathbf{x}_{k+1}^T \mathbf{P}_1 \mathbf{x}_{k+1} - \mathbf{x}_k^T \mathbf{P}_1 \mathbf{x}_k \\ &= \mathbf{x}_k^T \left[(\mathbf{F} + \mathbf{GK})^T \mathbf{P}_1 (\mathbf{F} + \mathbf{GK}) - \mathbf{P}_1 \right] \mathbf{x}_k. \end{aligned} \qquad \text{(a.35)}$$

Using the FARE (a.32), we obtain

$$\Delta V(\mathbf{x}_k) = -\mathbf{x}_k^T \left[\overline{\mathbf{Q}} + \mathbf{K}^T \mathbf{R} \mathbf{K} \right] \mathbf{x}_k. \qquad \text{(a.36)}$$

Since the assumption $\mathbf{P}_0 \geq \mathbf{P}_1$ implies $\overline{\mathbf{Q}}$ is non-negative definite and $\mathbf{R}$ is positive definite, hence we conclude that $\Delta V(\mathbf{x}_k)$ is negative definite. Thereby $V(\mathbf{x}_k)$ is indeed a Lyapunov function of

the system (a.23). The system is then asymptotically stable. Like in part (c) of the proof of Theorem A.1, minimum condition on the pair $\left[\mathbf{F},\overline{\mathbf{Q}}^{1/2}\right]$ is detectability and $\left[\mathbf{F},\mathbf{G}\right]$ supports the existence of the stabilizing $\mathbf{K}$. In addition, we have a result that $\left[\mathbf{F},\mathbf{Q}^{1/2}\right]$ detectable implies $\left[\mathbf{F},\overline{\mathbf{Q}}^{1/2}\right]$ detectable, if $\overline{\mathbf{Q}} > \mathbf{Q}$. The result can be proved by the followings:

The assumption that $\left[\mathbf{F},\mathbf{Q}^{1/2}\right]$ is detectable requires that every unstable modes of $\mathbf{F}$ must be observed by $\mathbf{Q}^{1/2}$ that is for every eigenvalues of $\mathbf{F}$ possessing magnitude greater than unity, its associated left eigenvector $\mathbf{x}$ makes $\mathbf{x}\mathbf{Q}^{1/2} \neq 0$. If we have $\overline{\mathbf{Q}}$ such that

$$\overline{\mathbf{Q}} \geq \mathbf{Q} \qquad \text{(a.37)}$$

We get $\mathbf{x}\mathbf{Q}^{1/2} \neq 0$ implies $\mathbf{x}\overline{\mathbf{Q}}^{1/2} \neq 0$. Hence, $\left[\mathbf{F},\mathbf{Q}^{1/2}\right]$ detectable implies $\left[\mathbf{F},\overline{\mathbf{Q}}^{1/2}\right]$ detectable. From the assumption $\mathbf{P}_0 \geq \mathbf{P}_1$, (a.37) is indeed the case. Thus, we complete the proof.

A.4. Robustness Considerations

In this section we commence with presentation of the robustness results of a unity feedback control system as the LQR can be interpret into this standard type of linear control systems. The standard configuration of a unity feedback system can be depicted in Figure A.1.

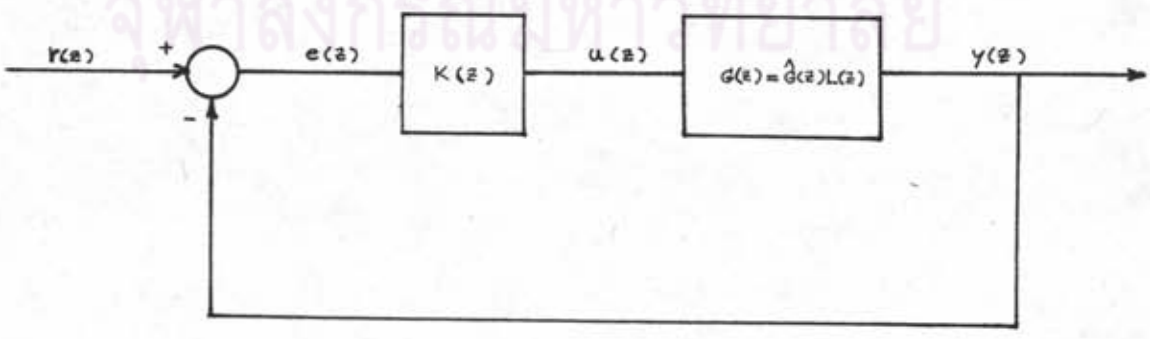

Figure A.1: A unity feedback system.

$\mathbf{K}(z)$ and $\mathbf{G}(z)$ are transfer functions of the designed controller and the true plant respectively. We use the adjective " true " to denote a plant model which is presumed to be strictly proper. We suppose that we have an identified plant model $\hat{\mathbf{G}}(z)$ available from identification process. We shall call $\hat{\mathbf{G}}(z)$ the nominal plant transfer function in the sequel. And we assume that $\mathbf{G}(z)$ and $\hat{\mathbf{G}}(z)$ are related by the multiplicative perturbation $\mathbf{L}(z)$ as can be expressed as follows.

$$\mathbf{G}(z) = \hat{\mathbf{G}}(z)\mathbf{L}(z) \tag{a.38}$$

At this point, we will treat the case where $\hat{\mathbf{G}}(z)$ is fixed and consider characterization of circumstances under which stabilizing feedback controller designed for $\hat{\mathbf{G}}(z)$ could retain stability also for $\mathbf{G}(z)$. $\hat{\mathbf{G}}(z)$ can be characterized by a state variable description of the form,

$$\xi_{k+1} = \Phi\xi_k + \Gamma\mathbf{e}_k \tag{a.39}$$

$$\mathbf{y}_k = \mathbf{H}\xi_k \tag{a.40}$$

where Φ, Γ, H represent the system, input and output matrices respectively. $\mathbf{e}_k$ is the error signal, while ξ_k is the internal state vector of the controller-plant cascade whose coordinate basis can be arbitrary. We can determine the nominal open loop characteristic equation as follows,

$$\phi_{ol} = \det[z\mathbf{I} - \Phi] \tag{a.41}$$

The equation's zeros are the poles of the nominal open loop transfer function $\hat{\mathbf{G}}(z)$. For closed loop configuration, we have

$$\mathbf{e}_k = \mathbf{r}_k - \mathbf{y}_k = \mathbf{r}_k - \mathbf{H}\xi_k \tag{a.42}$$

Replace (a.42) in the state variable description (a.40) , we then obtain

$$\xi_{k+1} = (\Phi - \Gamma H)\xi_k + \Gamma \mathbf{r}_k \tag{a.43}$$

Thus, the characteristic equation of the closed loop system is

$$\hat{\phi}_{cl} = \det[z\mathbf{I} - (\Phi - \Gamma H)] \tag{a.44}$$

The zeros of (a.44) are also the nominal closed loop poles that will play important role in determining stability of a feedback system. For obtaining the transfer function of a unity feedback system, we develop the followings.

$$\mathbf{y}(z) = \mathbf{G}(z)\mathbf{u}(z) \tag{a.45}$$

$$\mathbf{u}(z) = \mathbf{r}(z) - \mathbf{y}(z) \tag{a.46}$$

Replace for $\mathbf{u}(z)$ in (a.45) with (a.46), we get

$$\left(\mathbf{I} + \hat{\mathbf{G}}(z)\right)\mathbf{y}(z) = \hat{\mathbf{G}}(z)\mathbf{r}(z) \tag{a.47}$$

$$\mathbf{y}(z) = \left[\mathbf{I} + \hat{\mathbf{G}}(z)\right]^{-1}\hat{\mathbf{G}}(z)\mathbf{r}(z) \tag{a.48}$$

hence, $\left[\mathbf{I} + \hat{\mathbf{G}}(z)\right]^{-1}\hat{\mathbf{G}}(z)$ is the transfer function of the nominal unity feedback system. The matrix $\left[\mathbf{I} + \hat{\mathbf{G}}(z)\right]$ is called the return difference of a feedback system and will be a crucial quantity in defining the robustness margin. We also see that

$$\det\left[\mathbf{I} + \hat{\mathbf{G}}(z)\right] = \frac{\hat{\phi}_{cl}(z)}{\hat{\phi}_{ol}(z)} = \frac{\det[z\mathbf{I} - (\Phi - \Gamma H)]}{\det[z\mathbf{I} - \Phi]} \tag{a.49}$$

We now proceed to present the stability theorem of a unity feedback system by utilizing the Nyquist stability concepts. Asymptotic stability of a closed loop system could be drawn by examining some encirclement characteristics of the open loop transfer function evaluated along the Nyquist contour. As we consider things in discrete-time fashion, the Nyquist contour will correspond to the unit circle and we will denote it by Ω. The theorem for the nominal system comes as follows.

Theorem A.3 (7)

Consider a unity feedback system in Figure A.1, and with the notions above, the nominal closed loop system will be asymptotically stable iff the number of counter-clockwise encirclements of zeros by $\left|\mathbf{I}+\hat{\mathbf{G}}(z)\right|$, *as* z *traverses the contour* Ω *in a counter-clockwise sense, equals the number of zeros of* $\hat{\phi}_{ol}(z)$ *outside or on the unit circle.*

The proof of this fundamental theorem is generally known, hence will not be repeated here. It is clear that the above result does not apply only to stability of the nominal system. If we replace $\left|\mathbf{I}+\hat{\mathbf{G}}(z)\right|$ and $\hat{\phi}_{ol}(z)$ by $\left|\mathbf{I}+\mathbf{G}(z)\right|$ and $\phi_{ol}(z)$, the corresponding true close loop system will also be asymptotically stable. But this is not the case that the true stability is determined by directly examining the true transfer function of a plant system being considered which is never exactly available, at least at some frequencies, in pragmatic situation. What we often have got at hands is that of the nominal system and we want to clarify under what particular condition that the stability in the Nyquist sense concluded from $\left|\mathbf{I}+\hat{\mathbf{G}}(z)\right|$ and $\hat{\phi}_{ol}(z)$ can still be hold with $\left|\mathbf{I}+\mathbf{G}(z)\right|$ and $\phi_{ol}(z)$. This leads to the following theorem.

Theorem A.4 (7)

Consider the actual closed loop system associated with the nominal closed loop system considered in Theorem A.3, let its open loop transfer function denoted by $\mathbf{G}(z) = \hat{\mathbf{G}}(z)\mathbf{L}(z)$, *where* $\mathbf{L}(z)$ *is the multiplicative perturbation to that nominal system. And let denote the actual open loop and closed loop characteristic equations by* $\phi_{ol}(z)$ *and* $\phi_{cl}(z)$ *respectively. Then the actual closed loop system will be asymptotically stable. Provided that*

1. *The nominal closed loop system is asymptotically stable.*
2. $\phi_{ol}(z)$ *and* $\hat{\phi}_{ol}(z)$ *have the same number of zeros outside the unit circle.*
3. $\phi_{ol}(z)$ *and* $\hat{\phi}_{ol}(z)$ *have the same number of zeros on the unit circle.*
4. $\left|\mathbf{I} + (1-\varepsilon)\hat{\mathbf{G}}(z) + \varepsilon\mathbf{G}(z)\right| \neq 0$ *for all* $z \in \Omega$ *and for all* $\varepsilon \in [0,1]$.

Proof:

The central theme of the statement above is that as we consider the deforming path from the Nyquist diagram of $\hat{\mathbf{G}}(z)$ into that of $\mathbf{G}(z)$, the path must not deform passing through zero or infinity in order not to differ the number of encirclement of the origin by $\left|\mathbf{I} + \mathbf{G}(z)\right|$ from that of $\left|\mathbf{I} + \hat{\mathbf{G}}(z)\right|$ as z traverses anticlockwisely the unit circle. The fourth condition prevents the path passing through zero, while the third condition prevents passage through infinity. Hence, the introduction of these two conditions force $\left|\mathbf{I} + \mathbf{G}(z)\right|$ to possess the same number of the encirclement as occurs with $\left|\mathbf{I} + \hat{\mathbf{G}}(z)\right|$. The first condition is straightforward that it makes the number of the encirclement of $\left|\mathbf{I} + \hat{\mathbf{G}}(z)\right|$ equals the number of unstable roots of $\hat{\phi}_{ol}(z)$. Thereafter the second condition completes the implication that the actual closed loop system is also asymptotically stable. Thus, the theorem above is proved.

The establishment of the fourth condition in the above theorem provides a family of the Nyquist diagrams which represent a stabilizing model set neighboring around the nominal system. The stabilizing model set represents the possible values of the actual plant can be, as a model

perturbed away from that of the nominal system, while still possess asymptotic stability. This theorem at first time rigorously addresses our discussion into a demonstrable expression of the robustness, but the appearance of the robustness margin is somewhat implicit. To obtain the expression in a more comprehensive manner, we need to perform some matrix manipulation. According to (8), the following two lemmata are required.

Lemma A.1 (8)

For a constant complex-value matrix $\mathbf{A}$, *we define the maximum and minimum singular value of* $\mathbf{A}$ *by*

$$\overline{\sigma}(\mathbf{A}) = \sqrt{\lambda_{\max}\left(\mathbf{A}^H\mathbf{A}\right)}, \tag{a.50}$$

$$\underline{\sigma}(\mathbf{A}) = \sqrt{\lambda_{\min}\left(\mathbf{A}^H\mathbf{A}\right)}, \tag{a.51}$$

where $\lambda_{\max}(\bullet)$ *and* $\lambda_{\min}(\bullet)$ *are the maximum and the minimum eigenvalues of* $(\bullet)$ *respectively.* $(\bullet)^H$ *denotes the complex conjugate transpose of* $(\bullet)$. *Consider constant square matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *which are both nonsingular, Then, we will have*

$$|\mathbf{I} + \mathbf{AB}| \neq 0,$$

if

$$\overline{\sigma}\left(\mathbf{B}^{-1} - \mathbf{I}\right) < \underline{\sigma}(\mathbf{I} + \mathbf{A}) \tag{a.52}$$

Proof:

We notice that $\overline{\sigma}(\bullet)$ is the Euclidian norm of a matrix denoted by $\|\bullet\|_2$, and we also recall an important property of the Euclidian norm that

$$\underline{\sigma}(\mathbf{A}) = \frac{1}{\overline{\sigma}(\mathbf{A}^{-1})} \tag{a.53}$$

where $\mathbf{A}$ is a constant and nonsingular matrix. Consider (a.52), and using (a.53), we then have

$$\overline{\sigma}(\mathbf{B}^{-1} - \mathbf{I}) < \frac{1}{\overline{\sigma}([\mathbf{I} + \mathbf{A}]^{-1})} \tag{a.54}$$

$$\overline{\sigma}(\mathbf{B}^{-1} - \mathbf{I})\overline{\sigma}([\mathbf{I} + \mathbf{A}]^{-1}) < 1 \tag{a.55}$$

Since, the left side term of (a.55) is the product of the norms, by appealing to the Cauchy-Schwarz inequality, then we can obtain

$$\overline{\sigma}([\mathbf{B}^{-1} - \mathbf{I}][\mathbf{I} + \mathbf{A}]^{-1}) < 1 \tag{a.56}$$

or

$$\|(\mathbf{B}^{-1} - \mathbf{I})(\mathbf{I} + \mathbf{A})^{-1}\|_2 < 1 \tag{a.57}$$

Consider the term $(\mathbf{I} + \mathbf{AB})$, we can rewrite it as

$$\mathbf{I} + \mathbf{AB} = [(\mathbf{B}^{-1} - \mathbf{I})(\mathbf{I} + \mathbf{A})^{-1} + \mathbf{I}](\mathbf{I} + \mathbf{A})\mathbf{B} \tag{a.58}$$

The result (a.57) implies that $(\mathbf{B}^{-1} - \mathbf{I})(\mathbf{I} + \mathbf{A})^{-1}$ is nonsingular, hence the term in the bracket is also nonsingular. The nonsingularity of $(\mathbf{I} + \mathbf{A})$ and $\mathbf{B}$ are presumed from the conditions of the lemma. Thus, we now reach the conclusion that $(\mathbf{I} + \mathbf{AB})$ is nonsingular, i.e.,

$$|\mathbf{I}+\mathbf{AB}| \neq 0 \tag{a.59}$$

The lemma is then proved.

Lemma A.2 (8)

If $\mathbf{A}$ *is a nonsingular matrix and* $\mathbf{P}(\varepsilon)$ *is defined as*

$$\mathbf{P}(\varepsilon) = (1-\varepsilon)\mathbf{I} + \varepsilon\mathbf{A} \tag{a.60}$$

for $\varepsilon \in [0,1]$, *then*

$$\overline{\sigma}\left(\mathbf{A}^{-1} - \mathbf{I}\right) < \alpha \leq 1 \tag{a.61}$$

implies

$$\overline{\sigma}\left(\mathbf{P}(\varepsilon)^{-1} - \mathbf{I}\right) < \alpha \tag{a.62}$$

Proof:

From matrix theory, $\overline{\sigma}\left(\mathbf{P}(\varepsilon)^{-1} - \mathbf{I}\right) < \alpha$ is an equivalence of the following inequality,

$$\alpha^2 \mathbf{P}(\varepsilon)^H \mathbf{P}(\varepsilon) - \left(\mathbf{P}(\varepsilon) - \mathbf{I}\right)^H \left(\mathbf{P}(\varepsilon) - \mathbf{I}\right) > 0 \tag{a.63}$$

Use (a.60) to replace $\mathbf{P}(\varepsilon)$ in (a.63), then (a.63) comes as

$$\varepsilon^2\left[\alpha^2\mathbf{A}^H\mathbf{A}-(\mathbf{A}-\mathbf{I})^H(\mathbf{A}-\mathbf{I})\right]+\alpha^2(1-\varepsilon)\left[(1-\varepsilon)\mathbf{I}+\varepsilon\left(\mathbf{A}+\mathbf{A}^H\right)\right]>0 \quad \text{(a.64)}$$

In the first left hand side term, the term in the bracket is the equivalence of $\overline{\sigma}\left(\mathbf{A}^{-1}-\mathbf{I}\right)<\alpha$, hence, it is positive definite as in (a.64). Further with $\alpha \le 1$, this condition also yields

$$\mathbf{A}+\mathbf{A}^H>\mathbf{I}+\left(1-\alpha^2\right)\mathbf{A}^H\mathbf{A}>0 \quad \text{(a.65)}$$

Thus, the second term of the left hand side of (a.65) is also positive definite. Now we can complete the proof that (a.61) indeed implies (a.62).

As a result of applying the two lemmata above to the fourth condition of Theorem A.4, the following theorem comes out directly.

Theorem A.5 (7)

Consider the actual and nominal plant system whose associated notions are as in Theorem A.4 . Then, the actual closed loop system will be asymptotically stable, i.e, $\phi_{cl}(z)$ has no zeros outside or on the unit circle. Provided,

1. *The nominal closed loop systems is asymptotically stable.*
2. *$\phi_{ol}(z)$ and $\hat{\phi}_{ol}(z)$ have the same number of zeros outside the unit circle.*
3. *$\phi_{ol}(z)$ and $\hat{\phi}_{ol}(z)$ have the same number of zeros on the unit circle.*
4. *$\overline{\sigma}\left(\mathbf{L}^{-1}(z)-\mathbf{I}\right)<\min(\alpha(z),1)$ at each $z\in\Omega$, where*

$$\alpha(z)=\underline{\sigma}\left(\mathbf{I}+\hat{\mathbf{G}}(\mathbf{z})\right) \quad \text{(a.66)}$$

Proof:

The theorem is Theorem A.4 with the fourth assumption modified by the application of Lemma A.1 and A.2. Applying Lemma A.1, we take $\mathbf{A} = \hat{\mathbf{G}}(z)$ and $\mathbf{B} = (1-\varepsilon)\mathbf{I} + \varepsilon\mathbf{L}$. We have

$$\bar{\sigma}\left(\left[(1-\varepsilon)\mathbf{I} + \varepsilon\mathbf{L}\right]^{-1} - \mathbf{I}\right) < \underline{\sigma}\left(\mathbf{I} + \hat{\mathbf{G}}(z)\right). \tag{a.67}$$

Applying Lemma A.2, we take $\mathbf{A} = \mathbf{L}$ and $\alpha(z) = \underline{\sigma}\left(\mathbf{I} + \hat{\mathbf{G}}(z)\right)$, then

$$\bar{\sigma}\left(\mathbf{L}^{-1} - \mathbf{I}\right) < \underline{\sigma}\left(\mathbf{I} + \hat{\mathbf{G}}(z)\right) \le 1. \tag{a.68}$$

Thus, the proof is completed.

We observe that the term $\bar{\sigma}\left(\mathbf{L}^{-1} - \mathbf{I}\right)$ is upper bounded by a designed value, . $\bar{\sigma}\left(\mathbf{L}^{-1} - \mathbf{I}\right)$ is the Euclidian norm that measures the size of $\left(\mathbf{L}^{-1} - \mathbf{I}\right)$ which arises from the mismatch in modeling process. Indeed $\left(\mathbf{L}^{-1} - \mathbf{I}\right)$ represents the percentage error of the frequency response of the plant/controller combination as can be shown by the following expression,

$$\mathbf{L}^{-1} - \mathbf{I} = \mathbf{G}(z)^{-1}\left[\hat{\mathbf{G}}(z) - \mathbf{G}(z)\right] \tag{a.69}$$

While $\underline{\sigma}\left(\mathbf{I} + \hat{\mathbf{G}}(z)\right)$ represents a quantity that comes from control design process. The theorem, therefor, directly demonstrates the robustness of a unit feedback system to multiplicative perturbations provided the percentage of plant/model mismatch is suitably small as implicitly limited

by a quantity depending upon the controller's frequency characteristics. As a result, the control design should be penalize to provide the minimum singular value of the return difference being sufficiently large so that a large amount of modeling error can be allowed while asymptotic stability is also maintained. It is to guaranteed that in practical situation the inaccuracy of a model used in the control design will not lead to instability of the achieved system.

Having established the appropriate assumption to obtain guaranteed stability margin for a unity feedback system, we now shall proceed to adopt this result with our regulator in the framework of the infinite horizon LQR. Since from the asymptotic stability proof in Section 3.3.2, we have already shown that the stability property of a one-step-ahead horizon LQR can be equivalently studied as that of an infinite horizon LQR with the weighting $\overline{\mathbf{Q}}$ that is larger than $\mathbf{Q}$ of the one-step-ahead problem, but the same weighting $\mathbf{R}$. Hence, from now on the robustness consideration of our one-step-ahead regulator will be carried out as the implication of the infinite horizon results. For an infinite horizon LQR with full state feedbacks, it can be shown that stability robustness is always guaranteed by the presence of the lower bound of the smallest singular value of the return difference matrix. This means that the stability margin permitting some plant modeling error will never disappear as it will be strictly positive everywhere on the unit circle when full state feedbacks in the LQ sense is applied. The guaranteed stability margin of the LQR can be obtained by virtue of an important algebraic equation, called the *Return Difference Equality*. The equality can be considered as the fundamental frequency domain characterization of the basic Algebraic Riccati Equation (ARE) of the infinite horizon LQ optimum control, since many of the robustness results stem from it. We now present in details its derivation as follows.

We commence with the ARE (a.12),

$$\mathbf{P} = (\mathbf{F} + \mathbf{GK})^T \mathbf{P}(\mathbf{F} + \mathbf{GK}) + \mathbf{Q} + \mathbf{K}^T\mathbf{RK}$$

where

$$\mathbf{K} = -\left(\mathbf{G}^T\mathbf{P}\mathbf{G} + \mathbf{R}\right)^{-1}\mathbf{G}^T\mathbf{P}\mathbf{F}$$

It can be rewritten in another form as below.

$$\mathbf{P} = \mathbf{F}^T\mathbf{P}\mathbf{F} - \mathbf{K}^T\left(\mathbf{G}^T\mathbf{P}\mathbf{G} + \mathbf{R}\right)\mathbf{K} + \mathbf{Q} \tag{a.70}$$

Rewrite the above in terms of the following,

$$\mathbf{S} = \mathbf{G}^T\mathbf{P}\mathbf{G} + \mathbf{R} \tag{a.71}$$

We obtain

$$\mathbf{P} = \mathbf{F}^T\mathbf{P}\mathbf{F} - \mathbf{K}^T\mathbf{S}\mathbf{K} + \mathbf{Q} \tag{a.72}$$

and

$$\mathbf{Q} - \mathbf{K}^T\mathbf{S}\mathbf{K} = \mathbf{P} - \mathbf{F}^T\mathbf{P}\mathbf{F} \tag{a.73}$$

From the right hand side of the above, the following identity may be established.

$$\mathbf{P} - \mathbf{F}^T\mathbf{P}\mathbf{F} = \left(z^{-1}\mathbf{I} - \mathbf{F}\right)^T\mathbf{P}(z\mathbf{I} - \mathbf{F}) + \left(z^{-1}\mathbf{I} - \mathbf{F}\right)^T\mathbf{P}\mathbf{F} + \mathbf{F}^T\mathbf{P}(z\mathbf{I} - \mathbf{F}) \tag{a.74}$$

Thus,

$$\mathbf{Q} - \mathbf{K}^T\mathbf{S}\mathbf{K} = \left(z^{-1}\mathbf{I} - \mathbf{F}\right)^T\mathbf{P}(z\mathbf{I} - \mathbf{F}) + \left(z^{-1}\mathbf{I} - \mathbf{F}\right)^T\mathbf{P}\mathbf{F} + \mathbf{F}^T\mathbf{P}(z\mathbf{I} - \mathbf{F})$$

(a.75)

Next, we premultiply and postmultiply the above by $\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}$ and $(z\mathbf{I}-\mathbf{F})^{-1}$ respectively, yielding

$$\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{Q}(z\mathbf{I}-\mathbf{F})^{-1}=\mathbf{P}+\mathbf{PF}(z\mathbf{I}-\mathbf{F})^{-1}+\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{F}^{T}\mathbf{P}$$
$$+\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{K}^{T}\mathbf{SK}(z\mathbf{I}-\mathbf{F})^{-1}$$

(a.76)

Further, the above is premultiplied by $\mathbf{G}^{T}$ and postmultiplied by $\mathbf{G}$, thereafter $\mathbf{R}$ is added to both side of it.

$$\mathbf{R}+\mathbf{G}^{T}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{Q}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}=\left(\mathbf{G}^{T}\mathbf{PG}+\mathbf{R}\right)+\mathbf{G}^{T}\mathbf{PF}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}$$
$$+\mathbf{G}^{T}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{F}^{T}\mathbf{PG}$$
$$+\mathbf{G}^{T}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{K}^{T}\mathbf{SK}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}$$

(a.77)

Rewrite (a.77) in terms of $\mathbf{S}$, that is

$$\mathbf{R}+\mathbf{G}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{Q}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}=\mathbf{S}-\mathbf{SK}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}-\mathbf{G}^{T}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{K}^{T}\mathbf{S}$$
$$+\mathbf{G}^{T}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{K}^{T}\mathbf{SK}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}$$

$$=\left[\mathbf{I}-\mathbf{K}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-1}\mathbf{G}\right]^{T}\mathbf{S}\left[\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right]$$

(a.78)

Thus, we finally obtain the return difference equality result as follows.

$$\mathbf{R}+\mathbf{G}^T\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-T}\mathbf{Q}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}=\left[\mathbf{I}-\mathbf{K}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-1}\mathbf{G}\right]^T\left(\mathbf{G}^T\mathbf{P}\mathbf{G}+\mathbf{R}\right)\left[\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right] \quad \text{(a.79)}$$

From the result above, we note that for full state feedback, the transfer function of the controller/plant combination is given by

$$\hat{\mathbf{G}}(z)=-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G} \quad \text{(a.80)}$$

And the return difference matrix is

$$\mathbf{I}+\hat{\mathbf{G}}(z)=\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G} \quad \text{(a.81)}$$

The results lead us directly to the establishment of the following theorem.

Theorem A.6 (7)

Consider the infinite horizon LQR with full state feedback as above, for which we assume the following condition:

- $[\mathbf{F},\mathbf{G}]$ *is stabilizable.*
- $[\mathbf{F},\mathbf{Q}]$ *is detectable.*
- $\mathbf{R}$ *has full rank.*

Then, one has the following bound on the minimum singular value of the return difference matrix. That is there exists $0<\overline{\alpha}\le 1$ *such that*

$$\underline{\sigma}\left(\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right)\geq\overline{\alpha}\text{, for all } z\in\Omega \tag{a.82}$$

Proof:

The first and the second conditions are provided to support the existence of the ARE, and hence the return difference equality. Consider (a.79), we observe that the second term on the left hand side of it is always positive for z on the unit circle. Hence , we get

$$\left[\mathbf{I}-\mathbf{K}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-1}\mathbf{G}\right]^{T}\left(\mathbf{G}^{T}\mathbf{P}\mathbf{G}+\mathbf{R}\right)\left[\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right]>\mathbf{R} \tag{a.83}$$

According to the result in (9), the finite maximal and positive definite solution, $\mathbf{P}$, of the ARE exists. Therefore, $\left(\mathbf{G}^{T}\mathbf{P}\mathbf{G}+\mathbf{R}\right)$ is positively finite. And then it follows that there exists $\beta>0$, such that

$$\mathbf{G}^{T}\mathbf{P}\mathbf{G}+\mathbf{R}<\beta\mathbf{I} \tag{a.84}$$

Replacing the result in (a.83), we then obtain

$$\left[\mathbf{I}-\mathbf{K}\left(z^{-1}\mathbf{I}-\mathbf{F}\right)^{-1}\mathbf{G}\right]^{T}\left[\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right]>\frac{1}{\beta}\bullet\mathbf{R} \tag{a.85}$$

which implies that

$$\underline{\sigma}\left(\mathbf{I}-\mathbf{K}(z\mathbf{I}-\mathbf{F})^{-1}\mathbf{G}\right)\geq\frac{\underline{\sigma}(\mathbf{R})}{\beta}\text{.} \tag{a.86}$$

Thus, $\overline{\alpha}$ exists and at least equals $\frac{\underline{\sigma}(\mathbf{R})}{\beta}$. $\overline{\alpha}\leq 1$ can be noted from the result (a.84). And $\mathbf{R}$ has to be full rank to yield $\underline{\sigma}(\mathbf{R})\neq 0$ and hence, $\overline{\alpha}>0$.

The presentation of Theorem A.6 is to show that if an infinite horizon LQ controller is designed on the basis of a nominal plant, then the corresponding nominal return difference, $\mathbf{I}+\hat{\mathbf{G}}(z)$, will have its smallest singular value lower bounded by $\overline{\alpha}$. Hence, this LQ controller will be capable of stabilizing a neighborhood of the nominal plant with a robustness margin at least as large as $\overline{\alpha}$.

A.5 The Recursive Least Square Identification

In a control law utilizing the certainty equivalence principle, there is a need to retrieve plant model information to base the on-line control design incorporated within the control loop. This can be achieved by performing recursive identification module in which the plant model parameters can be extracted from input ad measured output of the real plant. The formulation of an identification process bases on a particular form of a plant model that is called the linear parameterized model. A physical model can be parametrically characterized so as to express measured output in terms of a parameter vector.

Consider a plant system whose transfer function is

$$\mathbf{y}_k = \hat{\mathbf{P}}(z)\mathbf{u}_k \qquad \text{(a.87)}$$

where $\mathbf{y}_k$ is the measurable out put of the plant, $\mathbf{u}_k$ is the vector of the input signals of the plant. The problem here is that we want to identify the model $\hat{\mathbf{P}}(z)$, when values of $\mathbf{y}_k$ and $\mathbf{u}_k$. are given. The model $\hat{\mathbf{G}}(z)$ can be parameterized by a selected parameter vector whose value is expressed in terms of some physical parameters. In other word, we may in very many case express $\mathbf{y}_k$ as follows.

$$\mathbf{y}_k = \phi_k^T(\theta)\theta + \mathbf{e}_k(\theta), \qquad \text{(a.88)}$$

where θ is the selected parameter vector, $\phi_k(\theta)$ is a matrix of pseudoregressors at time k, and $\mathbf{e}_k(\theta)$ represents the error to correlate the prediction output using a parameterized model with the measured output. If the parameterized model above is available, we can then predict the output $\mathbf{y}_k$ by

$$\hat{\mathbf{y}}_k = \phi_k^T(\theta)\theta, \tag{a.89}$$

where $\hat{\mathbf{y}}_k$ is the predicted value of $\mathbf{y}_k$. The prediction error is then

$$\mathbf{e}_k(\theta) = \mathbf{y}_k - \hat{\mathbf{y}}_k \tag{a.90}$$

To have the best estimate of the model, we should impose a criterion that exhibits minimum of the prediction error accumulated on a number of measurement data. In the least square methodology, the criterion is to seek for a parameter vector θ that will minimize

$$V_N(\theta,\eta) = \frac{1}{N}\sum_{k=1}^{N}\left[\mathbf{e}_k^f(\theta,\eta)\right]^2, \tag{a.91}$$

where N stands for the number of data measurements used in the process. $\mathbf{e}_k^f(\theta,\eta)$ denotes the prediction error filtered through a stable linear filter with transfer function, $\mathbf{D}(z,\eta)$, i.e.,

$$\mathbf{e}_k^f(\theta,\eta) = \mathbf{D}(z,\eta)\mathbf{e}_k(\theta). \tag{a.92}$$

The parameter vector η indicates another set of parameters on which the filter may depend. This filter is conventionally inserted to predistort the input signals to an identifier in order for achieving some objectives. The determination of the parameter estimate can be implicitly posed as

$$\theta_N = \arg\min_{\theta\in D_\theta} V_N(\theta,\eta) \tag{a.93}$$

where $\hat{\theta}_N$ is the optimizing parameter estimate according to N data measurements. D_θ represents the model set characterized by a number of parameter vectors. The solution to the above problem is available in explicit form only when the parameterized model is linear in the parameters, i.e., ϕ_k is not a function of θ. In most case, the solution of (a.93) can not be stated explicitly as a closed form expression. One of many existing iterative algorithms must be used to compute the solution.

A.6 Convergence Point of A RLS Identifier in Closed Loop with A LQ Regulator

When a RLS identifier is constructed to be used within a closed loop system, a question naturally arises is that whether or not the operation of the identifier will affect the controller performance and in converse, there should be some effects of the controller that can disturb quality of on-line estimation. If we reconsider the misinformation problem as posed in (3.120), but this time, we let the identification occurred within a closed loop system, it is obvious to us that the regulator and the prefilter themselves are also functions of the model parameters. As can be posed by the following optimization,

$$\theta^* = \arg\min_{\theta \in D_\theta} \overline{V}(\theta, \eta(\theta), \rho(\theta)), \qquad \text{(a.94)}$$

where $\rho(\theta)$ is a parameter vector that characterizes the regulator. This represents a highly nonlinear misinformation problem whose solution is not yet guaranteed to exist. And if the solution exists, we do not know whether it is unique or not. Furthermore, the existence of its unique solution does not guarantee that it will yield a stabilizing closed loop system. This point of view will be central to the discussion in this section. The case needed to be clarified is that we want to examine on what conditions that small perturbation on the identified model parameters caused from the LQ regulator performance will not make the resulting model swaying out of a stabilizing model set. For the term "stabilizing model set", we refer to a set of parameter vectors, obtained from the least square misinformation, characterizing parameterized models that can yield stable regulators. Based

on the recent results from (7), it can be proved that if the parameter update algorithm is initialized with a value $\hat{\theta}_0$ that is close enough to the optimal value (a.93).and if the parameter update is carried out slowly with respect to the closed loop dynamics, then the trajectory of the parameter estimation algorithm will remain within a stabilizing model set. And further, the estimate will converge to a neighborhood of a point being close to the optimal solution θ^*.

In the discussion to determine a potential convergence point of the recursive estimation, we shall be concerned with the case that the true plant system is not within the model set D_θ for whatever values of the parameters chosen. This is almost always the case in practice, because the true system is of much higher complexity than the chosen model structure. When the true system, however, is within the model set and when all necessary assumptions for the estimation algorithm to converge can be met, the identified model will, indeed, converge to the true system , and therefore, the regulator that is designed preferably on the basis of the true system by the methodology as discussed in the previous section will certainly have asymptotic stability. But in facing of plant /model mismatch, unless some special conditions are imposed, there is no obvious reason why a model that minimizes a prediction error criterion should also minimize a control performance criterion and thus stabilize a controlled plant. Despite, the robustness results can be appealed to further hold stability provided that the plant/model mismatch satisfies the robustness criterion, there has seen no direct way yet to connect such small mismatch with the misinformation of the prediction error criterion. Refer to the essential robustness condition as discussed before,

$$\overline{\sigma}\left(\mathbf{L}^{-1}-\mathbf{I}\right)<\underline{\sigma}\left(\mathbf{I}+\hat{\mathbf{G}}\right), \qquad \text{(a.95)}$$

we know that the LQ design methodology guarantees that the right side of the above always exists as a positive value that is lower bounded by a computable quantity, hence always leaving some room for plant/model mismatch to occur, whose allowable size is termed by the left side of (a.95). But we see that it is not clear that the identified model from the prediction error misinformation will make a

quantity, $\overline{\sigma}\left(\mathbf{L}^{-1}-\mathbf{I}\right)$, suitably small with the limitation above. According to this point of view, if we can elaborate some special precautions imposed on the identification, it may possible to define an identified model set that would yield small relative error enough, $\overline{\sigma}\left(\mathbf{L}^{-1}-\mathbf{I}\right)$, to satisfy the robustness inequality above. Hence the overall stability of the control system can still be guarantee.

Consider the problem of designing a LQ regulator and suppose that we have a model set D_θ that covers all physically possible values of a parameter vector based upon the model structure we have chosen, for any model θ in our model set the LQ solution leads to a particular regulator that can be parameterized by a vector $\rho(\theta)$. If we adopt this regulator to work with the actual plant, we then get an LQ optimal cost $J(\theta)$. This optimal is the achieved optimal cost obtained from actual measurements of all inputs and outputs, while the actual closed loop is performing. We then define the best model within our model set D_θ as the model that yields a regulator that will minimize the achieved cost $J(\theta)$,as can be posed by the following expression.

$$\theta^{**} = \arg\min_{\theta \in D_\theta} J(\theta) \tag{a.96}$$

This can be possible by referring to the LQ design method and utilizing additional appropriate conditions. However, we should note that θ^{**} need not be obtainable as a solution the prediction error criterion misinformation, and by now, there is no trend to bridge the gap between the two optimal criterion. Our proceeding from now on is to show that one can define a model θ^{*} obtained via the misinformation of the prediction error criterion, that is close enough to the θ^{**} model to be also stabilizing, provided that the amount of unmodeled dynamics representing the plant/model mismatch is not too large and bounded by an expressible constraint and provided, the domain D_θ is suitably restricted. Before we proceed to propose such assumptions, we shall introduce some relevant notations. First, we will write the true system description as

$$\mathrm{y}_k = \phi_k^T\theta^{**} + \zeta_k + \upsilon_k \tag{a.97}$$

This can be thought of as introducing the definition of the unmodeled dynamics ζ_k, where ϕ_k is a regressor of past inputs and outputs, and θ^{**} defined as (a.96). υ_k is a zero mean white noise process added to facilitate stochastic analysis. By definition, the description is considered to be strictly proper, and the LQ design based upon the model characterized by θ^{**} will presumedly result in an asymptotically stable regulator. We shall discuss the situation when the true plant is operated in a feedback loop with a stabilizing optimal regulator, but designed with a model θ resulted from performing an identifier also incorporated within that closed loop. The model used in the design can be written in terms of the measured output as follows.

$$\mathbf{y}_k = \phi_k^T(\theta)\theta + \zeta_k(\theta) \tag{a.98}$$

Note that ϕ_k and ζ_k are functions of θ, since ϕ_k is preferred to be a pseudo regressor to cover the plant model that is not linear in the parameters, and it is obvious that a particular value of θ will implicitly determine the amount of unmodeled dynamics. We shall then assume that υ_k is uncorrelated with both ϕ_k and ζ_k. It is now appropriate to state the essential assumption.

Assumption A.1 (7)

There exists a closed hypersphere

$$B_r(\theta^{**}) = \{\theta : |\theta - \theta^{**}| \le r\}$$

centered on θ^{**} *with radius* r *such that*

1. *for all* $\theta \in B_r(\theta^{**})$ *the closed loop system is stable,*
2. *There exists positive constants* α, β, δ, k *such that for all* $\theta \in B_r(\theta^{**})$

$$E\left|\zeta_k(\theta^{**})\right| \le \alpha\left(E\left|\phi_k(\theta^{**})\right| + k\right), \tag{a.99}$$

$$E\left|\frac{\partial \phi_k(\theta)}{\partial \theta}\right| < \beta, \tag{a.100}$$

$$E\left|\frac{\partial \zeta_k(\theta)}{\partial \theta}\right| < \delta, \tag{a.101}$$

with α, β *and* δ *small enough so that*

$$r\lambda > 2\left[\beta m r^2 + \delta m r + \beta\delta r^2 + \alpha(m+k)(m+\beta r+\delta)\right], \tag{a.102}$$

where

$$m = E\left|\phi_k(\theta^{**})\right|, \tag{a.103}$$

$$\lambda = \lambda_{\min} E\left[\phi_k(\theta^{**})\phi_k^T(\theta^{**})\right], \tag{a.104}$$

$\lambda_{\min}$ *denotes the smallest eigenvalue.*

Assumption A.1 provides the existence of a stabilizing model set that we prefer to have in order to bridge the gap between the two optimal criterion: one is of the control and one is of the estimation, through the determination of the closed hypersphere B_r that is centered by the LQ control optimizing model θ^{**} as defined in (a.96). The first condition tells us that around the optimizing model θ^{**} , there exists parameterized models that are also stabilizable, hence B_r is defined to be a stabilizing model set. If we refer to the stability robustness margin of the LQ control design, the first condition is very reasonable as the design always yields some room for model

perturbation, thereby it implies that the radius r of B_r should never be zero. The size of the ball B_r is implicitly defined by the inequality (a.102) in the second condition. The condition is elaborated from several important quantities. (a.99) provides the limit of the amount of unmodeled dynamics with respect to the quantity linearly relating the amount of the regressor. (a.100) and (a.101) assume smoothness of the regressor and unmodeled dynamics in the model parameters as implied from that the size of their first derivatives with respect to θ must be finite. As may be guessed, the level of persistency of excitation via λ affects the size of the model set as evident in (a.102). For the reason that will be clear later in the proof of the following lemma, the inequality (a.102) is essential for the applicability of the LQ regulator and the RLS identifier cooperation. This will be stated in Lemma A.3.

Lemma A.3 (7)

Consider θ^{**} *defined as (a.96) and its associated closed loop hypersphere* $B_r(\theta^{**})$ *satisfying Assumption A.1, then there exists* θ^{*}*,a parameterized model within* $B_r(\theta^{**})$ *which is the solution of the prediction error misinformation problem in the least square sense as the following.*

$$\theta^{*} = \arg\min_{\theta \in B_r(\theta^{**})} \overline{V}(\theta, \rho(\theta), \eta(\theta)). \tag{a.105}$$

Proof:

From the statement above, if Assumption A.1 is satisfied, for any θ to be a solution of the LS prediction error misinformation, it must be interior point of $B_r(\theta^{**})$. Therefore, we shall proof that for any θ on the boundary of $B_r(\theta^{**})$ its corresponding LS criterion cost must be more than that of θ^{**}, i.e., Lemma A.3 is equivalent to

$$\overline{V}(\theta_o) > \overline{V}(\theta^{**}),\ \theta_o \in \{\theta : |\theta - \theta^{**}| = r\}. \tag{a.106}$$

From (a.97), it is clear that when the optimizing model is used , the difference between the real and the predicted output of the plant is

$$\begin{aligned}\mathbf{e}_k(\theta^{**}) &= \mathbf{y}_k - \hat{\mathbf{y}}_k(\theta^{**}), \\ &= \zeta_k(\theta^{**}) + \upsilon_k \end{aligned} \tag{a.107}$$

The LS cost is therefore,

$$\overline{V}(\theta^{**}) = E\left[\mathbf{e}_k(\theta^{**})^2\right] \tag{a.108}$$

Based on the fundamental assumption that ζ_k and υ_k are uncorrelated, the covariance of ζ_k and , hence, diminishes. We then get

$$\overline{V}(\theta^{**}) = E\left[\zeta_k(\theta^{**})^2 + \upsilon_k^2\right]. \tag{a.109}$$

Consider any model θ_o on the boundary of $B_r(\theta^{**})$, its corresponding LS cost is

$$\begin{aligned}\overline{V}(\theta_o) &= E\left[\mathbf{y}_k - \hat{\mathbf{y}}_k(\theta_o)\right]^2, \\ &= E\left[\phi_k^T(\theta_o)\tilde{\theta} + \zeta_k(\theta_o) + \upsilon_k\right]^2, \end{aligned} \tag{a.110}$$

where $\tilde{\theta}$ denotes $\theta_o - \theta^{**}$. Applying Taylor's expansion about an operating point θ^{**} for the expression of ϕ_k and ζ_k at θ_o, we have

$$\phi_k(\theta_o) = \phi(\theta^{**}) + \tilde{\theta}^T \left.\frac{\partial \phi_k}{\partial \theta}\right|_{\theta=\theta_1}. \tag{a.111}$$

$$\zeta_k(\theta_o) = \zeta_k(\theta^{**}) + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta=\theta_2} . \tag{a.112}$$

where θ_1 and θ_2 are two intermediate points in $B_r(\theta^{**})$ between θ_o and θ^{**}. The above represents approximates by linearizing ϕ_k and ζ_k with respect to θ. Substitute (a.111) and (a.112) into (a.110) and distribute the squared term as follows.

$$\overline{V}(\theta_o) = E\left[\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \zeta_k(\theta^{**}) + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} + \upsilon_k \right]^2$$

$$= E\left[\left(\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \right) + \left(\zeta_k(\theta^{**}) + \upsilon_k \right) \right]$$

$$= 2E\left[\zeta_k(\theta^{**}) + \upsilon_k \right] \left(\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \right)$$

$$+ E\left[\zeta_k(\theta^{**}) + \upsilon_k \right]^2 + E\left[\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \right]^2 .$$

(a.113)

At this point we note that the first term on the right side of the above is $\overline{V}(\theta^{**})$, hence for (a.106) to hold, it follows that the two other terms must be positive.

$$2E\left[\zeta_k(\theta^{**}) + \upsilon_k \right] \left(\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \right)$$

$$+ E\left[\phi_k^T(\theta^{**})\tilde{\theta} + \tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta} + \tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \right]^2 > 0 \tag{a.114}$$

The second term of the left side can be distributed as follows.

$$E\left[\phi_k^T(\theta^{**})\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\right]^2=E\Big[\tilde{\theta}^T\phi_k(\theta^{**})\phi_k^T(\theta^{**})\tilde{\theta}$$

$$+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\phi_k^T(\theta^{**})\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\phi_k^T(\theta^{**})\tilde{\theta}+\phi_k^T(\theta^{**})\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}$$

$$+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\phi_k^T(\theta^{**})\tilde{\theta}\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}$$

$$+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\Big] \qquad \text{(a.115)}$$

$$E\left[\phi_k^T(\theta^{**})\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\right]^2=E\Big[\tilde{\theta}^T\phi_k(\theta^{**})\phi_k^T(\theta^{**})\tilde{\theta}$$

$$+2\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\phi_k^T(\theta^{**})\tilde{\theta}+2\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\phi_k^T(\theta^{**})\tilde{\theta}+2\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}$$

$$+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\Big] \qquad \text{(a.116)}$$

Then, (a.114) comes out as

$$2E\Big[\zeta_k(\theta^{**})+\upsilon_k\Big]\left(\phi_k^T(\theta^{**})\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\right)$$

$$+E\Big[\tilde{\theta}^T\phi_k(\theta^{**})\phi_k^T(\theta^{**})\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}\tilde{\theta}^T\left.\frac{\partial\phi_k}{\partial\theta}\right|_{\theta_1}\tilde{\theta}+\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}\tilde{\theta}^T\left.\frac{\partial\zeta_k}{\partial\theta}\right|_{\theta_2}$$

$$+2\tilde{\theta}^T \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta}\phi_k^T(\theta^{**})\tilde{\theta} + 2\tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \phi_k^T(\theta^{**})\tilde{\theta} + 2\tilde{\theta}^T \frac{\partial \zeta_k}{\partial \theta}\bigg|_{\theta_2} \frac{\partial \phi_k}{\partial \theta}\bigg|_{\theta_1} \tilde{\theta}\Bigg] > 0 \tag{a.117}$$

Next, we appeal to the Cauchy-Schwartz inequality that, for any column vector $\mathbf{x}, \mathbf{y} \in \mathbf{R}^n$

$$\mathbf{x}^T\mathbf{y} \le |\mathbf{x}\|\mathbf{y}|, \tag{a.118}$$

which can be rewritten as

$$\mathbf{x}^T\mathbf{y} = \rho|\mathbf{x}\|\mathbf{y}| \ , \rho \text{ is a scalar that } 0 \le |\rho| \le 1. \tag{a.119}$$

Thus, the possible lowest value of $\mathbf{x}^T\mathbf{y}$ is

$$-|\mathbf{x}\|\mathbf{y}| \tag{a.120}$$

As a consequence of implementing the possible lowest value of scalar products of two vectors via the Cauchy-schwartz inequality, and as we adopt the proposed bounds in the second conditions of Assumption A.1 with (a.117), we then obtain

$$2\alpha(m+k)\left(-mr - \beta r^2 - \delta r\right) + \lambda r^2 - \beta^2 r^4 - \delta^2 r^2 - 2\beta m r^{32} - 2\delta m r^2 - 2\delta\beta r^3 > 0 \tag{a.121}$$

where $r = \left|\tilde{\theta}\right|$ and the size of the matrix $\phi_k\phi_k^T$ is referred to as being its associated smallest eigenvalue that is denoted by λ. Recall that β and δ are assumed to be suitably small, their squares may be considered having negligible values, then $\beta^2 r^4$ and $\delta^2 r^2$ vanish from (a.121). We further factor out (a.121) by r, readjust the terms to end up the result with the form,

$$\lambda r > 2\left[\beta m r^2 + \delta m r + \beta\delta r^2 + \alpha(m+k)(m+\beta r+\delta)\right]$$

Since the above is satisfied directly by construction of Assumption A.1 as one of the assumed properties of $B_r(\theta^{**})$, thereby, we can conclude (a.106) and the proof is completed.

Lemma A.3 has pointed that if some appropriate conditions, i.e., variations in parameters of the regressor and unmodeled dynamics are smooth and the amount of unmodeled dynamics is suitably restricted, are satisfied, we can indeed obtain a stabilizing model θ^* that is also the solution of the off-line prediction error misinformation problem and its presence is confined to be a neighborhood of the LQ control optimizing model θ^{**}. Hence, it is clear that Lemma 3.3 implicitly connects the LQ optimum control with the RLS estimation. This result, however, does not involve the on-line estimation case which is to be adopted in an indirect adaptive control scheme. To make the result applicable with the on-line estimation, we need to appeal to the powerful integral manifold theory with an assumption of slow adaptation.

The further task is to demonstrate that with a suitably restricted search domain and a sufficiently slow rate of parameter adaptation, the solution of recursive least square identification implemented in an adaptive closed loop will converge to the neighborhood of stabilizing models around θ^*. In order to achieve total stability result, we have to consider the complete nonlinear dynamic equations of the total adaptive control system which consists of the controlled plant dynamics, the controller dynamics and the identifier dynamics. The total dynamics of the whole control system will be highly complicated and of several difficulties to be directly analyzed. Despite we have known in details behaviors of each separated part. Nevertheless, by appealing to integral manifold theory with the assumption of slow adaptation, we can argue that the whole system dynamics can be split up into fast θ-dependent combined state-space equations, that describe combined dynamics of the whole adaptive system excluding only that of the identifier, and a parameter update equation whose operation is assumed to be carried out slowly with relative to the

combined dynamics. The assumption of slow parameter update can be satisfied by using a suitably small adaptation gain in the recursive algorithm. With this time scale separation of dynamics, it can be shown that the averaged behavior of the recursive algorithm can be described by an ordinary differential equation whose solution is close to the solution θ^* derived from the off-line identification problem, provided unmodeled dynamics is suitably restricted and the excitation to the identifier is persistent. Under appropriate conditions similar to those of Assumption 3.1, the solution of the recursive identification algorithm can then be shown by the integral manifold arguments to converge to a limiting solution which itself is close to θ_o with an additional condition that the initial parameter value to start the recursive algorithm is close to θ_o.

we commence with a combined set of dynamic equations of the whole adaptive control system. We prefer that they can be represented by the following equations.

$$\Xi_{k+1} = \mathbf{A}(\theta_k)\Xi_k + \mathbf{B}(\theta_k)\mathbf{n}_k \quad ,\Xi \in \mathbf{R}^s \tag{a.122}$$

$$\theta_{k+1} = \theta_k + \gamma f_k(\theta_k, \Xi_k) \quad ,\theta \in \mathbf{R}^d \tag{a.123}$$

where Ξ is the combined state including the state of the true plant, of the plant model, of the regulator, and of any other dynamics that may be considered within the total closed loop. $\mathbf{n}_k$ denotes a vector made up of all external signals. (a.123) is a parameter update equation obtained from the recursive least square algorithm. γ is a constant adaptation gain which is assumed to be small. $f_k(\theta_k, \Xi_k)$ is a time-varying nonlinear function of both θ and Ξ.

Assumption A.2 (7)

1. There exists a compact set Θ containing θ^* and constants $\lambda \in (0,1)$ and $K_1 \geq 1$ such that $\forall \theta \in \Theta$ and $\forall t \geq 0$

$$\left|A(\theta)^t\right| \le K_1 \lambda^t \tag{a.124}$$

2. There exists constants c , c_1 *and* c_2 *such that the frozen parameter response*

$$\upsilon_t(\theta) = \sum_{j=0}^{\infty} A^j(\theta) B(\theta) n_{t-j-1} \tag{a.125}$$

and its sensitivity $\frac{\partial \upsilon_t}{\partial \theta}(\theta)$ *satisfy*

$$\left|\upsilon_t(\theta)\right| \le c, \quad \left|\frac{\partial \upsilon_t}{\partial \theta}(\theta)\right| \le c_1$$

$$\left|\frac{\partial \upsilon_t}{\partial \theta}(\theta) - \frac{\partial \upsilon_t}{\partial \theta}(\theta^*)\right| \le c_2 |\theta - \theta^*|$$

for all $t \in Z$ *and all* $\theta, \theta^* \in \Theta$.

3. The function $f_t(\theta, \Xi)$ *in (a.123) is bounded, Lipschitz in* θ *and* Ξ *uniformly with respect to* t, $\theta \in \Theta$ *and* Ξ *in compact sets.*

The first part of Assumption A.2 is to make up a compact set of models around θ^*, the models within the set can produce exponentially stable closed loop systems. The second part assumes smoothness of the closed loop response and its sensitivity with respect to θ during a frozen time interval that the parameter vector θ is presumed to be constant. This can be made possible as a result of applying the assumption of slow parameter adaptation that is the parameter update equation having a small adaptation gain. The appropriate correction function $f_t(\theta, \Xi)$ is characterized by the third condition. The function must be bounded and be able to have its first derivative with respect to θ and Ξ in the compact set. This is also the implication of having smoothness and continuity of

$f_t(\theta,\Xi)$ over the compact set. Like Assumption A.1 that defines the hypersphere $B_r(\theta^{**})$ of a certain size for the prediction error minimizing θ^* to stay within, in Assumption A.2 a compact set is defined by some smoothness conditions to address an appropriate region where the solution of the recursive algorithm possesses much potential to stay close to the off-line minimizing θ^*. the following integral manifold theory based upon the presumed compact set can indeed show us that the above potential is possible. According to (10), it is possible to consider dynamics of a total indirect adaptive control system to have two separated time scales to proceed: one is for dynamics of Ξ and another is for that of θ, which is assumed to proceed much slower with respect to that of Ξ. Based on this time scale separation, the solution the recursive identification algorithm can be approximated by the solution of an averaged equation,

$$\overline{\theta}_{k+1} = \overline{\theta}_k + \gamma \bar{f}(\overline{\theta}_k), \tag{a.126}$$

where $\bar{f}$ comes from averaging f_k over the interval that $\overline{\theta}$ is assumed to be fixed, Hence, asymptotic behavior of $\overline{\theta}$ can be represented by the solution of the ordinary differential equation,

$$\frac{d\hat{\theta}}{dt} = \bar{f}(\hat{\theta}). \tag{a.127}$$

The following theorem will proof that the asymptotic solution of the above is close to the off-line prediction error minimizing θ^*, provided the minimized cost $\overline{V}(\theta^*)$ is small enough and there exists persistency of excitation for any θ within the compact set.

Theorem A.7

Consider θ^* as the off-line solution of the prediction error misinformation problem,

$$\theta^* = \arg\min_{\theta \in B_\kappa(\theta^*)} \overline{V}(\theta,\eta(\theta),\rho(\theta)), \tag{a.128}$$

where $B_K(\theta^*)$ *represents the compact set around* θ^* *satisfying Assumption A.3,*

$$B_k(\theta^*) = \{\theta \in \mathbf{R}^d : |\theta - \theta^*| \le k\}. \tag{a.129}$$

The size of its interior is defined such that the vector of the filtered regressor ψ_k^f *is persistently exciting for* $\forall \theta \in B_K(\theta^*)$. *If* $\overline{V}(\theta^*)$ *is small enough, then*

1. the ODE ((a.127)has asymptotically stable equilibrium point θ^o *such that*

$$|\theta^o - \theta^*| \le b\overline{V}(\theta^*) \quad \text{for some finite } b; \tag{a.130}$$

2. given $\chi > 0$, *there exists a sufficiently small* $\gamma^*(\chi)$ *such that, for* $\gamma \in (0, \gamma^*)$, *the equation (a.123) possesses a bounded uniformly asymptotically stable solution* $\tilde{\theta}_k(\gamma)$ *which is close to* θ^o,

$$\lim_{\gamma \to 0} |\tilde{\theta}_k(\gamma) - \theta^o| = 0; \tag{a.131}$$

3. every solution $\theta_k(\gamma)$ *of (a.123) with* $\theta_0(\gamma) \in B_{K-\chi}(\theta^*)$ *satisfies, for* $\gamma \in (0, \gamma^*)$,

$$\theta_k(\gamma) \in B_K(\theta^*), \quad \lim_{k \to \infty} |\theta_k(\gamma) - \tilde{\theta}_k(\gamma)| = 0. \tag{a.132}$$

The proof and detailed mathematical manipulation can be found in (10) and (11). We shall not repeat them here for the sake of brevity.

Theorem A.7 is based on the compact set from Assumption A.3 with additional conditions of persistency of excitation of th filtered regressor for any θ in the set and if the minimizing cost is

small enough. The first result tells us that the asymptotic solution of the averaged parameter update equation will converge to the model θ^o that is close to the model θ^*. How far θ^o is with relative to θ^* is proportional to the amount of the minimizing cost $\overline{V}(\theta^*)$. We, therefore, see that the quality of the averaged solution of the recursive algorithm to be close to the off-line solution is dependent upon the value of the minimizing prediction error, hence, suitably restricted unmodeled dynamics is important to this content. The second result provides the existence of a bounded γ^* of the adaptation gain and states that for lower γ, from a bounded γ^*, the recursive solution $\tilde{\theta}(\gamma)$ at any time k, during a presumed θ-frozen time, will be guaranteed to be close to the ODE solution in the first result. The third result expresses the properties of the recursive algorithm solution under all the above conditions that the initial condition $\theta_0(\gamma)$ made sufficiently close to θ^* by the first and second results, as time proceeds, every estimate obtained from the recursive algorithm will be kept closer and closer to the asymptotically stable solution $\tilde{\theta}(\gamma)$ in the second result. These will infer the close connection from each available estimate obtained from the recursive identification at each time step to the off-line solution θ^*. Thus, from Lemma A.3, this also draw close connection with the LQ control optimizing model θ^{**}. Although, this inference is somewhat complicated, it can really show us that there is such a connection that can yield stable cooperation between the LQ regulator and the RLS identifier indeed.

APPENDIX B

Control Program

```
/* Hardware_oriented include file for SCS, Chula2: file; hardw_1.h */

#include <stdio.h>
#include <dos.h>
#include <math.h>
#include <process.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

/* declaration of symbolic constants */

#define IRQ9            0x71    /* hardware interrupt 9 */
#define IRQ10           0x72    /* hardware interrupt 10 */
#define IRQ11           0x73    /* hardware interrupt 11 */
#define IRQ12           0x74    /* hardware interrupt 12 */
#define IRQ14           0x76    /* hardware interrupt 14 */
#define IRQ15           0x77    /* hardware interrupt 15 */
#define GIO_U5PA        0x1280  /* GIO card, u5 device, I/O port A addr. */
#define GIO_U5PB        0x1281  /* GIO card, u5 device, I/O port B addr. */
#define GIO_U5PC        0x1282  /* GIO card, u5 device, I/O port C addr. */
#define GIO_U5CW        0x1283  /* GIO card, u5 device, control word addr.*/
#define GIO_U6CT0       0x1284  /* GIO card, u6 device, counter 0 addr. */
#define GIO_U6CT1       0x1285  /* GIO card, u6 device, counter 1 addr. */
#define GIO_U6CT2       0x1286  /* GIO card, u6 device, counter 2 addr. */
#define GIO_U6CW        0x1287  /* GIO card, u6 device, control word addr. */
#define GIO_U7PA        0x1294  /* GIO card, u7 device, I/O port A addr. */
#define GIO_U7PB        0x1295  /* GIO card, u7 device, I/O port B addr. */
#define GIO_U7PC        0x1296  /* GIO card, u7 device, I/O port C addr. */
#define GIO_U7CW        0x1297  /* GIO card, u7 device, control word addr.*/
#define PA_PB_MODE1 0xAE    /* I/O port A and B, mode 1 */
#define CT0_SW_BCD  0x37    /* counter 0, square wave generator, BCD code */
#define CT1_ITC_BIN 0x70    /* counter 1, interrupt on terminal count, BIN */
#define CT2_SW_BCD  0x97    /* counter 2, square wave generator, BCD */
#define CT0_500KHZ_LO  0x04   /* counter 0, LSB count value,100 kHz output */
#define CT0_500KHZ_HI  0x00   /* counter 0, MSB count value,100 kHz output */
#define CT1_TIMER_COUNT 0xFF      /* counter 1, full counter load, 132 ms */
#define CT2_500KHZ  0x04        /* counter 2, 500 kHz pulse generator */
#define PA_I_PBC_O  0x90        /* I/O port A; input, port B and C; output */
#define CLAMP_ACT   0x0C        /* amplifier clamp active code */
#define INH_1       0x02  /* amplifier inhibit code 1 */
#define INH_2       0x04  /* amplifier inhibit code 2 */
#define AMP_1_ACT   0x07        /* amplifier #1 clamp active bit "1" */
#define AMP_1_INA   0x06        /* amplifier #1 clamp inactive bit "0" */
```

```
#define AMP_2_ACT   0x09        /* amplifier #2 clamp active bit "1" */
#define AMP_2_INA   0x08        /* amplifier #2 clamp inactive bit "0" */
#define AMP_3_ACT   0x0B        /* amplifier #3 clamp active bit "1" */
#define AMP_3_INA   0x0A        /* amplifier #3 clamp inactive bit "0" */
#define ADSTART                 0x0B    /* output byte to start A/D converter */
#define ADSTOP                  0x0A    /* output byte to stop A/D converter */
#define PAGE_REG7       0x008A /* address of 7th DMA page register */
#define CBPFF           0x00D8 /* address of clear byte pointer flip-flop */
#define BASE_ADDR_7  0x00CC /* address of 7th DMA base address register*/
#define BASE_WCNT_7  0x00CE /* address of 7th DMA base word count reg. */
#define COMMAND_REG         0x00D0 /* address of 7th DMA command register */
#define MODE_REG        0x00D6 /* address of 7th DMA mode register */
#define MASK_REG_BIT0x00DE /* address of DMA mask bit register */
#define NUM_DMA_WORD_L   0x07    /* lsb of number of DMA word transfer */
#define NUM_DMA_WORD_H   0x00    /* msb of number of DMA word transfer */
#define COMMAND_WORD     0x00    /* command word */
#define MODE_WORD    0x57    /* mode word */
#define MASK_WORD    0x06    /* mask word : ch7 enable */
#define MCO1             0x11    /* MUX channel selection byte of MCO1 */
#define MCO2             0x33    /* MUX channel selection byte of MCO2 */
#define MCO3          0x55       /* MUX channel selection byte of MCO3 */
#define LSB              0x12A3 /* LSB port address of A/D output */
#define MSB              0x12A4 /* MSB port address of A/D output */
#define DA1              0x12A0 /* D/A port address, channel #1 */
#define DA2              0x12A1 /* D/A port address, channel #2 */
#define DA3              0x12A2 /* D/A port address, channel #3 */
#define ZEROVOLT     0x80    /* D/A zero voltage code */
#define MTR_TRQ_CST1         0.118554 /* motor torque constant 1 (Nm/A) */
#define MTR_TRQ_CST2         0.040986 /* motor torque constant 2 (Nm/A) */
#define MTR_TRQ_CST3         0.083270 /* motor torque constant 3 (Nm/A) */
#define AD_DELAY   40                 /* A/D conversion delay for 500 kHz clock */

/* macro to use two general purpose hardware strobe signals */
#define TTRAJ_HIGH    outportb(GIO_U7CW, 0x0F); /* TTRAJ output high */
#define TTRAJ_LOW     outportb(GIO_U7CW, 0x0E); /* TTRAJ output low */
#define TSAMP_HIGH    outportb(GIO_U7CW, 0x0D); /* TSAMP output high */
#define TSAMP_LOW     outportb(GIO_U7CW, 0x0C); /* TSAMP output low */

#define  NUMJOINTS    3       /* number of joints of the manipulator */
#define NUMPOINTS     500     /* number of data points to be collected */
#define NUMTRANS      250 /* max. number of com1-com2 transmission buffers */
#ifndef RAND_MAX
#define RAND_MAX 32767
#endif

/* macros for two-computer communication */
#define ENABLE_OUT   outportb(GIO_U5CW, 0x0D);
#define DISABLE_OUT  outportb(GIO_U5CW, 0x0C);
#define ENABLE_IN    outportb(GIO_U5CW, 0x05);
#define DISABLE_IN   outportb(GIO_U5CW, 0x04);
#define ADAPTATION   geninterrupt(IRQ12); /* invoke adaptive mechanisms */

/* A set of macro to insert End Of Interrupt command */
#define EOI_11  outportb(0x20, 0x62); outportb(0xA0, 0x63);
#define EOI_12  outportb(0x20, 0x62); outportb(0xA0, 0x64);
```

```
#define EOI_15  outportb(0x20, 0x62); outportb(0xA0, 0x67);
typedef unsigned char unc;

/* declaration of external variables */

unsigned long encoder[4];           /* DMA transfer buffers, 16-byte frame */
float ang_rad[NUMJOINTS];       /* joint angle in rad(joint 3 translation in m)*/
float angrate_rad[NUMJOINTS]; /* joint velocity in rad/sec(joint 3; m/sec) */
float ang2rate_rad[NUMJOINTS]; /* joint acceleration in rad/sec^2(m/sec^2) */
float ang_tmp[NUMJOINTS];   /* temporary angle for direction determination */
float angrate_tmp[NUMJOINTS];  /* temporary vel. for acc. computation */
float ang2rate_tmp[NUMJOINTS];          /* temporary acc. for filtering acc. */
float torque_msr[NUMJOINTS];   /* measured joint torques in Nm */
float x[2*NUMJOINTS];                   /* current state vector */
float x_ref[2*NUMJOINTS];               /* reference state vector */
float x_diff[2*NUMJOINTS];              /* differential state vector */
float T_operating[NUMJOINTS];  /* output joint torque in Nm. */
unsigned char ad_byte[2*NUMJOINTS];  /* A/D buffers */
unsigned int index;           /* index for starting control from SDP */
float timer;        /* time period captured by hardware timer (500 kHz) */
float Kp[NUMJOINTS] = {1100.0, 0.0, 0.0};       /* proportional gains */
float Kd[NUMJOINTS] = {3.0, 0.0, 0.0}; /* derivative gains */
float ttrj = 0.050;          /* time between trajectory points (sec) */
int trj_num = 500;          /* number of points to be performed */
float inp_amp[NUMJOINTS] = {1.0, 0.5, 0.1}; /* amplitudes of reference input signals */
float inp_frq[NUMJOINTS] = {0.5, 0.5, 0.5}; /* frequencies of reference input signals */
int char_index;             /* index to transmission charactors */
unsigned char trans_charactor[NUMTRANS];/* com1-com2 transmission buffers */

/* maximum magnitude of actual servo signals */
float max_ang_rad[NUMJOINTS] = {5.5, 2.5, 0.10}; /* rad, rad, m */
float max_angrate_rad[NUMJOINTS] = {10, 20, 0.50}; /* rad/s, rad/s, m/s */
float max_ang2rate_rad[NUMJOINTS] = {100.0, 200.0, 5.0};/* rd/s^2,rd/s^2,m/s^2*/
float max_torque_msr[NUMJOINTS] = {500.0, 400.0, 200.0};/* Nm, Nm, N */

/* parameters for bin-to-volt function of the A/D converter */
float AD_slope[NUMJOINTS] = {-0.00491, -0.00491, -0.00491}; /* volt/step */
float AD_const[NUMJOINTS] = {10.06514, 10.06514, 10.06514}; /* axis intercept */

/* parameters for volt-to-bin functions of D/A converters */
float DA_slope[NUMJOINTS] = {-13.191, -13.168, -13.115}; /* step/volt */
float DA_const[NUMJOINTS] = {128.002, 128.354, 128.593}; /* axis intercept */

/* motor+amp first-order constants for each joint */
float mtr_amp_cst[NUMJOINTS] = {0.26113, 0.09028, 0.18341};/* Nm/volt */

/* transmission ratio of each joint compositon */
float trans_ratio[NUMJOINTS] = {160.00, 160.00, 618.42375 /* rad/m */ };

/* transmission efficiency: 98%, 99% and 90% for joint 1,2,3 respectively */
float trans_efficiency[NUMJOINTS] = {0.98, 0.99, 0.90};

float time_point[NUMPOINTS];   /* discrete time points associated to data */
float data_1[NUMPOINTS];       /* trajectory of joint 1 angle */
float data_2[NUMPOINTS];       /* tracking history of joint 1 angle */
```

```
float data_3[NUMPOINTS];        /* trajectory of joint 2 angle */
float data_4[NUMPOINTS];        /* tracking history of joint 2 angle */
float data_5[NUMPOINTS];        /* trajectory of joint 3 displacement */
float data_6[NUMPOINTS];        /* tracking history of joint 3 displacement */
float data_7[NUMPOINTS];        /* trajectory of joint 1 velocity */
float data_8[NUMPOINTS];        /* tracking history of joint 1 velocity */
float data_9[NUMPOINTS];        /* trajectory of joint 2 velocity */
float data_10[NUMPOINTS];       /* tracking history of joint 2 velocity */
float data_11[NUMPOINTS];       /* trajectory of joint 3 velocity */
float data_12[NUMPOINTS];       /* tracking history of joint 3 velocity */
float data_13[NUMPOINTS];       /* trajectory of joint 1 acceleration */
float data_14[NUMPOINTS];       /* tracking history of joint 1 acceleration */
float data_15[NUMPOINTS];       /* trajectory of joint 2 acceleration */
float data_16[NUMPOINTS];       /* tracking history of joint 2 acceleration */
float data_17[NUMPOINTS];       /* trajectory of joint 3 acceleration */
float data_18[NUMPOINTS];       /* tracking history of joint 3 acceleration */
float data_19[NUMPOINTS];       /* history of output torque of joint 1 */
float data_20[NUMPOINTS];       /* history of measured torque of joint 1 */
float data_21[NUMPOINTS];       /* history of output torque of joint 2 */
float data_22[NUMPOINTS];       /* history of measured torque of joint 2 */
float data_23[NUMPOINTS];       /* history of output toruqe of joint 3 */
float data_24[NUMPOINTS];       /* history of measured torque of joint 3 */
int data_pnt;                   /* index to data points for data_collect function */
int now;                /* variable contains seed number for random func. */
/* function prototypes */

void power_up(void);
unsigned long get_raddr(unsigned long *iptr);
void install_isr(void interrupt (*faddr)(), int inum);
void program_8237_2(unc addr1, unc addr2, unc addr3);
void interrupt init_DMA7(void);
void FAC1_active(void);
void RAC1_active(void);
void MHC1_active(void);
void FAC2_active(void);
void RAC2_active(void);
void MHC2_active(void);
void FAC3_active(void);
void RAC3_active(void);
void MHC3_active(void);
void AMP1_ready(void);
void AMP2_ready(void);
void AMP3_ready(void);
void data_sampling(void);
void lowlevel_setup(void);
void lowlevel_reset(void);
void out_torque(float *iptr);
void data_collect(void);
void data_write(void);
void home(void);
double uniform(void);
void SDP_communication_setup(void);

void power_up(void)
{
```

```
        outportb(DA1,ZEROVOLT);
        outportb(DA2,ZEROVOLT);
        outportb(DA3,ZEROVOLT);
        outportb(GIO_U5CW,PA_PB_MODE1);
        DISABLE_OUT
        DISABLE_IN
        outportb(GIO_U6CW,CT0_SW_BCD);
        outportb(GIO_U6CW,CT1_ITC_BIN);
        outportb(GIO_U6CW,CT2_SW_BCD);
        outportb(GIO_U6CT0,CT0_500KHZ_LO);
        outportb(GIO_U6CT0,CT0_500KHZ_HI);
        outportb(GIO_U6CT2,CT2_500KHZ);
        outportb(GIO_U7CW,PA_I_PBC_O);
        outportb(GIO_U7CW,CLAMP_ACT);
        outportb(GIO_U7CW,INH_1);
        outportb(GIO_U7CW,INH_2);
        outportb(GIO_U7CW,AMP_1_ACT);
        outportb(GIO_U7CW,AMP_1_INA);
        outportb(GIO_U7CW,AMP_2_ACT);
        outportb(GIO_U7CW,AMP_2_INA);
        outportb(GIO_U7CW,AMP_3_ACT);
        outportb(GIO_U7CW,AMP_3_INA);
        outportb(GIO_U7PB,MCO1);
        printf("CHULA2 HARDWARE INTERFACE IS READY\n");
}

void install_isr(void interrupt (*faddr)(), int inum)
{
        setvect(inum, faddr);
}

unsigned long get_raddr(unsigned long *iptr)
{
        unsigned long seg,
        raddr;
        unsigned int off;

        seg = FP_SEG(iptr);
        off = FP_OFF(iptr);
        seg <<= 4;
        raddr = seg + off;
        return raddr;
}

void program_8237_2(unc pag_reg, unc base_addr_hi, unc base_addr_lo)
{
        outportb(PAGE_REG7, pag_reg); /* coding page register of ch7 */
        outportb(CBPFF, 0x00);  /* clear byte pointer flip-flop */
        outportb(BASE_ADDR_7, base_addr_lo); /* LSB of base addr register */
        outportb(BASE_ADDR_7, base_addr_hi); /* MSB of base addr register */
        outportb(BASE_WCNT_7, NUM_DMA_WORD_L); /* LSB of No. of DMA words */
        outportb(BASE_WCNT_7, NUM_DMA_WORD_H); /* MSB of N0. of DMA words */
        outportb(COMMAND_REG, COMMAND_WORD); /* program 8237 command reg. */
        outportb(MODE_REG, MODE_WORD); /* program 8237 mode register */
        outportb(MASK_REG_BIT, MASK_WORD);/* program 8237 mask register bit*/
```

```
}

void interrupt init_DMA7(void)
{
        unsigned int tmp3;
        unsigned long addr, tmp1, tmp2;
        unsigned char page_reg7, base_addrh, base_addrl;
        extern unsigned long encoder[4]; /* storage for DMA transfer */
        extern unsigned int index;

        addr = get_raddr(encoder);          /* get physical address */
        tmp1 = addr & 0x00FE0000ul;
        tmp1 >>= 16;
        page_reg7 = tmp1;          /* get 7th DMA page register */
        tmp2 = addr & 0x0001FE00ul;
        tmp2 >>= 9;
        base_addrh = tmp2;
        tmp3 = addr & 0x000001FEul;
        tmp3 >>= 1;
        base_addrl = tmp3;
        program_8237_2(page_reg7, base_addrh, base_addrl);
        printf("DMA controller setup\n");
        printf("Servo data transmission begins\n");
        index = 1; /*index to main controller that SDP begins transmitting*/
        EOI_15
}

void FAC1_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x02);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x06);
        outportb(GIO_U7CW, 0x07);
        outportb(GIO_U7CW, 0x06);
        printf("joint 1, clockwise inhibited\n");
}

void RAC1_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x06);
        outportb(GIO_U7CW, 0x07);
        outportb(GIO_U7CW, 0x06);
        printf("joint 1, counter clockwise inhibited\n");
}

void MHC1_active(void)
{
        outportb(DA1, ZEROVOLT);
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x04);
```

```
        outportb(GIO_U7CW, 0x06);
        outportb(GIO_U7CW, 0x07);
        outportb(GIO_U7CW, 0x06);
        printf("Motor 1 electrically braked\n");
}

void AMP1_ready(void)
{
        outportb(GIO_U7CW, 0x0D);
        outportb(GIO_U7CW, 0x06);
        outportb(GIO_U7CW, 0x07);
        outportb(GIO_U7CW, 0x06);
        printf("Amplifiler # 1 ready\n");
}

void FAC2_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x02);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x08);
        outportb(GIO_U7CW, 0x09);
        outportb(GIO_U7CW, 0x08);
        printf("joint 2, clockwise inhibited\n");
}

void RAC2_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x08);
        outportb(GIO_U7CW, 0x09);
        outportb(GIO_U7CW, 0x08);
        printf("joint 2, counter clockwise inhibited\n");
}

void MHC2_active(void)
{
        outportb(DA2, ZEROVOLT);
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x04);
        outportb(GIO_U7CW, 0x08);
        outportb(GIO_U7CW, 0x09);
        outportb(GIO_U7CW, 0x08);
        printf("Motor 2 electrically braked\n");
}

void AMP2_ready(void)
{
        outportb(GIO_U7CW, 0x0D);
        outportb(GIO_U7CW, 0x08);
        outportb(GIO_U7CW, 0x09);
        outportb(GIO_U7CW, 0x08);
```

```
        printf("Ampliflier # 2 ready\n");
}

void FAC3_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x02);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x0A);
        outportb(GIO_U7CW, 0x0B);
        outportb(GIO_U7CW, 0x0A);
        printf("joint 3, clockwise inhibited\n");
}

void RAC3_active(void)
{
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x05);
        outportb(GIO_U7CW, 0x0A);
        outportb(GIO_U7CW, 0x0B);
        outportb(GIO_U7CW, 0x0A);
        printf("joint 3, counter clockwise inhibited\n");
}

void MHC3_active(void)
{
        outportb(DA3, ZEROVOLT);
        outportb(GIO_U7CW, 0x0C);
        outportb(GIO_U7CW, 0x03);
        outportb(GIO_U7CW, 0x04);
        outportb(GIO_U7CW, 0x0A);
        outportb(GIO_U7CW, 0x0B);
        outportb(GIO_U7CW, 0x0A);
        printf("Motor 3 electrically braked\n");
}

void AMP3_ready(void)
{
        outportb(GIO_U7CW, 0x0D);
        outportb(GIO_U7CW, 0x0A);
        outportb(GIO_U7CW, 0x0B);
        outportb(GIO_U7CW, 0x0A);
        printf("Ampliflier # 3 ready\n");
}

void data_sampling(void)
{
        unsigned char i, time_lsb, time_msb;
        long tmp1, tmp3;
        float tmp4;
        unsigned int tmp2;
        unsigned char lsb, msb;
        extern unsigned long encoder[4];
        extern float ang_rad[NUMJOINTS];
```

```
extern float angrate_rad[NUMJOINTS];
extern float ang2rate_rad[NUMJOINTS];
extern float ang_tmp[NUMJOINTS];
extern float angrate_tmp[NUMJOINTS];
extern float ang2rate_tmp[NUMJOINTS];
extern float torque_msr[NUMJOINTS];
extern unsigned char ad_byte[2 * NUMJOINTS];
extern float timer;

/* calculate sampling time interval, sec */
outportb(GIO_U6CT1, 0xFF); /* stop and reload the timer */
time_lsb = inportb(GIO_U6CT1); /* read LSB count value on the timer */
time_msb = inportb(GIO_U6CT1); /* read MSB count value on the timer */
outportb(GIO_U6CT1, 0xFF); /* reload and start the timer */
tmp2 = (256*time_msb)+time_lsb; /* timer limited between .002-137.0 msec */
if (tmp2 == 65535) {printf("timer value is not valid\n"); exit(0);}
timer = ((65535 - tmp2) * 0.002e-3); /* interval ,seconds */

/* start data acquisition module */
for (i = 0; i < NUMJOINTS; i++) {
        outportb(GIO_U7PB, MCO1 + 0x22 * i); /* set MUX for MCOi */
        outportb(GIO_U5CW, ADSTART);         /* start A/D conversion */
        if (i == 0) {

                /* calculate angular position of joint 1, rad */
                tmp1 = encoder[0];
                tmp1 >>= 8;
                ang_rad[0] = tmp1 * 4.793690e-6;/* rad/step */

                /* software position limit of joint 1 */
                if (fabs(ang_rad[0]) > 3.0) {
                        MHC1_active();
                        printf("joint 1, limiter is active!\n");
                        printf("program terminated!\n");
                        exit(0);
                }

                /* calculate angular velocity of joint 1, rad/sec */
                tmp1 = encoder[2];
                tmp2 = tmp1 >>= 16;
                if (tmp2 == 0xFFFF)
                        angrate_rad[0] = 0.00;
                else if (tmp2 != 0xFFFF) {
                        if ((ang_rad[0] - ang_tmp[0]) > 0.00)
                                angrate_rad[0] = 9.587380 / (65535 - tmp2);
                        else if ((ang_rad[0] - ang_tmp[0]) < 0.00)
                                angrate_rad[0] = -9.587380 / (65535 - tmp2);
                }
                ang_tmp[0] = ang_rad[0];

                /* calculate acceleration of joint 1, rad/sec^2 */
                tmp4 = angrate_rad[0] - angrate_tmp[0];
                ang2rate_rad[0] = tmp4/timer;
                if (fabs(ang2rate_rad[0]) > 6.0) {
                        angrate_rad[0] = angrate_tmp[0];
```

```
                ang2rate_rad[0] = ang2rate_tmp[0];
            }
            angrate_tmp[0] = angrate_rad[0];
            ang2rate_tmp[0] = ang2rate_rad[0];

            /* calculate measured torque of joint 1, Nm. */
            outportb(GIO_U5CW, ADSTOP); /* stop A/D conversion */
            ad_byte[0] = lsb = inportb(LSB);
            ad_byte[1] = msb = inportb(MSB) & '\x0F';
            torque_msr[0] = -1.0*(AD_slope[0]*((msb*256)+lsb)+AD_const[0])
            *mtr_amp_cst[0]*trans_ratio[0]*trans_efficiency[0];
}
else if (i == 1) {

            /* calculate angular position of joint 2, rad */
            tmp1 = encoder[1];
            tmp1 <<= 8;
            tmp1 >>= 8;
            ang_rad[1] = tmp1 * 9.817477e-6;/* rad/step */

            /* calculate angular velocity of joint 2, rad/sec */
            tmp1 = encoder[3];
            tmp1 <<= 16;
            tmp2 = tmp1 >>= 16;
            if (tmp2 == 0xFFFF)
                angrate_rad[1] = 0.00;
            else if (tmp2 != 0xFFFF) {
                if ((ang_rad[1] - ang_tmp[1]) > 0.00)
                    angrate_rad[1] = 19.634954 / (65535 - tmp2);
                else if ((ang_rad[1] - ang_tmp[1]) < 0.00)
                    angrate_rad[1] = -19.634954 / (65535 - tmp2);
            }
ang_tmp[1] = ang_rad[1];

            /* calculate acceleration of joint 2, rad/sec^2 */
            tmp4 = angrate_rad[1] - angrate_tmp[1];
            ang2rate_rad[1] = tmp4/timer;
            if (fabs(ang2rate_rad[1]) > 10.0) {
                angrate_rad[1] = angrate_tmp[1];
                ang2rate_rad[1] = ang2rate_tmp[1];
            }
            angrate_tmp[1] = angrate_rad[1];
            ang2rate_tmp[1] = ang2rate_rad[1];

            /* calculate measured toruqe of joint 2, Nm.*/
            outportb(GIO_U5CW, ADSTOP);
            ad_byte[2] = lsb = inportb(LSB);
            ad_byte[3] = msb = inportb(MSB) & '\x0F';
            torque_msr[1] = -1.0*(AD_slope[1]*((msb*256)+lsb)+AD_const[1])
            *mtr_amp_cst[1]*trans_ratio[1]*trans_efficiency[1];
}
else if (i == 2) {

            /* calculate linear position of joint 3, meters. */
            tmp1 = encoder[1];
```

```
                        tmp3 = tmp1 >>= 24;
                        tmp1 = encoder[2];
                        tmp1 <<= 16;
                        tmp1 >>= 8;
                        tmp1 += tmp3;
                        ang_rad[2] = tmp1 * 1.240234e-7; /* m/step */

                        /* calculate linear velocity of joint 3, m/sec */
                        tmp1 = encoder[3];
                        tmp2 = tmp1 >>= 16;
                        if (tmp2 == 0xFFFF)
                                angrate_rad[2] = 0.00;
                        else if (tmp2 != 0xFFFF) {
                                if ((ang_rad[2] - ang_tmp[2]) > 0.00)
                                        angrate_rad[2] = 0.248047 / (65535 - tmp2);
                                else if ((ang_rad[2] - ang_tmp[2]) < 0.00)
                                        angrate_rad[2] = -0.248047 / (65535 - tmp2);
                        }
                        ang_tmp[2] = ang_rad[2];

                        /* calculate acceleration of joint 3, m/sec^2 */
                        tmp4 = angrate_rad[2] - angrate_tmp[2];
                        ang2rate_rad[2] = tmp4/timer;
                        if (fabs(ang2rate_rad[2]) > 0.3) {
                                angrate_rad[2] = angrate_tmp[2];
                                ang2rate_rad[2] = ang2rate_tmp[2];
                        }
                        angrate_tmp[2] = angrate_rad[2];
                        ang2rate_tmp[2] = ang2rate_rad[2];

                        /* calculate translational force of joint 3, Newtons */
                        outportb(GIO_U5CW, ADSTOP); /* stop A/D conversion */
                        ad_byte[4] = lsb = inportb(LSB);
                        ad_byte[5] = msb = inportb(MSB) & '\x0F';
                        torque_msr[2] = -1.0*(AD_slope[2]*((msb*256)+lsb)+AD_const[2])
                        *mtr_amp_cst[2]*trans_ratio[2]*trans_efficiency[2];
                }
        }
}

void lowlevel_setup(void)
{
        install_isr(init_DMA7, IRQ15);
        outportb(0x21, 0xF8);   /* master controller: enable timer tick,
                                                        keyboard, slave controller */
        outportb(0xA1, 0x5F);   /* slave controller: enable coprocessor,
                                                        IRQ15 */
        outportb(0x70, 0x80);   /* off NMI */
        printf("setup BIOS parameters for SDP transmission\n");
}

void lowlevel_reset(void)
{
        outportb(GIO_U5CW,PA_PB_MODE1); /* refresh transmission mechanism */
        outportb(0x21, 0x00);   /* master controller, BIOS level standby */
```

```
        outportb(0xA1, 0x9C);    /* slave controller, BIOS level standby */
        outportb(0x70, 0x00);    /* on NMI */
        printf("reinstall BIOS parameters to the original condition\n");
}

void out_torque(float *torque)
{
        unsigned int i;
        unsigned int output_byte[NUMJOINTS];
        float volt[NUMJOINTS];

        for (i = 0; i < NUMJOINTS; i++) {

                /* convert actuating torque to voltage */
                volt[i] = torque[i]/(trans_efficiency[i]*
                        trans_ratio[i])/mtr_amp_cst[i];

                /* saturation limit of voltage value */
                if (volt[i] >= 10.0)
                        volt[i] = 10.0;
                else if (volt[i] <= -10.0)
                        volt[i] = -10.0;

                /* convert voltage to 8-bit binary code */
                output_byte[i] = (int)((DA_slope[i]*volt[i])+DA_const[i]);

                /* compensation for small deadzone in D/A + Amp */
                if (output_byte[i] >= 0x7B && output_byte[i] <= 0x82)
                        output_byte[i] = 0x80;
                else if (output_byte[i] >= 0x71 && output_byte[i] < 0x7B)
                        output_byte[i] = 0x71;
                else if (output_byte[i] > 0x82 && output_byte[i] <= 0x88)
                        output_byte[i] = 0x88;

                /* send the binary code to D/A converter */
                outportb(DA1 + i, output_byte[i]);
        }
}

void data_collect(void)
{
        extern float ang_rad[NUMJOINTS];
        extern float angrate_rad[NUMJOINTS];
        extern float ang2rate_rad[NUMJOINTS];
        extern float T_operating[NUMJOINTS];
        extern float torque_msr[NUMJOINTS];
        extern int data_pnt;
        extern float data_2[NUMPOINTS];
        extern float data_4[NUMPOINTS];
        extern float data_6[NUMPOINTS];
        extern float data_8[NUMPOINTS];
        extern float data_10[NUMPOINTS];
        extern float data_12[NUMPOINTS];
        extern float data_14[NUMPOINTS];
        extern float data_16[NUMPOINTS];
```

```
        extern float data_18[NUMPOINTS];
        extern float data_19[NUMPOINTS];
        extern float data_20[NUMPOINTS];
        extern float data_21[NUMPOINTS];
        extern float data_22[NUMPOINTS];
        extern float data_23[NUMPOINTS];
        extern float data_24[NUMPOINTS];

        data_2[data_pnt] = ang_rad[0];
        data_4[data_pnt] = ang_rad[1];
        data_6[data_pnt] = ang_rad[2];
        data_8[data_pnt] = angrate_rad[0];
        data_10[data_pnt] = angrate_rad[1];
        data_12[data_pnt] = angrate_rad[2];
        data_14[data_pnt] = ang2rate_rad[0];
        data_16[data_pnt] = ang2rate_rad[1];
        data_18[data_pnt] = ang2rate_rad[2];
        data_19[data_pnt] = T_operating[0];
        data_20[data_pnt] = torque_msr[0];
        data_21[data_pnt] = T_operating[1];
        data_22[data_pnt] = torque_msr[1];
        data_23[data_pnt] = T_operating[2];
        data_24[data_pnt] = torque_msr[2];
        data_pnt++;
}

void data_write(void)
{
        FILE *stream;
        extern float time_point[NUMPOINTS];
        extern float data_1[NUMPOINTS];
        extern float data_2[NUMPOINTS];
        extern float data_3[NUMPOINTS];
        extern float data_4[NUMPOINTS];
        extern float data_5[NUMPOINTS];
        extern float data_6[NUMPOINTS];
        extern float data_7[NUMPOINTS];
        extern float data_8[NUMPOINTS];
        extern float data_9[NUMPOINTS];
        extern float data_10[NUMPOINTS];
        extern float data_11[NUMPOINTS];
        extern float data_12[NUMPOINTS];
        extern float data_13[NUMPOINTS];
        extern float data_14[NUMPOINTS];
        extern float data_15[NUMPOINTS];
        extern float data_16[NUMPOINTS];
        extern float data_17[NUMPOINTS];
        extern float data_18[NUMPOINTS];
        extern float data_19[NUMPOINTS];
        extern float data_20[NUMPOINTS];
        extern float data_21[NUMPOINTS];
        extern float data_22[NUMPOINTS];
        extern float data_23[NUMPOINTS];
        extern float data_24[NUMPOINTS];
        int i;
```

```
        stream = fopen("d:\\pos.csv", "w");
        for (i = 0; i < NUMPOINTS; i++) {
        fprintf(stream, "%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_1[i], data_2[i], data_3[i], data_4[i], data_5[i],
        data_6[i]);
        }
        fclose(stream);
        stream = fopen("d:\\vel.csv", "w");
        for (i = 0; i < NUMPOINTS; i++) {
        fprintf(stream, "%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_7[i], data_8[i], data_9[i], data_10[i], data_11[i],
        data_12[i]);
        }
        fclose(stream);
        stream = fopen("d:\\acc.csv", "w");
        for (i = 0; i < NUMPOINTS; i++) {
        fprintf(stream, "%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_13[i], data_14[i], data_15[i], data_16[i],
        data_17[i], data_18[i]);
        }
        fclose(stream);
        stream = fopen("d:\\tor.csv", "w");
        for (i = 0; i < NUMPOINTS; i++) {
        fprintf(stream, "%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_19[i], data_20[i], data_21[i], data_22[i],
        data_23[i], data_24[i]);
        }
        fclose(stream);
        printf("data write complete\n");
}

void home(void)
{
        int i;
        float current_position[NUMJOINTS];

        AMP1_ready();
        AMP2_ready();
        AMP3_ready();
        for (i = 0; i < NUMJOINTS; i++) {
                current_position[i] = ang_rad[i];
                if (current_position[i] < 0.0) {
                        while (ang_rad[i] < -0.00002) {
                                outportb(DA1+i, 0x71);
                                data_sampling();
                        }
                }
                else if (current_position[i] > 0.0) {
                        while (ang_rad[i] > 0.00002) {
                                outportb(DA1+i, 0x88);
                                data_sampling();
                        }
                }
                data_sampling();
```

```
                switch (i) {
                        case 0 : MHC1_active();
                        printf("joint 1 home, %#6.6f rds\n", ang_rad[i]);break;
                        case 1 : MHC2_active();
                        printf("joint 2 home, %#6.6f rds\n", ang_rad[i]);break;
                        case 2 : MHC3_active();
                        printf("joint 3 home, %#6.6f rds\n", ang_rad[i]);break;
                        default : break;
                }
        }
}

double uniform()
{
   return((double)(rand() & RAND_MAX) / RAND_MAX - 0.5);
}

void SDP_communication_setup(void)
{
        extern unsigned int index;
        int key;
        long i;

        index = 0;
        lowlevel_setup();
        while (index == 0) {
                printf("Start SDP transmission\n");
                for (i = 0; i < 1000000; i++) {}
        }
        if (encoder[0] == 0xEA) {
        printf("Servo data transmission is correct!\n");
        printf("You can continue on selecting menu\n");
        }
        else {
        printf("mulfunction of transmission system!\n");
        }
}

/* PD controller for the Chula2 manipulator system: file; pdcon1.h */

/* function prototypes */

void refPD_update(int num);
void onepoint_movePD(void);
void controller_PD(void);

void refPD_update(int num)
{
        extern float data_1[NUMPOINTS];
        extern float data_3[NUMPOINTS];
        extern float data_5[NUMPOINTS];
        extern float data_7[NUMPOINTS];
        extern float data_9[NUMPOINTS];
        extern float data_11[NUMPOINTS];
        extern float x_ref[2*NUMJOINTS];
```

```
        x_ref[0] = data_1[num];
        x_ref[1] = data_3[num];
        x_ref[2] = data_5[num];
        x_ref[3] = data_7[num];
        x_ref[4] = data_9[num];
        x_ref[5] = data_11[num];
}

void onepoint_movePD(void)
{
        extern float x_ref[2*NUMJOINTS];
        extern float ang_rad[NUMJOINTS];
        extern float angrate_rad[NUMJOINTS];
        int i, j;

        for (i = 0; i < 22; i++) {  /* @ i = 22, ttrj = 50.0 msec */
                data_sampling();
                if (i == 10) TTRAJ_LOW
                for (j = 0; j < NUMJOINTS; j++)
                        T_operating[j] = - Kp[j]*(ang_rad[j] - x_ref[j])
                                         - Kd[j]*(angrate_rad[j] - x_ref[j+3]);
                out_torque(T_operating);
        }
        data_collect();
}

void controller_PD(void)  /* max. sampling frequency = 401.6064 Hz */
{
        extern float Kp[NUMJOINTS];
        extern float Kd[NUMJOINTS];
        extern float ttrj;
        extern int trj_num;
        extern float inp_amp[NUMJOINTS];
        extern float inp_frq[NUMJOINTS];
        int trj_idx[NUMJOINTS] = {4, 4, 4};
        int yy = 0, key;
        long n;

        printf("PD path tracking controller\n");
        while (yy == 0) {
                printf("1 => enter proportional gains (1100 0 0)\n");
                printf("2 => enter derivative gains (3 0 0)\n");
                printf("3 => select standard input (4 4 4)\n");
                printf("4 => enter input amplitude (1.0 0.5 0.1)\n");
                printf("5 => enter input frequency (0.5 0.5 0.5)\n");
                printf("6 => enter trajectory period (0.050)\n");
                printf("7 => number of trajectory points (500)\n");
                printf("8 => start control\n");
                scanf("%d", &key);
                switch (key) {
                        case 1: scanf("%f %f %f", Kp, Kp+1, Kp+2); break;
                        case 2: scanf("%f %f %f", Kd, Kd+1, Kd+2); break;
                        case 3: scanf("%d %d %d", trj_idx,trj_idx+1,trj_idx+2);break;
                        case 4: scanf("%f %f %f", inp_amp,inp_amp+1,inp_amp+2);break;
```

```
                        case 5: scanf("%f %f %f", inp_frq,inp_frq+1,inp_frq+2);break;
                        case 6: scanf("%f", &ttrj); break;
                        case 7: scanf("%d", &trj_num); break;
                        case 8: yy = 1; break;
                        default : exit(0);
                }
        }
        make_ref(trj_idx);
        AMP1_ready();
        AMP2_ready();
        AMP3_ready();
        for (n = 0; n < trj_num; n++) { /* ttrj = 50.0 msec */
                TTRAJ_HIGH
                refPD_update(n);
                onepoint_movePD();
        }
        MHC1_active();
        MHC2_active();
        MHC3_active();
        printf("end of path tracking operation\n");
}

./* self-tuning controller program for SCS, file : STCON1.H */

/* declaration of external variables */

float T_nominal[NUMJOINTS];    /* nominal torque along trajectory (Nm.) */
float K[NUMJOINTS][2*NUMJOINTS];  /* adaptive gain matrix [3*6] */


/* function prototypes */

void lowlevel_setup_ST(void);      /* BIOS-parameters setup routine */
void interrupt com1_com2_ST(void);        /* com1-com2 transmitting routine */
void interrupt com2_com1_ST(void);        /* com2-com1 transmitting routine */
void do_trans_ST(void);            /* routine for charactor transmitting control */
void trans_torque_msr(void);/* measured torques picked up for transmitting */
void trans_ang_rad(void);   /* joint angle picked up for transmitting */
void trans_angrate_rad(void);/* joint velocity picked up for transmitting */
void trans_ang2rate_rad(void);/* joint accel. picked up for transmitting */
void trans_x_diff(void);   /* state eror vector picked up for transmitting */
void receiv_T_nominal(void);/* forming received charactors as T_nominal */
void receiv_K(void);               /* forming received charactors as K */
void setting_ST_PD(void);          /* parameter setting changing ST to be PD scheme */
void refST_update(int num); /* update reference input to controller */
void onepoint_moveST(void);        /* moving one point along traj. in ST manner */
void controller_ST(void); /* performing ST controller */

void lowlevel_setup_ST(void)
{
        install_isr(com1_com2_ST, IRQ12);
        install_isr(com2_com1_ST, IRQ11);
        outportb(0x21, 0x00);      /* master controller: enable all */
        outportb(0xA1, 0x00);      /* slave controller: enable all */
        outportb(0x70, 0x80);      /* off NMI */
```

```
        DISABLE_OUT
        DISABLE_IN
        printf("setup BIOS-parameters for operating Self-tunning controller\n");
}

void interrupt com1_com2_ST(void)
{
        EOI_12
        do_trans_ST();
}

void interrupt com2_com1_ST(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];

        EOI_11
        switch (char_index) {
                case 39: DISABLE_IN receiv_T_nominal(); do_trans_ST(); break;
                case 95: DISABLE_IN receiv_K(); do_trans_ST(); break;
                case 151: DISABLE_IN receiv_K(); char_index = 0;
                        TTRAJ_LOW break;
                default:trans_charactor[char_index] = inportb(GIO_U5PB);
                        char_index++;
        }
}

void do_trans_ST(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];

        switch (char_index) {
                case 0: ENABLE_OUT TTRAJ_HIGH  trans_torque_msr(); break;
                case 8: trans_ang_rad(); break;
                case 16: trans_angrate_rad(); break;
                case 24: trans_ang2rate_rad(); break;
                case 31: DISABLE_OUT ENABLE_IN
                 outportb(GIO_U5PA, trans_charactor[char_index]);
                        char_index++;break;
                case 40: ENABLE_OUT trans_torque_msr(); break;
                case 48: trans_x_diff(); break;
                case 63: DISABLE_OUT ENABLE_IN
                 outportb(GIO_U5PA, trans_charactor[char_index]);
                        char_index++;break;
                case 96; ENABLE_OUT trans_torque_msr(); break;
                case 104: trans_x_diff(); break;
                case 119: DISABLE_OUT ENABLE_IN
                default: outportb(GIO_U5PA, trans_charactor[char_index]);
                                char_index++;
        }
}

void trans_torque_msr(void)
{
```

```
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float torque_msr[NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = torque_msr;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(GIO_U5PA, trans_charactor[char_index]);
        char_index++;
}

void trans_ang_rad(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float ang_rad[NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = ang_rad;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(GIO_U5PA, trans_charactor[char_index]);
        char_index++;
}

void trans_angrate_rad(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float angrate_rad[NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = angrate_rad;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(GIO_U5PA, trans_charactor[char_index]);
        char_index++;
}

void trans_ang2rate_rad(void)
{
        extern int trans_char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float ang2rate_rad[NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = ang2rate_rad;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(GIO_U5PA, trans_charactor[char_index]);
```

```
        char_index++;
}

void trans_x_diff(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float x_diff[2*NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = x_diff;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        iptr = iptr + 4;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+8+i] = *(iptr+i);
        outportb(GIO_U5PA, trans_charactor[char_index]);
        char_index++;
}

void receiv_T_nominal(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float T_nominal[NUMJOINTS];
        unsigned char *iptr;
        int i;

        trans_charactor[char_index] = inportb(GIO_U5PB);
        iptr = T_nominal;
        for (i = 0; i < 8; i++)
                *(iptr+i) = trans_charactor[char_index-7+i];
        char_index++;
}

void receiv_K(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float K[NUMJOINTS][2*NUMJOINTS];
        unsigned char *iptr;
        int i;

        trans_charactor[char_index] = inportb(GIO_U5PB);
        iptr = K;
        for (i = 0; i < 8; i++)
                *(iptr+i) = trans_charactor[char_index-31+i];
        iptr += 12;
        for (i = 0; i < 8; i++)
                *(iptr+i) = trans_charactor[char_index-23+i];
        iptr += 12;
        for (i = 0; i < 8; i++)
                *(iptr+i) = trans_charactor[char_index-15+i];
        iptr += 12;
```

```
        for (i = 0; i < 8; i++)
                *(iptr+i) = trans_charactor[char_index-7+i];
        char_index++;
}

void setting_ST_PD(void)
{
        extern float Kp[NUMJOINTS];
        extern float Kd[NUMJOINTS];
        extern float T_nominal[NUMJOINTS];
        extern float K[NUMJOINTS][2*NUMJOINTS];
        int i;

        for (i = 0; i < NUMJOINTS; i++)
                T_nominal[i] = 0.0;
        K[0][1] = K[0][2] = K[0][4] = K[0][5] = 0.0;
        K[1][0] = K[1][2] = K[1][3] = K[1][5] = 0.0;
        K[2][0] = K[2][1] = K[2][3] = K[2][4] = 0.0;
        for (i = 0; i < NUMJOINTS; i++) {
                K[i][i] = Kp[i];
                K[i][i+3] = Kd[i];
        }
}

void refST_update(int num)
{
        extern float data_1[NUMPOINTS];
        extern float data_3[NUMPOINTS];
        extern float data_5[NUMPOINTS];
        extern float data_7[NUMPOINTS];
        extern float data_9[NUMPOINTS];
        extern float data_11[NUMPOINTS];
        extern float x_ref[2*NUMJOINTS];

        x_ref[0] = data_1[num];
        x_ref[1] = data_3[num];
        x_ref[2] = data_5[num];
        x_ref[3] = data_7[num];
        x_ref[4] = data_9[num];
        x_ref[5] = data_11[num];
}

void onepoint_moveST(void)
{
        extern float x[2*NUMJOINTS];
        extern float x_ref[2*NUMJOINTS];
        extern float x_diff[2*NUMJOINTS];
        extern float T_nominal[NUMJOINTS];
        extern float T_operating[NUMJOINTS];
        int i, j, l;

        for (i = 0; i < 17; i++) { /* @ i = 1, exec. time must be 4 ms */
                data_sampling();
                for (j = 0; j < NUMJOINTS; j++) {
                        x[j] = ang_rad[j];
```

```
                x[j+3] = angrate_rad[j];
            }
            for (j = 0; j < (2*NUMJOINTS); j++)
                x_diff[j] = x[j] - x_ref[j];
            for (j = 0; j < NUMJOINTS; j++) {
                T_operating[j] = T_nominal[j];
                for (l = 0; l < (2*NUMJOINTS); l++)
                    T_operating[j] -= K[j][l] * x_diff[l];
            }
            out_torque(T_operating);
        }
        data_collect();
}

void controller_ST(void)  /* 50 msec per point */
{
        extern float Kp[NUMJOINTS];
        extern float Kd[NUMJOINTS];
        extern float ttrj;
        extern int trj_num;
    extern float inp_amp[NUMJOINTS];
        extern float inp_frq[NUMJOINTS];
        int trj_idx[NUMJOINTS] = {4, 4, 4};
        int yy = 0, key;
        long n;

        lowlevel_setup_ST();
        printf("Self-tuning controller\n");
        while (yy == 0) {
            printf("1 => enter proportional gains (1100 0 0)\n");
            printf("2 => enter derivative gains (3 0 0)\n");
            printf("3 => select standard input (4 4 4)\n");
            printf("4 => enter input amplitude (1.0 0.5 0.10)\n");
            printf("5 => enter input frequency (0.5 0.5 0.5)\n");
            printf("6 => enter trajectory period (0.050)\n");
            printf("7 => number of trajectory points (500)\n");
            printf("8 => start control\n");
            scanf("%d", &key);
            switch (key) {
                case 1: scanf("%f %f %f", Kp, Kp+1, Kp+2); break;
                case 2: scanf("%f %f %f", Kd, Kd+1, Kd+2); break;
                case 3: scanf("%d %d %d", trj_idx,trj_idx+1,trj_idx+2);break;
                case 4: scanf("%f %f %f", inp_amp,inp_amp+1,inp_amp+2);break;
                case 5: scanf("%f %f %f", inp_frq,inp_frq+1,inp_frq+2);break;
                case 6: scanf("%f", &ttrj); break;
                case 7: scanf("%d", &trj_num); break;
                case 8: yy = 1; break;
                default : exit(0);
            }
        }
        make_ref(trj_idx);
        AMP1_ready();
        AMP2_ready();
        AMP3_ready();
        setting_ST_PD();
```

```
        for (n = 0; n < trj_num; n++) {
                ADAPTATION   /* overall adaptation rate must be 50 msec */
                refST_update(n);
                onepoint_moveST();
        }
        MHC1_active();
        MHC2_active();
        MHC3_active();
        printf("end of path tracking operation\n");
}

./* self-tuning scheme implemented on ECS, file : stcon2.h */

/* function prototypes */

void lowlevel_setup_ST(void);
void com1_com2_ST(void);
void do_trans_ST(void);
void trans_T_nominal(void);
void trans_K(void);
void receiv_get_A_ST(void);
void receiv_get_K(void);
void refST_update(int num);
void get_A_matrix_ST(void);
void get_Phi_matrix(void);
void get_K_matrix(void);
void parameter_collect_ST(void);
void data_write_ST(void);
void compute_T_nominal(void);
void controller_ST(void);

void lowlevel_setup_ST(void)
{
        outportb(0x21, 0x00);     /* master controller: enable all */
        outportb(0xA1, 0x0C);     /* slave controller: disable IRQ10, 11 */
        outportb(0x70, 0x80);     /* off NMI */
        outportb(SDI_U21CW, CONTROL_WORD); /* setup transmission mechanism */
        DISABLE_OUT
        ENABLE_IN
        printf("setup BIOS-level parameters for Self-tunning controller\n");
}

void com1_com2_ST(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];

        switch (char_index) {
                case 31: DISABLE_IN receiv_get_A_ST(); compute_T_nominal();
                        do_trans_ST(); TTRAJ_HIGH break;
                case 63:
                case 119: DISABLE_IN receiv_get_K(); get_K_matrix();
                        do_trans_ST(); break;
                default: trans_charactor[char_index] = inportb(SDI_U21PA);
                        char_index++;
```

```
                }
}

void do_trans_ST(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        int i;

      for (i = 0; i < 100; i++); /* i = 100 OK! */
        switch (char_index) {
                case 32: ENABLE_OUT trans_T_nominal(); break;
                case 39: DISABLE_OUT ENABLE_IN
                         outportb(SDI_U21PB, trans_charactor[char_index]);
                         char_index++;break;
                case 64: ENABLE_OUT trans_K(); break;
                case 95: DISABLE_OUT ENABLE_IN
                 outportb(SDI_U21PB, trans_charactor[char_index]);
                         char_index++;break;
                case 120: ENABLE_OUT trans_K(); break;
                case 151: DISABLE_OUT ENABLE_IN  TTRAJ_LOW
                  outportb(SDI_U21PB, trans_charactor[char_index]);
                          char_index = 0; break;
                default: outportb(SDI_U21PB, trans_charactor[char_index]);
                         char_index++;
        }
}

void trans_T_nominal(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float T_nominal[NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = T_nominal;
        for (i = 0; i < 8; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(SDI_U21PB, trans_charactor[char_index]);
        char_index++;
}

void trans_K(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float K[NUMJOINTS][2*NUMJOINTS];
        unsigned char *iptr;
        int i;

        iptr = K;
        for (i = 0; i < 32; i++)
                trans_charactor[char_index+i] = *(iptr+i);
        outportb(SDI_U21PB, trans_charactor[char_index]);
```

```
        char_index++;
}

void receiv_get_A_ST(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float ang_rad[NUMJOINTS];
        extern float angrate_rad[NUMJOINTS];
        extern float ang2rate_rad[NUMJOINTS];
        extern float torque_msr[NUMJOINTS];
        unsigned char *iptr1, *iptr2, *iptr3, *iptr4;
        int i;

        trans_charactor[char_index] = inportb(SDI_U21PA);
        iptr1 = torque_msr;
        iptr2 = ang_rad;
        iptr3 = angrate_rad;
        iptr4 = ang2rate_rad;
        for (i = 0; i < 8; i++) {
        *(iptr1+i) = trans_charactor[char_index-31+i];
                *(iptr2+i) = trans_charactor[char_index-23+i];
                *(iptr3+i) = trans_charactor[char_index-15+i];
                *(iptr4+i) = trans_charactor[char_index-7+i];
        }
        char_index++;
}

void receiv_get_K(void)
{
        extern int char_index;
        extern unsigned char trans_charactor[NUMTRANS];
        extern float torque_msr[NUMJOINTS];
        extern float x_diff[2*NUMJOINTS];
        unsigned char *iptr1, *iptr2;
        int i;

        trans_charactor[char_index] = inportb(SDI_U21PA);
        iptr1 = torque_msr;
        iptr2 = x_diff;
        for (i = 0; i < 8; i++) {
                *(iptr1+i) = trans_charactor[char_index-23+i];
                *(iptr2+i) = trans_charactor[char_index-15+i];
                *(iptr2+8+i) = trans_charactor[char_index-7+i];
        }
        char_index++;
}

void refST_update(int num)
{
        extern float z_ajcref[4*NUMJOINTS+1];
        extern float data_1[NUMPOINTS];
        extern float data_3[NUMPOINTS];
        extern float data_7[NUMPOINTS];
        extern float data_9[NUMPOINTS];
```

```
        extern float data_13[NUMPOINTS];
        extern float data_15[NUMPOINTS];

        z_ajcref[0] = data_13[num];
        z_ajcref[1] = data_15[num];
        z_ajcref[2] = data_7[num];
        z_ajcref[3] = data_9[num];
        z_ajcref[4] = sin(data_1[num]);
        z_ajcref[5] = cos(data_1[num]);
        z_ajcref[6] = sin(data_3[num]);
        z_ajcref[7] = cos(data_3[num]);
        z_ajcref[8] = 1.00;         /* by definition */
}

void get_A_matrix_ST(void) /* execution time 12 msec */
{
        extern float ang_rad[NUMJOINTS];
        extern float angrate_rad[NUMJOINTS];
        extern float ang2rate_rad[NUMJOINTS];
        extern float torque_msr[NUMJOINTS];
        extern float A_matrix[NUMJOINTS][4*NUMJOINTS+1];
        float p_A[4*NUMJOINTS+1][4*NUMJOINTS+1];
        float z_ajc[4*NUMJOINTS+1];
        float T[NUMJOINTS];
        float k[4*NUMJOINTS+1];
        float s[NUMJOINTS];
        float tmp1[4*NUMJOINTS+1];
        float tmp2, tmp3;
        unsigned int i, j, t;

        /* setup input variables */
        z_ajc[0] = ang2rate_rad[0];
        z_ajc[1] = ang2rate_rad[1];
        z_ajc[2] = angrate_rad[0];
        z_ajc[3] = angrate_rad[1];
        z_ajc[4] = sin(ang_rad[0]);
        z_ajc[5] = cos(ang_rad[0]);
        z_ajc[6] = sin(ang_rad[1]);
        z_ajc[7] = cos(ang_rad[1]);
        z_ajc[8] = 1;
        T[0] = torque_msr[0];
        T[1] = torque_msr[1];
        /* recursive least square identification */
        for (i = 0; i < (4*NUMJOINTS+1); i++)
                for (j = 0; j < (4*NUMJOINTS+1); j++)
                        p_A[i][j] = 1000000.00;
        for (t = 0; t < 4; t++) {  /* 4 times iteration */
                for (i = 0; i < (4*NUMJOINTS+1); i++) {
                        tmp1[i] = 0.00;
                        for (j = 0; j < (4*NUMJOINTS+1); j++)
                                tmp1[i] += p_A[i][j] * z_ajc[j];
                }
                tmp2 = 0.00;
                for (i = 0; i < (4*NUMJOINTS+1); i++)
                        tmp2 += z_ajc[i] * tmp1[i];
```

```
                tmp2 += 1.00;
                for (i = 0; i < (4*NUMJOINTS+1); i++)
                        k[i] = tmp1[i]/tmp2;
                for (i = 0; i < (4*NUMJOINTS+1); i++)
                        for (j = 0; j < (4*NUMJOINTS+1); j++) {
                                if (i == j)
                                        p_A[i][j] = 1 - k[i] * z_ajc[j];
                                else
                                        p_A[i][j] = - k[i] * z_ajc[j];
                        }
                for (i = 0; i < NUMJOINTS; i++) {
                        tmp3 = 0.00;
                        for (j = 0; j < (4*NUMJOINTS+1); j++)
                                tmp3 += z_ajc[j] * A_matrix[i][j];
                        s[i] = T[i] - tmp3;
                        for (j = 0; j < (4*NUMJOINTS+1); j++)
                                A_matrix[i][j] += k[j] * s[i];
                }
        }
}

void get_Phi_matrix(void) /* execution time 8 msec */
{
        extern float torque_msr[NUMJOINTS];
        extern float x_diff[2*NUMJOINTS];
        extern float T_nominal[NUMJOINTS];
        extern float T_diffmeasure[NUMJOINTS];
        extern float Phi[2*NUMJOINTS][3*NUMJOINTS];
        float p_Phi[3*NUMJOINTS][3*NUMJOINTS];
        float z[3*NUMJOINTS];
        float w[2*NUMJOINTS];
        float k[3*NUMJOINTS];
        float s[2*NUMJOINTS];
        float tmp1[3*NUMJOINTS];
        float tmp2, tmp3;
        unsigned int i, j, t;

        /* setup input variables */
        /* calculate differential torques */
        T_diffmeasure[0] = torque_msr[0] - T_nominal[0];
        T_diffmeasure[1] = torque_msr[1] - T_nominal[1];
        z[0] = x_diff[0];
        z[1] = x_diff[1];
        z[2] = x_diff[2];
        z[3] = x_diff[3];
        z[4] = T_diffmeasure[0];
        z[5] = T_diffmeasure[1];
        for (i = 0; i < (2*NUMJOINTS); i++)
                w[i] = 0.5 * x_diff[i];
        /* recursive least square identification */
        for (i = 0; i < (3*NUMJOINTS); i++)
                for (j = 0; j < (3*NUMJOINTS); j++)
                        p_Phi[i][j] = 1000000.00;
        for (t = 0; t < 5; t++) {
                for (i = 0; i < (3*NUMJOINTS); i++) {
```

```
                        tmp1[i] = 0.00;
                        for (j = 0; j < (3*NUMJOINTS); j++)
                                tmp1[i] += p_Phi[i][j] * z[j];
                }
                tmp2 = 0.00;
                for (i = 0; i < (3*NUMJOINTS); i++)
                        tmp2 += z[i] * tmp1[i];
                tmp2 += 1.00;
                for (i = 0; i < (3*NUMJOINTS); i++)
                        k[i] = tmp1[i]/tmp2;
                for (i = 0; i < (3*NUMJOINTS); i++)
                        for (j = 0; j < (3*NUMJOINTS); j++) {
                                if (i == j)
                                        p_Phi[i][j] = 1 - k[i] * z[j];
                                else
                                        p_Phi[i][j] = - k[i] * z[j];
                        }
                for (i = 0; i < (2*NUMJOINTS); i++) {
                        tmp3 = 0.00;
                        for (j = 0; j < (3*NUMJOINTS); j++)
                                tmp3 += z[j] * Phi[i][j];
                        s[i] = w[i] - tmp3;
                        for (j = 0; j < (3*NUMJOINTS); j++)
                                Phi[i][j] += k[j] * s[i];
                }
        }
}

void get_K_matrix(void)
{
        extern float K[NUMJOINTS][2*NUMJOINTS];
        extern float Phi[2*NUMJOINTS][3*NUMJOINTS];
        float tmp1[NUMJOINTS][2*NUMJOINTS];
        float tmp2[NUMJOINTS][NUMJOINTS];
        float tmp3;
        float tmp4[NUMJOINTS][NUMJOINTS];
        float tmp5[NUMJOINTS][2*NUMJOINTS];
        float Q[4];
        float R[2];
        int i, j, k;

        get_Phi_matrix(); /* get linearized perturbation model */
        Q[0] = Q[1] = Q[2] = Q[3] = 1.00; /* tracking eror weighting */
        R[0] = R[1] = 1.00;       /* control force weighting */
        /* compute BQ as tmp1, [2*4] */
        for (i = 0; i < 2*NUMJOINTS; i++) {
                tmp1[0][i] = Phi[i][4] * Q[i];
                tmp1[1][i] = Phi[i][5] * Q[i];
        }
        /* compute R + BQB as tmp2, [2*2] */
        tmp2[0][0] = R[0] + tmp1[0][0]*Phi[0][4] + tmp1[0][1]*Phi[1][4]
                    + tmp1[0][2]*Phi[2][4] + tmp1[0][3]*Phi[3][4];
        tmp2[0][1] = tmp1[0][0]*Phi[0][5] + tmp1[0][1]*Phi[1][5]
                    + tmp1[0][2]*Phi[2][5] + tmp1[0][3]*Phi[3][5];
        tmp2[1][0] = tmp1[1][0]*Phi[0][4] + tmp1[1][1]*Phi[1][4]
```

```
                + tmp1[1][2]*Phi[2][4] + tmp1[1][3]*Phi[3][4];
        tmp2[1][1] = R[1] + tmp1[1][0]*Phi[0][5] + tmp1[1][1]*Phi[1][5]
                + tmp1[1][2]*Phi[2][5] + tmp1[1][3]*Phi[3][5];
        /* compute det of (R + BQB) as tmp3 */
        tmp3 = tmp2[0][0]*tmp2[1][1] - tmp2[0][1]*tmp2[1][0];
        /* compute inverse of (R + BQB) as tmp4  */
        tmp4[0][0] = tmp2[1][1]/tmp3;
        tmp4[0][1] = -tmp2[0][1]/tmp3;
        tmp4[1][0] = -tmp2[1][0]/tmp3;
        tmp4[1][1] = tmp2[0][0]/tmp3;
        /* compute inv(R + BQB)BQ as tmp5 */
        for (i = 0; i < 2*NUMJOINTS; i++) {
                tmp5[0][i] = tmp4[0][0]*tmp1[0][i]
                        + tmp4[0][1]*tmp1[1][i];
                tmp5[1][i] = tmp4[1][0]*tmp1[0][i]
                        + tmp4[1][1]*tmp1[1][i];
        }
        /* compute optimal gain matrix, K */
        for (i = 0; i < NUMJOINTS; i++)
                for (j = 0; j < 2*NUMJOINTS; j++) {
                        K[i][j] = 0.00;
                        for (k = 0; k < 2*NUMJOINTS; k++)
                                K[i][j] += tmp5[i][k]*Phi[k][j];
                }
}

void parameter_collect_ST(void)
{
        extern float data_5[NUMPOINTS];
        extern float data_6[NUMPOINTS];
        extern float data_11[NUMPOINTS];
        extern float data_12[NUMPOINTS];
        extern float data_17[NUMPOINTS];
        extern float data_18[NUMPOINTS];
        extern float data_19[NUMPOINTS];
        extern float data_20[NUMPOINTS];
        extern float data_21[NUMPOINTS];
        extern float data_22[NUMPOINTS];
        extern float data_23[NUMPOINTS];
        extern float data_24[NUMPOINTS];
        extern float data_25[NUMPOINTS];
        extern float data_26[NUMPOINTS];
    extern float A_matrix[NUMJOINTS][4*NUMJOINTS+1];
        extern float T_nominal[NUMJOINTS];
    extern float K[NUMJOINTS][2*NUMJOINTS];

        data_5[ref_pnt] = A_matrix[0][0];
        data_6[ref_pnt] = A_matrix[0][1];
        data_11[ref_pnt] = A_matrix[1][0];
        data_12[ref_pnt] = A_matrix[1][1];
        data_17[ref_pnt] = T_nominal[0];
        data_18[ref_pnt] = T_nominal[1];
        data_19[ref_pnt] = K[0][0];
        data_20[ref_pnt] = K[0][1];
        data_21[ref_pnt] = K[0][2];
```

```
        data_22[ref_pnt] = K[0][3];
        data_23[ref_pnt] = K[1][0];
        data_24[ref_pnt] = K[1][1];
        data_25[ref_pnt] = K[1][2];
        data_26[ref_pnt] = K[1][3];
}

void data_write_ST(void)
{
        FILE *stream;
        extern float data_5[NUMPOINTS];
        extern float data_6[NUMPOINTS];
        extern float data_11[NUMPOINTS];
        extern float data_12[NUMPOINTS];
        extern float data_17[NUMPOINTS];
        extern float data_18[NUMPOINTS];
      extern float data_19[NUMPOINTS];
        extern float data_20[NUMPOINTS];
        extern float data_21[NUMPOINTS];
        extern float data_22[NUMPOINTS];
        extern float data_23[NUMPOINTS];
        extern float data_24[NUMPOINTS];
        extern float data_25[NUMPOINTS];
        extern float data_26[NUMPOINTS];
        int i;

        stream = fopen("b:\\st1.csv", "w");
        for (i = 0; i < NUMPOINTS; i++)
        fprintf(stream, "%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_5[i], data_6[i], data_11[i], data_12[i], data_17[i],
        data_18[i]);
        fclose(stream);
        stream = fopen("b:\\st2.csv", "w");
        for (i = 0; i < NUMPOINTS; i++)
        fprintf(stream,
"%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f,%#6.6f\n",
        time_point[i], data_19[i], data_20[i], data_21[i], data_22[i], data_23[i],
        data_24[i], data_25[i], data_26[i]);
        fclose(stream);
}

void compute_T_nominal(void)
{
        extern float T_nominal[NUMJOINTS];
        extern float z_ajcref[4*NUMJOINTS+1];
        extern float A_matrix[NUMJOINTS][4*NUMJOINTS+1];
        int i, j;

        get_A_matrix_ST();
        refST_update(ref_pnt);
        for (i = 0; i < NUMJOINTS; i++) {
                T_nominal[i] = 0.00;
                for (j = 0; j < (4*NUMJOINTS+1); j++)
                        T_nominal[i] += A_matrix[i][j] * z_ajcref[j];
        }
```

```
        parameter_collect_ST();
        ref_pnt++;
}

void controller_ST(void)
{
        extern float inp_amp[NUMJOINTS];
        extern float inp_frq[NUMJOINTS];
        extern float ttrj;
        extern int trj_num;
        int trj_idx[NUMJOINTS] = {4, 4};
        int yy = 0, key;
        long i, n;

        printf("self-tuning algorithms\n");
        while (yy == 0) {
                printf("1 => select standard input (4 4)\n");
                printf("2 => enter input amplitude (1.0 0.5)\n");
                printf("3 => enter input frequency (0.5 0.5)\n");
                printf("4 => enter trajectory period (0.050)\n");
                printf("5 => number of trajectory points (500)\n");
                printf("6 => inherent algorithms ready\n");
                scanf("%d", &key);
                switch (key) {
                        case 1: scanf("%d %d", trj_idx, trj_idx+1); break;
                        case 2: scanf("%f %f", inp_amp, inp_amp+1); break;
                        case 3: scanf("%f %f", inp_frq, inp_frq+1); break;
                        case 4: scanf("%f", &ttrj); break;
                        case 5: scanf("%d", &trj_num); break;
                        case 6: yy = 1; break;
                        default : exit(0);
                }
        }
        ref_pnt = 0;
        n = 0;
        make_ref(trj_idx);
        lowlevel_setup_ST();
        for (i = 0; i < 10000000; i++) {
                if ((0x08 & inportb(0xA0)) == 0x08) {
                        do_trans_ST();
                        n++;
                }
                if ((0x04 & inportb(0xA0)) == 0x04) {
                        com1_com2_ST();
                        n++;
                }
                if (n == 75500) i = 10000000;
        }
        printf("number of talking interrupts = %ld\n", n);
}
```

ABOUT THE AUTHOR

Mr. Pravit Phongsopa was born in Chantaburi province, Thailand, on January 14, 1967. He received the bachelor degree in mechanical engineering from Chulalongkorn university, Bangkok, Thailand, in 1987. From 1987 to present, He has worked in his own business company, Vilasinee Co., Ltd., Bangkok, Thailand.