Windows NT Workstation, the VMEbus, and Real-Time Control

Warren D. Carew and Karl J. Prince

With advances in computing power, the design and simulation of complex control algorithms has become relatively easy. However, the application of these algorithms as robust, real-time control systems is a nontrivial exercise in computer systems engineering, requiring careful choice of hardware and software platforms. Flotation processes are important in the mining industries and because of their multivariable nature, require a more advanced approach for the control systems. The level control used in the flotation process is not a trivial problem and needs to be addressed so as to optimize the entire multivariable system.

Windows NT® Workstation is becoming a more prevalent choice of a robust operating system, providing features such as multi-tasking, a Graphical User Interface (GUI), and hardware portability. This operating system was evaluated as a platform for the realization of a real-time control system. The personal computer (PC), though popular, is not suitable for use as industrial specification equipment; an industrial hardware solution must be sought. Over the last decade, the VMEbus has emerged as an industrial standard. These computer systems are designed to withstand harsh industrial environments, providing robust, stable solutions for control problems. The system assessed in this article is a VME computer based on the Intel processor, manufactured by OR Industrial Computers running the Windows NT Workstation operating system.

This article demonstrates the successful application of the Windows NT-VMEbus computer system to a practical control application. The generalized predictive control (GPC) scheme, which has proven popular in industry, is applied to a multivariable laboratory flotation plant.

Introduction

Flotation processes are important in the mining and minerals extraction industries, serving to separate the valuable minerals from the less valuable materials. Usually it is the interaction between different phases of the flotation process that leads to the complications creating a multivariable system. Because of this complexity it is necessary to use control systems beyond simple designs.

The level control of a flotation process is an especially complicated and difficult exercise [1]. The simple PI controllers utilized so often in industry are not suited for level control without taking into account the entire multivariable nature of the system. Ideally a more advanced control system needs to be employed for an optimized approach.

Expensive computer systems of the past limited the generation of computer control algorithms and simulators to but a few specialized control centers. Today, computing power is inexpensive and freely available in the form of higher-end workstations down to desktop personal computer (PC) systems. Writing complex code has become a daily and common exercise. While control knowledge is increasing, sight is perhaps being lost of the application of the theory and the implications of implementing an overall robust control system.

On the industrial plant the computing power is not the only priority. Almost of greater concern is the reliability of the control system hardware. To meet the needs of industrial computing, manufacturers have developed a number of industrial-grade computers of which the VMEbus is one of the emerging standards. Although the cost of industrial computer hardware is high in comparison to the desktop machines, the savings in terms of reduced downtime and improved plant performance by virtue of the more advanced control system can result in increased profits for the plant.

Having an industrial standard hardware platform without robust and reliable software would be ineffectual. An operating system that handles all error conditions in a predefined way is essential to guarantee system stability. While many operating systems are available, cost and accessibility are important. The Microsoft Windows platforms have proved popular and the Windows NT Workstation is a suitable solution.

An adaptive control algorithm, of some complexity, is the Generalized Predictive Control (GPC) algorithm by Clarke et al. [2]. A study of the performance of GPC on an industrial multivariable problem of the flotation process was done [3]. In this article, the technical application problems and the results of applying four GPC controllers to a 4x4 laboratory flotation plant are presented.
The purpose of this article is to give insights into the technical issues that arise when control algorithms are implemented on real plants. In particular, this article demonstrates the features of Windows NT Workstation and a VMEbus industrial computer system that are necessary to realize a real-time multivariable GPC control system on a flotation plant.

The Flotation Plant

The flotation process is central to the mineral extraction industry. Essentially, it concerns the separation of valuable solid raw materials from a slurry (a mixture of water and particles). The minerals are removed as a frothy concentrate which is induced by air bubbles introduced into the slurry.

In order to demonstrate the effectiveness of a GPC control system in stabilizing and controlling a flotation plant, it was employed on a laboratory flotation plant. The flotation rig is large enough that noisy industrial conditions are generated and represents a fair test of the control system. The control problem is a simple level control exercise but is complicated by the fact that the entire process is multivariable (i.e., there are interactions between plant variables). A photograph of the flotation rig is shown in Fig. 1.

![Fig. 1. The flotation plant.](image)

The four tanks in the circuit emulate the rougher, scavenger, cleaner, and re-cleaner of a flotation process. The transfer function matrix model of the system is:

\[
\begin{bmatrix}
    y_1 \\
    y_2 \\
    y_3 \\
    y_4
\end{bmatrix} = \begin{bmatrix}
    g_{11} & g_{12} & g_{13} & g_{14} \\
    g_{21} & g_{22} & g_{23} & g_{24} \\
    g_{31} & g_{32} & g_{33} & g_{34} \\
    g_{41} & g_{42} & g_{43} & g_{44}
\end{bmatrix} \begin{bmatrix}
    u_1 \\
    u_2 \\
    u_3 \\
    u_4
\end{bmatrix}
\]

where

- \( y_1 = \text{Rougher Level} \)
- \( y_2 = \text{Scavenger Level} \)
- \( y_3 = \text{Cleaner Level} \)
- \( y_4 = \text{Recleaner Level} \)

and \( g_{ij} \) are estimated plant models of the form

\[
g_a = \begin{bmatrix} V' \end{bmatrix} \begin{bmatrix} b_2 + b_3 q^{-1} \\ 1 + a q^{-1} \end{bmatrix} g_v = \frac{b_2 + b_3 q^{-1}}{1 + a q^{-1}}
\]

for \( i \neq j \), representing interaction terms.

Capacitive level probes are used in each tank and return a voltage signal proportional to the level. The product dynamics of the flotation plant are controlled via the pneumatic valves installed on the tailing outputs of each of the flotation cells. Each valve has a current to pressure converter to convert the electrical control signal into valve action.

The Generalized Predictive Control Algorithm

The generalized predictive control (GPC) method was developed by Clarke, Mohtadi, and Tuffs [2] in the mid-'80s. It is one of a number of adaptive predictive control algorithms that has proven popular. It has been applied to various control applications [4, 5, 6]. The basic GPC algorithm is outlined in Table 1.

The GPC strategy predicts the process outputs based on future sets of control signals and minimizes some cost function in order to select the appropriate control action for implementation.

A simple example demonstrating the implementation of the GPC algorithm is given in Appendix A. This example also shows that the algorithm is fairly mathematically complex and would require a substantial amount of computing power for real-time implementation.

Since the flotation plant is a 4x4 system and the GPC algorithm used is essentially a single-input single-output method, a diagonal control matrix of four GPC controllers is applied to the plant. While this may not be an optimum solution, it would demonstrate the adaptability of GPC in accommodating for the changes in plant model. Note that this method makes no attempt at eliminating all of the interaction between plant inputs and outputs [3].

**Implementation Issues**

By definition from the uninet comp.real-time real-time FAQ:

A real-time system is one in which the correctness of the computation not only depends on the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

Thus a real-time system requires that all I/O operations and calculations take place at the correct time instance. This imposes the following requirements:

- Sufficient processing power (the factors being CPU and RAM)
- Sufficiently high-speed I/O peripherals (A/D and D/A hardware)

In industrial applications, where every minute is money, reliability is essential and so is robustness—a plant nothing is guaranteed. The system hardware must be able to endure harsh environmental conditions, and the control system must be robust in terms of noise and plant model variations. Extending the list of requirements:

- Reliability (the factors here are hardware endurability and operating system stability)

<table>
<thead>
<tr>
<th>Table 1. The Basic GPC Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Consider a particular time ( t ) as the present.</td>
</tr>
<tr>
<td>• A prediction model is generated based on the plant state at time ( t ).</td>
</tr>
<tr>
<td>• For a given future setpoint, the process output is calculated, using the prediction model, over a prediction horizon. The prediction horizon being the range of future times used for predictions.</td>
</tr>
<tr>
<td>• Various sets of control actions are suggested for the prediction but only the best strategy (i.e., one that minimizes some appropriate cost function of output errors and control actions) will be selected.</td>
</tr>
<tr>
<td>• The chosen set of control actions is applied at time ( t ).</td>
</tr>
<tr>
<td>• The procedure moves on to the next time instant, ( t + 1 ), and is repeated.</td>
</tr>
</tbody>
</table>

IEEE Control Systems
Robustness (control algorithm and the predictable response of the operating system under all conditions)

The combination of Windows NT Workstation and VMEbus Industrial Computer system give a sound platform for the implementation of an industrial control system. In the following two sections, the critical features of Windows NT Workstation and the VMEbus Industrial Computer are discussed.

Windows NT Workstation

Overview. Windows NT Workstation, or just NT, is a 32-bit multi-tasking multi-processor operating system which has a graphical user interface (GUI). It is designed to be robust, responding predictably to both hardware and software error conditions. One unique feature of the operating system is its portability. Windows NT has been designed to be run on different hardware platforms, such as x86, MIPS, and ALPHA. User-written applications need only be recompiled for the particular platform in order to be ported.

Windows NT is a reliable and robust system, having "exception checking" as an integral part of the operating system as opposed to leaving it to the application writer. A modular approach has been adopted—in concept as well as in coding. This has resulted in NT being a layered and server/client-based operating system. The lowest level, known as the Hardware Abstraction Layer (HAL), isolates all the platform-specific code, presenting the kernel and executive with the same virtual machine irrespective of the underlying hardware. The kernel at the heart of NT is a dispatcher; i.e., when events (interrupts, thread scheduling, time-outs, message posting, etc.) occur within the system, it dispatches a call to the relevant routines to deal with the situation. Device drivers extend the operating system by telling it how to deal with specific devices (virtual or real) when events occur. These are the only procedures that have a privilege level high enough to directly access the hardware. By using the HAL interface, platform and processor dependency is avoided.

The Windows NT kernel and the device drivers form the basis of the operating system executive, the Windows NT "core" (see Fig. 2). The executive controls all the allocation and use of system resources. This includes allocating processing time—NT employs pre-emptive multi-tasking to provide time slices for each application. This is significantly different from the way Windows 3.1 multi-tasks, since it uses cooperative multi-tasking, i.e., the process would tell the operating system when it was done and another process could continue execution.

The executive exists within its own memory space, referred to as system space, and all its parts execute at privileged processor level ring 0. This implementation of the executive ensures that user applications are not able to interfere with the operating system and that the operating system has available the full processing power of the CPU available.

Within NT resources are dealt with as objects, each carrying its own data and operational code, being independent of other code. This approach has facilitated easy maintenance—as long as the interface remains fixed, code can be improved without adversely affecting other procedures. Since the operating system controls the creation of objects and therefore the resources, it is able to isolate user applications and prevent conflicts from occurring. Security has also been ensured by the calling of the security system on the creation of any object. All applications—or, rather, processes—are assigned their own memory space and set of resources, thus existing in their own virtual machine managed by the Virtual Machine Manager (VMM) in the executive. This memory space is termed the linear memory, as opposed to the physical system memory. The operating system and its critical code and data are put in the upper 2 gigabytes (GB) of the virtual memory and are inaccessible to user-level applications in the lower 2GB, thereby preventing the user application from stalling or "hanging" the operating system.

User applications can and must still interact with the NT executive, and this is
Table 2. Reliability and Robustness

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception handing</td>
<td>Built into all layers of the operating system, assuring that all error</td>
</tr>
<tr>
<td></td>
<td>conditions are handled in a predefined manner.</td>
</tr>
<tr>
<td>Layering of the operating system</td>
<td>The layers have fixed interfacing protocols and are mutually independent.</td>
</tr>
<tr>
<td>Pre-emptive multi-tasking</td>
<td>If an application locks up, the executive will still move on to the next</td>
</tr>
<tr>
<td></td>
<td>process.</td>
</tr>
<tr>
<td>Isolation of applications to their own virtual</td>
<td>Applications are not able to crash the operating system.</td>
</tr>
<tr>
<td>machines</td>
<td></td>
</tr>
</tbody>
</table>

Achieved through a server/client relationship. The currently executing process requests a service (through the Local Procedure Call, LPC) from the system (server) and which on completion signals the calling process (client). This allows asynchronous I/O since the client does not need to wait for the service to complete the task before continuing on its own, but this brings in the added complexity of synchronization.

The HAL, device drivers, kernel, and executive form the engine of the NT operating system. Using the facilities and concepts outlined here, NT is able to grow, meeting the needs of the computer platform and applications that are to run on it. This is achieved by adding daemons, services, environments, and applications to the NT engine. For the purposes of this article it is not necessary to enter into these concepts in any depth; for completeness, a brief description on each will be given:

- Daemons and services are user-level background applications that typically handle functions such as printer spoolers, TCP/IP sockets, etc.
- The native environment is the Win32 application programming interface (API)—NT can allow other environments, such as OS/2, POSIX, DOS, Windows 3.1, and Windows 95 to run on the NT engine.
- Applications are the user programs; the structure of these will be discussed in the next section.

Fig. 3 shows the complete NT structure, grouping the extended components, the NT core, and user applications.

Reliability and Robustness. Given the structure of NT described above, the inherent robustness and reliability of NT can be ascribed to the properties in Table 2.

Real-Time and Multi-Tasking Capabilities. In order to understand the real-time capability of NT, it is necessary to take a closer look at multi-tasking and related operations that allow the implementation of real-time systems.

To understand how applications are able to "multi-task" it is necessary to understand how programs run under Windows NT [7]. Once an application is invoked, this instance of the running program is known as a process. No code is executed at this time since processes really only make available address space (and at times other resources) for code and data. In order for the code to be executed, a process must contain one or more threads.

If a process has many threads, they are seen to be able to execute code concurrently in the process—this is accomplished by the Windows NT executive, which provides each thread with time quantums. (See Fig. 4.) It is possible to set the priority of threads so that a time-critical thread, with highest priority, is executed first and most frequently.

As mentioned earlier, NT is also a multi-processor system and employs symmetric multiprocessing. This allows the operating system to schedule tasks (execution of threads), including operating system functions, evenly across the
VMEbus conforms to the 19" standard, which makes fitting components and hardware upgrading easy.

The flexibility of the VMEbus allows it to be used as a development platform as well as for the actual implementation of the control system. This not only saves cost, but also decreases the risk of failure due to porting from the development platform to the implementation platform.

To summarize: the advantage of using the VMEbus is that there are very few concerns after having chosen the hardware for the applications environmental specification. The VMEbus hardware has a modular design based on a common back plane. The choice of CPU and I/O configuration is left entirely up to the engineer’s needs. The authors familiarity with Intel processors and PCs influenced the choice of a 486DX2-50 CPU module with 16MB of RAM and a 540MB hard drive. The VMEbus back plane has a number of expansion slots for I/O purposes (12 in the above system).

For harsh environmental conditions, off-the-shelf solutions are available to the specifications shown in Table 3.

<table>
<thead>
<tr>
<th>Temperature Range</th>
<th>-40°C / + 85°C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact and Rugged Euro-Card Sizes</td>
<td>19&quot; rack</td>
</tr>
<tr>
<td>High Shock and Vibration Immunity</td>
<td>up to 40G</td>
</tr>
<tr>
<td>Conduction Cooling</td>
<td>IEEE 1101.2</td>
</tr>
<tr>
<td>Moisture Protection</td>
<td>MIL-1-46058</td>
</tr>
</tbody>
</table>

*Reference—Embedded Single Board PCs OR also see Motorola VME Module Selection Guide

| August 1997 | 81 |
and the second is the implementation of the mathematics of the control algorithm and user interface to this hardware. The solution to part one is found in writing a device driver and to part two in the development of a user interface for the control algorithm. Typically its functions are:

- Registering hardware resources (IoReportResourceUsage)
- Creates the DriverObject (IoCreateDevice)
- Specifies the functions to be called when the device driver is accessed farther by user applications or the operating system (Device IoControl and DriverUnload functions)
- Creates a symbolic link (IoCreateSymbolicLink)
- The specific VMEbus driver functions implemented here are:
  - Initializing the VMEbus interface
  - Maps the VMEbus window to linear address space

### Table 4. The Device Driver

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject, IN PUNICODE RegistryPath)</strong></td>
<td>This function is called by the Operating System at the time of starting the device driver. Typically its functions are:</td>
</tr>
</tbody>
</table>
- Registering hardware resources (IoReportResourceUsage)  
- Creates the DriverObject (IoCreateDevice)  
- Specifies the functions to be called when the device driver is accessed farther by user applications or the operating system (Device IoControl and DriverUnload functions)  
- Creates a symbolic link (IoCreateSymbolicLink)  
- The specific VMEbus driver functions implemented here are:  
  - Initializing the VMEbus interface  
  - Maps the VMEbus window to linear address space |
| **NTSTATUS VMEbusDispatch (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)** | This function is the entry point for the user application calls to the device driver using the DeviceIoControl function. The Irp structure facilitates the selection of minor functions and the transfer of data between the device driver and application. Specifically the VMEbus driver function implement here is returning the pointer to the VMEbus window. |
| **NTSTATUS VMEbusUnload (IN PDRIVER_OBJECT DriverObject)** | This function, specified in the DriverEntry procedure, is called when the device driver is stopped. Typically its functions include:  
- Freeing any resources being used by the device driver  
- Removing entries from registry  
- Deleting the symbolic link and the device object (IoDeleteSymbolicLink and IoDeleteDevice) |

The VMEbus, is constructed globally for the application example. This was due to the relatively long sample time of 1 second required on the flotation plant.

The device driver can be compiled by using the device driver kit (DDK) from Microsoft (available as part of the Microsoft Development Platform) [9]. For ease of using the device driver functions, a standard C++ class can be developed to hide all the details from the user’s application.

### I/O Interface

A single instance of the C++ class, VMEbus, is constructed globally for the application process. All service calls to the device driver with the DeviceIoControl function are made from within this class. This removes concern over the device driver from the application writer. For the VMEbus, the functions that are required are A/D reading and D/A writing functions. The public functions of the VMEbus class are given in Table 5.

It is possible that various threads within the process would make calls to these functions.
Table 5. The Public Functions of the VMEbus Class

VMEbus :: VMEbus()
The class constructor implements the following:
- Creates a handle for the device driver (CreateFile)
- Obtains the pointer to the VMEbus window (DeviceIoControl)
- Initializes the critical sections (InitializeCriticalSection)
- Does the standard ADC and DAC initialization through the memory window

VMEbus :: VMEbusO()

double VMEbus :: ADC (UCHAR channel_)
- Indicates the beginning of a critical section (EnterCriticalSection)
- Accesses the A/D through the memory window
- Indicates the end of the critical section (LeaveCriticalSection)
- Converts the 12-bit number to a voltage value

double VMEbus :: DAC (UCHAR channel_, double value_)
- Indicates the beginning of a critical section (EnterCriticalSection)
- Accesses the A/D through the memory window
- Indicates the end of the critical section (LeaveCriticalSection)
- Converts the 12-bit number to a voltage value

VMEbus :: ~VMEbus ()
The class destructor frees the critical sections (DeleteCriticalSection)

Application Software

The application defines the user interface, generates the real-time threads, and constructs the instance of the I/O interface class, VMEbus. The skeleton of the critical application code is given in Table 6.

Using the Microsoft Foundation Classes (MFC) in Visual C++ facilitates the easy generation of a user interface. The CGPCDoc class, created from the MFC CDocument class, forms the basis for each controller implementation. The user is able to open several documents, or GPC controllers, as separate windows within the application.

Within the CGPCDoc class the real-time thread is generated using CreateThread and the priority set using SetThreadPriority. The thread is associated with the real-time process via a pointer in a dummy thread procedure, RealTimeDummyProc.

The CGPCDoc, and any instances of the same, run within the single user interface thread. The primary purpose of CGPCDoc is to allow the user to configure the specific controller (e.g., the GPC parameters, running time of plant, I/O interface ports, etc.). All these are stored in a common data/status area so that the
Table 6. Critical Application Code

```c
CGPCDoc :: CGPdoc()
{
    // Initialize variables
...

    // Variable test to true to terminate thread execution
    bFinished = FALSE;

    // Create thread ...
    hThread = CreateThread(NULL,0,
        (unsigned long(__stdcall*)(void*)
            &RealTimeDummyProcedure,
        this,0,(LPDWORD)&dwIDThread));

    // Set thread priority ...
    SetThreadPriority(hThread,
        THREAD_PRIORITY_TIME_CRITICAL);

    inline long WINAPI RealTimeDummyProc (CGPCDoc *fpObj)
    {
        // This is a dummy function to call the actual real-time thread
        // This function must be included as a friend function to GPCDoc
        return fpObj->RealTimeProcess();
    }

    void CGPCDoc :: RealTimeProcess()
    {
        // Specify the sample time interval (in milliseconds)
        long sample_time = 1000;
        // Specify the idle time interval (in milliseconds)
        // this is the amount of time before the thread will wake to check
        // if the bStarted flag is set
        long idle_time = 10;
        // While thread still active ...
        while (!bFinished)
        {
            // Start process if flag set in Common Status variables
            if (bStarted) {
                // Initialize variables for new run
                t = 0;
                do {
                    // Get the starting time of this sample instance
                    clockticks = GetTickCount();
                    // Get the current plant output
                    plant_output = VMEbus.DAC (channel);
                    // Calculate the GPC control signal
                    control_signal = GPC(control_signal,plant_output,t);
                    // Output control signal to plant
                    VMEbus.ADC (channel,control_signal);
                    // Post any update screen messages, etc.
                    ...
                    // Sleep until next sample time
                    Sleep (sample_time - (GetTickCount() - clockticks));
                    t = t + sample_time;
                } while ((t < end_time) && (!bFinished));
            } else
                Sleep (idle_time);
            ...
        }
    }

    CGPCDoc :: ~CGPCDoc()
    {
        // Indicate that thread must finish
        bFinished = TRUE;
        // Close thread handle
        CloseHandle (hThread);
        ...
    }
```

real-time thread has access to the information and can adapt the running of the controller to match these settings.

The timing for the real-time thread comes from the use of the Sleep function. In this instance the thread puts itself to sleep for a specified time interval. When the interval expires, the "Executive" continues the processing of the thread. At the highest priority level, this happens immediately after the interval expires.

A 4x4 GPC Diagonal Controller

To control the flotation plant, four GPC controllers are required. In NT, this corresponds to the user opening four controller "documents," thus creating four instances of CGPCDoc and the associated real-time threads.

Each GPC controller is associated with an input/output pair for the specific tank it is to control. Fig. 6 shows the application structure and tank positioning in the system. The separate threads are indicated as well as the I/O interface.

Each of the four documents share one user interface thread. On their creation, they generate a separate real-time thread for the controller they represent. The I/O interface section, which is driven by the real-time controller, deals with the critical sections, the device driver interface, and ultimately the flotation tanks.

GPC Flotation Results

To demonstrate the control system, one of the tank levels on the flotation plant is stepped and the levels of the other tanks are held constant. A typical run is shown in Fig. 7.

Results for step changes the other tank levels is shown in Appendix B. The control system developed using the GPC method is shown to be capable of controlling the tank levels well. While there is still some interaction present, the disturbances via the model changes are eliminated.

An output screen of the program is in Fig. 8. The panels are the four instances of the controllers (CGPCDoc classes): Rougher, Recleaner, Scavenger, and Cleaner.

This run was taken when the plant was in an equilibrium state and the controllers were activated at time \( t = 0 \). The time divisions are at 100-second intervals. Fig. 9 shows a maximized view of the Recleaner Tank with the setpoint being stepped.

IEEE Control Systems
A typical dialog box, in which the GPC parameters can be altered, is included on the display. As is evident, the interface provides for clear and easy observation and editing of control parameters, viewing parameters, etc.

**Conclusions**

The application of GPC to the flotation plant proved successful. The algorithm was sufficiently adaptive to deal with the changes in plant model due to plant interaction. As is evident by the success of the control system, the computing platform has proven adequate.

A necessary realization in the control industry is the fact that real problems need practical solutions. The use of computers in industry requires not only that there be processing power, but that the computer system be rugged and capable of handling the harsh conditions prevalent in industry.

The Microsoft Windows NT operating system is suited to providing a solution for those looking for a robust operating system, yet with an easy-to-use GUI. It has been shown that the system is portable and can be customized, thus allowing any hardware platform to be used.

From the development of this GPC control system, it has become evident that a general-purpose real-time control system environment would be easily implemented. This would allow different control algorithms to be incorporated into a common interfacing package.

**Appendix A**

The complexity of the GPC algorithm is demonstrated by a simple example. The plant assumed is of the following order:

\[ u(t) = \frac{b_2 + b_1 q^{-1}}{1 + a_1 q^{-1}} u(t-1) \]

The output is given by \( y \), the control signal by \( a \). The reference (setpoint) signal is indicated by \( r \). \( q^{-1} \) is the backward shift operator and \( a_i, b_i \) are real constants.

The GPC parameters are selected to be

- \( N_1 = 1: \) min. prediction horizon
- \( N_2 = 3: \) max. prediction horizon
- \( N_u = 3: \) min. control horizon
- \( \lambda = 0.1: \) damping factor

From the recurrence equations:

- \( e_1 = f_{10} \)
- \( E_{2} = \mathcal{E}_{2} + q^{2} e_{2} \quad F_{2} = F_{1} - A \Delta e_{1} \)

Initializing:

- \( E_1 = 1 \quad F_1 = q^{-1}(1 - A \Delta) \quad e_0 = 1 \quad f_{10} = (1 - a_i) \quad f_{11} = a_i \)

Continuing,

- \( E_2 = E_1 + q^{2} e_{1} \quad F_2 = F_1 - A \Delta e_{1} \)
- \( E_3 = E_2 + q^{2} e_{2} \quad F_3 = F_2 - A \Delta e_{1} \)

For the rest of the example we shall choose a specific plant to demonstrate the procedure (for an on-line GPC strategy the plant is identified via a recursive least squares algorithm—RLS).

Let
\[ A(q^{-1}) = 1 - 0.5q^{-1} \]
\[ B(q^{-1}) = 1 - 0.9q^{-1} \]

From the GPC identity: \( 1 = E_j \Delta + q T_j \).

Note that the C polynomial is set to 1 for this example.

\[
\begin{align*}
J_1 &= 1 = (1 - 1.5q^{-1} - 0.5q^{-2}) \\
J_2 &= 2 = (1 + 1.5q^{-1})(1 - 1.5q^{-1} - 0.5q^{-2}) + q^{-2}(1.75 - 0.75q^{-1}) \\
J_3 &= 3 = (1 + 1.5q^{-1} + 1.75q^{-2})(1 - 1.5q^{-1} - 0.5q^{-2}) + q^{-2}(1.875 - 0.875q^{-1})
\end{align*}
\]

Now find the predicted process outputs, splitting them into information known and not known at time \( t \),

\[
\begin{bmatrix}
\dot{y}(t + 1) \\
\dot{y}(t + 2) \\
\dot{y}(t + 3)
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 \\
2.4 & 1 & 0 \\
3.1 & 2.4 & 1
\end{bmatrix}
\begin{bmatrix}
\Delta u(t) \\
\Delta u(t + 1) \\
\Delta u(t + 2)
\end{bmatrix}
+ \begin{bmatrix}
1.5y(t) - 0.5y(t - 1) + 0.9\Delta y(t - 1) \\
1.75y(t) - 0.75y(t - 1) + 1.35\Delta y(t - 1) \\
1.875y(t) - 0.875y(t - 1) + 1.575\Delta y(t - 1)
\end{bmatrix}
\]

In order to find the control law, we now have to find

\[
(G^*G + I\lambda)^{-1}G^*
\]

Using the first row of the above matrix, a setpoint and the previously defined vector \( f \), the GPC control law can be obtained:

\[
\Delta u(t) = [\text{first row of } (G^*G + I\lambda)^{-1}G^*] \begin{bmatrix} f \end{bmatrix}
\]

The control signal then implemented at time \( t \) is:

\[
u(t) = 0.252r(t + 1) + 0.2653r(t + 2) - 0.0691r(t + 3) - 1.1222y(t) + 0.401y(t - 1) - 0.2782u(t - 1) - 0.7218u(t - 2)
\]

Appendix B

For completeness, the typical step responses of the other tanks are presented here. Note that the scales on the graph of the stepped tank and the graphs of the other tanks are different.

References

Fig. 10. Results for cleaner stepped.

Fig. 11. Results for scavenger stepped.
Fig. 12. Results for recleaner step.


