Omuses

a tool for the

Optimization of **mu**lti**s**tag**e** **s**ystems

and

HQP

a solver for sparse nonlinear optimization

Version 1.5

Rüdiger Franke
Dept. of Automation and Systems Engineering
Technical University of Ilmenau
Germany

September 13, 1998

Ohne das Instrumentarium der Informatik ist die Kybernetik denkbehindert, ohne systemtheoretische Konzeptionen ist die Informatik bei der Steuerung zweckbehindert (sichtbehindert).

Karl Reinisch, 1985

[Without the instruments of computer science, cybernetics is hindered in thinking; without system theoretical conceptions, computer science is aim hindered (hindered in viewing) in the case of control.]

Contents

Chapter 1

Omuses

1.1 Introduction

Omuses is a front-end to the large-scale nonlinear optimization solver HQP (see chapter 2). It supports the formulation and the solution of optimization problems for models described by differential and recurrence equations.

Small optimization problems can be formulated using the Omuses problem interface without stages. The multistage formulation is advantageous to the efficient application of the SQP-type solver HQP to large-scale optimization problems, if the problem structure allows it. Often the separation of optimization problems into multiple stages results in a faster solution. This is caused by the better approximation of the Lagrangian Hessians of the nonlinear problem parts in each stage by low rank updates. The idea of partitioned variable metric updates was introduced by Griewank and Toint (1982) for unconstrained minimization and is regarded in the literature up to now, e.g. (Fletcher, 1995).

In our case, the idea is applied to nonlinearly constrained problems using Powell's modified BFGS update for separate diagonal blocks of the Lagrangian Hessian. The cost is that the problem size is blown up. Additional variables and linear junction conditions between successive stages are introduced. But the resulting block-banded sparsity structure allows an efficient treatment. For problems with many free variables, e.g. to approximately describe a continuous control trajectory for a dynamic system, the better sparsity structure might even further speed up the solution. The additional variables often represent system states. In this way the specification and treatment of constraints is simplified, provided that an appropriate optimization solver is available (Arnold, 1987),(Arnold et al., 1994). A sparse interior point algorithm for convex quadratic programming has been developed and implemented in HQP for the efficient treatment of the linear-quadratic subproblems in the nonlinear SQP-iterations (Franke and Arnold, 1997). The algorithm is motivated by Wright (1993).

Omuses supports the sampling of a continuous time horizon and the flexible assignment of the samples to stages. It covers the calculation of exact derivatives and sensitivity equations by using automatic differentiation (Griewank et al., 1996). Standard numerical integration procedures are applied to the solution of differential equations. All stages are assigned to a large, nonlinearly constrained optimization problem. Constraints that result from the junction conditions between successive stages are inserted automatically.

The multistage interface of Omuses is strongly motivated by the separability structure of discrete-time optimal control problems. However, we think that this structure is of quite general use. The particularity of Omuses, compared to currently available general-purpose languages and tools for mathematical programming, e.g. (Fourer et al., 1993), is the support for differential equations. Advanced modeling languages and model compilers for continuous-time systems can be used to simplify the problem specification (Franke, 1997), (Franke and Arnold, 1996). In the context of control systems engineering, the solution method is referred to as multistage control parameterization or direct multiple shooting. The application of Omuses is currently limited to ordinary differential equations (ODEs). Differential-algebraic equation systems (DAEs) are considered in the problem interface. See e.g. (Pantelides et al., 1994) and (Maly and Petzold, 1996) for the extension of the approach to DAEs.

This chapter concentrates on the Omuses problem interface and the discussion of examples. Test runs of the underlying optimization solver HQP (see chapter 2) can also be performed via e-mail problem submission (Franke, 1996), using the low-level standard input format SIF (Conn et al., 1992). This can be useful for comparing HQP with other solvers. Large-scale nonlinear test examples out of several areas of application are available in the CUTE collection. Their free availability has been very helpful for the development of HQP. However, HQP does currently neither exploit the general separability structure expressed in SIF, nor does the CUTE solver interface support this (Bongartz et al., 1995). For more information about available optimization software and testing see e.g. H.D. Mittlemann's guide `http://plato.la.asu.edu/guide.html`.

The experience we have made when treating several optimization problems, practical applications and standard test examples, and the availability of 'productivity' languages and tools like C++, Tcl, and ADOL-C, confirmed us to develop Omuses.

1.2 Problem formulation

1.2.1 Problem without stages

An unstructured nonlinear programming problem in $\mathbf{x} \in \mathbb{R}^{n_x}$ variables can be formulated without use of stages as

$$\begin{aligned} f_0(\mathbf{x}) \;&\rightarrow\; \min_{\mathbf{x}}, \\ f_0 &: \mathbb{R}^{n_x} \mapsto \mathbb{R}^1, \end{aligned} \tag{1.1}$$

subject to the variable bounds

$$\mathbf{x}_\mathrm{l} \;\le\; \mathbf{x} \;\le\; \mathbf{x}_\mathrm{u}, \tag{1.2}$$

and the general constraints

$$\begin{aligned} \mathbf{c}_\mathrm{l} \;\le\; \mathbf{c}(\mathbf{x}) \;&\le\; \mathbf{c}_\mathrm{u}, \\ \mathbf{c} &: \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_c}. \end{aligned} \tag{1.3}$$

Omuses calculates the Jacobians of f_0 and $\mathbf{c}$ for the solution of the problem by HQP.

1.2.2 Multistage problem

The variables $\mathbf{x}^k \in \mathbb{R}^{n_{x,k}}$ and $\mathbf{u}^k \in \mathbb{R}^{n_{u,k}}$ are introduced for multistage problems with $K > 0$ stages $k = 0, 1, \ldots, K-1$. The variables $\mathbf{x}^k$ represent the initial states of each stage, whereas the variables $\mathbf{u}^k$ are additional control parameters. The optimization variables of the unstructured problem (1.1)–(1.3) serve in the multistage case as the final states $\mathbf{x}^K$. The state variables are constrained with the system equations

$$\begin{aligned} \mathbf{x}^{k+1} &= \mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k), \\ \mathbf{f}^k &: \mathbb{R}^{n_{x,k}} \times \mathbb{R}^{n_{u,k}} \mapsto \mathbb{R}^{n_{x,k+1}}. \end{aligned} \tag{1.4}$$

The solution for the variables is underdetermined by (1.4) alone. One way to define a solution, would be to specify the initial states $\mathbf{x}^0$ and the controls $\mathbf{u}^k$. The system equations could then be successively solved for the states $\mathbf{x}^k$, $k > 0$ and $\mathbf{x}^K$. However, the solution of the multistage problem is defined more generally by the mathematical programming problem

$$\begin{aligned} f_0^K(\mathbf{x}^K) \; + \; \sum_k f_0^k(\mathbf{x}^k, \mathbf{u}^k) \; &\rightarrow \; \min_{\mathbf{x}^k, \mathbf{u}^k, \mathbf{x}^K}, \\ f_0^K : \mathbb{R}^{n_{x,K}} \mapsto \mathbb{R}^1, \;\; f_0^k &: \mathbb{R}^{n_{x,k}} \times \mathbb{R}^{n_{u,k}} \mapsto \mathbb{R}^1, \end{aligned} \tag{1.5}$$

subject to the system equations (1.4), the variable bounds

$$\begin{aligned} \mathbf{x}_\mathrm{l}^k \;\le\; \mathbf{x}^k \;\le\; \mathbf{x}_\mathrm{u}^k, \quad \mathbf{x}_\mathrm{l}^K \;\le\; \mathbf{x}^K \;\le\; \mathbf{x}_\mathrm{u}^K, \\ \mathbf{u}_\mathrm{l}^k \;\le\; \mathbf{u}^k \;\le\; \mathbf{u}_\mathrm{u}^k, \end{aligned} \tag{1.6}$$

and the constraints

$$\begin{aligned} \mathbf{c}_\mathrm{l}^k \;\le\; \mathbf{c}^k(\mathbf{x}^k, \mathbf{u}^k) \;\le\; \mathbf{c}_\mathrm{u}^k, \quad \mathbf{c}_\mathrm{l}^K \;\le\; \mathbf{c}^K(\mathbf{x}^K) \;\le\; \mathbf{c}_\mathrm{u}^K, \\ \mathbf{c}^k : \mathbb{R}^{n_{x,k}} \times \mathbb{R}^{n_{u,k}} \mapsto \mathbb{R}^{n_{c,k}}, \; \mathbf{c}^K : \mathbb{R}^{n_{x,K}} \mapsto \mathbb{R}^{n_{c,K}}. \end{aligned} \tag{1.7}$$

Omuses calculates the Jacobians of (1.4), (1.5), and (1.7) in each stage and assigns all stages to a large-scale nonlinear programming problem.

1.2.3 Treatment of differential equations

Often problems of the form (1.4)–(1.7) arise from the discretization of differential equations. The functions $\mathbf{f}^k$ define then the solutions of the differential equations over specified time periods. Their Jacobians are the sensitivity matrices of the solutions with respect to the initial states $\mathbf{x}^k$ and the control parameters $\mathbf{u}^k$.

The time horizon $[t_0, t_f]$ is introduced for the treatment of differential equations by Omuses. It is divided into K communication periods $[t^k, t^{k+1}]$ with the communication time points $t_0 = t^0 < t^1 < \ldots < t^K = t_f$. Each stage of the multistage problem covers one communication period. The differential equations are defined for the continuous-time

state variables $\tilde{\mathbf{x}}^k \in \mathbb{R}^{n_{\tilde{x},k}}, n_{\tilde{x},k} \geq n_{x,k}$. In each stage, the continuous-time final states $\tilde{\mathbf{x}}^k(t^{k+1})$ are calculated by numerical integration of

$$\begin{aligned} \mathbf{F}^k(t, \tilde{\mathbf{x}}^k, \mathbf{u}^k, \dot{\tilde{\mathbf{x}}}^k) &= \mathbf{0}, \quad t \in [t^k, t^{k+1}], \\ \mathbf{F}^k &: \mathbb{R}^1 \times \mathbb{R}^{n_{\tilde{x},k}} \times \mathbb{R}^{n_{u,k}} \times \mathbb{R}^{n_{\tilde{x},k}} \mapsto \mathbb{R}^{n_{\tilde{x},k}}, \end{aligned} \tag{1.8}$$

starting with the initial conditions

$$\begin{aligned} \tilde{\mathbf{x}}^k(t^k) &= \mathbf{I}^k(t^k, \mathbf{x}^k, \mathbf{u}^k), \\ \mathbf{I}^k &: \mathbb{R}^1 \times \mathbb{R}^{n_{x,k}} \times \mathbb{R}^{n_{u,k}} \mapsto \mathbb{R}^{n_{\tilde{x},k}}. \end{aligned} \tag{1.9}$$

Currently only ODEs are treated. The initialization is defined by default as

$$\mathbf{I}^k_{\mathrm{ODE}}(t^k, \mathbf{x}^k, \mathbf{u}^k) \equiv \mathbf{x}^k. \tag{1.10}$$

In order to be able to treat higher index DAEs, the initialization of consistent initial conditions has to be extended. This can be done by the introduction of additional elements in $\tilde{\mathbf{x}}^k$, so called algebraic states or dummy derivatives (Mattsson and Söderlind, 1993). An other possibility is to use the initialization (1.10) also for DAEs, but to introduce additional constraints for the initial states into the optimization problem and to augment the integration procedure, see e.g. (Grupp, 1996). The choice of an appropriate extension and its implementation into Omuses is matter of further investigation.

The system equations (1.4), the criterion (1.5), and the constraints (1.7) are now defined as derived quantities of the continuous-time final states $\tilde{\mathbf{x}}^k(t^{k+1})$, besides $\mathbf{x}^k$ and $\mathbf{u}^k$, as

$$\begin{aligned} \mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k) &\equiv \mathbf{f}^k\left[\mathbf{x}^k, \mathbf{u}^k, \tilde{\mathbf{x}}^k(t^{k+1})\right], \\ f_0^k(\mathbf{x}^k, \mathbf{u}^k) &\equiv f_0^k\left[\mathbf{x}^k, \mathbf{u}^k, \tilde{\mathbf{x}}^k(t^{k+1})\right], \\ \mathbf{c}^k(\mathbf{x}^k, \mathbf{u}^k) &\equiv \mathbf{c}^k\left[\mathbf{x}^k, \mathbf{u}^k, \tilde{\mathbf{x}}^k(t^{k+1})\right]. \end{aligned}$$

In general not all elements of $\mathbf{x}^k$ are defined by differential equations. That is why the first $n_{d,k} \leq n_{\tilde{x},k}$ elements of $\tilde{\mathbf{x}}^k$ that are not defined in (1.8) are treated as constants during the integration process ($\tilde{x}_i^k(t) = \tilde{x}_i^k(t^k); 0 \leq i < n_{d,k}; t \in [t^k, t^{k+1}]$). The integration is omitted for problems without differential equations ($n_{d,k} = n_{\tilde{x},k}$).

1.2.4 Multiple sample periods per stage

Each stage k of the multistage problem (1.4)–(1.7) may cover $n_k \geq 1$ sample periods $k^0, k^1, \ldots, k^{n_k-1}$ within its communication period $[t^k, t^{k+1}]$, $t^k = t^{k,0} < t^{k,1} < \ldots < t^{k,n_k-1} < t^{k,n_k} = t^{k+1}$. The system equations (1.4) are solved successively over all sample periods, that is

$$\begin{aligned} \mathbf{f}^k(\mathbf{x}^k, \mathbf{u}^k) &\equiv \mathbf{f}^{k,n_k-1}(\mathbf{x}^{k,n_k-1}, \mathbf{u}^k), \\ \mathbf{x}^{k,l+1} &= \mathbf{f}^{k,l}(\mathbf{x}^{k,l}, \mathbf{u}^k), \quad l = 0, \ldots, n_k - 2, \\ \mathbf{x}^{k,0} &= \mathbf{x}^k. \end{aligned} \tag{1.11}$$

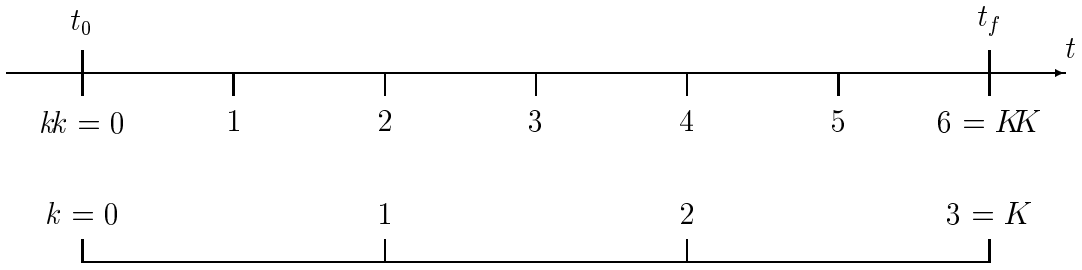

Figure 1.1: Exemplary separation of the time horizon $[t_0, t_f]$ into $K = 3$ stages with two sample periods per stage.

Specified differential equations are solved separately in each sample period as discussed in subsection 1.2.3. The optimization criterion (1.5) and the constraints (1.7) are defined as sums over all sample periods, that is

$$f_0^k(\mathbf{x}^k, \mathbf{u}^k) \equiv \sum_{l=0}^{n_k-1} f_0^{k,l}(\mathbf{x}^{k,l}, \mathbf{u}^k), \tag{1.12}$$

$$\mathbf{c}^k(\mathbf{x}^k, \mathbf{u}^k) \equiv \sum_{l=0}^{n_k-1} \mathbf{c}^{k,l}(\mathbf{x}^{k,l}, \mathbf{u}^k). \tag{1.13}$$

With one sample period per stage, the extended definitions (1.11), (1.12), and (1.13) reduce to $\mathbf{f}^k$, f_0^k, and $\mathbf{c}^k$ of the basic multistage formulation (1.4)–(1.7). On the other hand, small optimization problems can be formulated in one stage by using multiple sample periods (see also subsection 1.5.2).

With the help of this flexible mapping from the sample periods to the communication periods, the continuous time horizon can be separated according to model requirements (e.g. according to sampled measurement data), whereas the number of stages can be chosen and modified afterwards according to requirements of the optimization (e.g. useful optimization problem separation and communication periods).

Figure 1.1 shows an example with $K = 3$ and $n_k = 2,\ k = 0, 1, 2$. The counter kk is incremented globally over all stages, so that it can be used as index into vectors of sampled data.

1.2.5 Large-scale nonlinear programming problem

The multistage problem (1.4)–(1.7) can be treated as a high-dimensional nonlinear programming problem. The important point for its efficient solution is its nice separability structure. There are only linear constraints between the state variables of successive stages defined in (1.4). Consequently the Lagrangian Hessian of the problem has a block-diagonal sparsity structure with one block for each stage. The high-dimensional equation

system that results from the Karush-Kuhn-Tucker optimality conditions and that is solved by HQP has a block-banded sparsity structure that is worth exploiting, see (Franke and Arnold, 1997). Omuses accesses the DOCP interface of HQP (see section 2.4).

The multistage problem is transformed to a large-scale nonlinear programming problem of the form

$$J(\mathbf{x}) \to \min_{\mathbf{x}}, \quad J : \mathbb{R}^n \mapsto \mathbb{R}^1, \tag{1.14}$$

subject to:

$$\mathbf{h}(\mathbf{x}) = \mathbf{0}, \quad \mathbf{h} : \mathbb{R}^n \mapsto \mathbb{R}^{m_e}, \tag{1.15}$$

$$\mathbf{g}(\mathbf{x}) \geq \mathbf{0}, \quad \mathbf{g} : \mathbb{R}^n \mapsto \mathbb{R}^m. \tag{1.16}$$

J, $\mathbf{h}$, and $\mathbf{g}$ are assumed to be two times continuously differentiable with respect to $\mathbf{x}$. At least they should be sufficiently smooth to allow the treatment of the specific problem with an SQP algorithm.

It should be noted that neither the expanded states $\tilde{x}_i, n_{x,k} \leq i < n_{\tilde{x},k}$, nor the intermediate results at sample time points within communication periods appear in $\mathbf{x}$.

1.3 The Omuses problem interface

Optimization problems are formulated for Omuses in the programming language C++ by deriving an implementation from the interface class `Omu_Program`. Up to six methods can be overloaded in order to exploit the full functionality. Two methods (`setup()` and `update()`) must be implemented.

The documentation given in this section is intended to be used together with the examples in section 1.5 and the problem formulation in section 1.2 as a programmers reference. Small example codes are also listed in appendix B.1.

Vector and matrix data are defined with array data types of the Meschach library for matrix computations in C (Steward and Leyk, 1994) and the ADOL-C package for automatic differentiation (Griewank et al., 1996). Their application to the Omuses problem formulation is straightforward due to the use of C++ operator overloading. Small wrapper classes have been implemented for pointers to the Meschach data structures.

1.3.1 Problem setup

```
void setup_stages(IVECP ks, VECP ts)
```

The stages of the optimization problem are defined by the integer vector `ks` and the double vector `ts`. This can be done by calling the service routine

```
void stages_alloc(IVECP ks, VECP ts, int K, int sps,
                  double t0 = 0.0, double tf = 1.0)
```

with `K` the number of stages and `sps` the number of sample periods per stage. The optional arguments `t0` and `tf` define the time horizon $[t_0, t_f]$. The communication and sample time points are distributed equally over the time horizon by `stages_alloc()`. Furthermore the integers `_K` and `_KK`, that are members of `Omu_Program`, are set.

More generally, the vector `ts` $\in \mathbb{R}^{KK+1}$ can be initialized with increasing sample time points. The vector `ks` $\in \mathbb{I}^{K+1}$ can be initialized with increasing indices into `ts`, in order to determine the communication time points with the optimization.

The default implementation of `setup_stages()` initializes an optimization problem of the form (1.1)–(1.3) without stages.

Odc examples: `TP383omu`, `HS99omu`, `CranePar`, `Crane`

```
void setup(int k,
Omu_Vector &x, Omu_Vector &u, Omu_Vector &c)
```

The optimization variables are defined in each stage $k = 0, \ldots, K-1$ by the states $\mathbf{x}^k \in \mathbb{R}^{n_{x,k}}$, the controls $\mathbf{u}^k \in \mathbb{R}^{n_{u,k}}$, and the constraints $\mathbf{c}^k \in \mathbb{R}^{n_{c,k}}$. The method

```
void Omu_Vector::alloc(int n, int n_expand = -1)
```

is provided for the allocation of the variable vectors with dimension `n`. The optional argument `n_expand`, which defaults to `n`, can be specified for state variables. It determines the dimension $n_{\tilde{x},k} \geq n_{x,k}$ (see subsection 1.2.3).

Each `Omu_Vector` contains the members `min`, `max`, and `initial`, besides its actual elements. These vector attributes are used to store minimal, maximal and initial values. They default to `-Inf`, `Inf`, and `0.0`, respectively, for all their elements.

In the case $k = K$, only the final states $\mathbf{x}^K \in \mathbb{R}^{n_{x,K}}$ and the constraints $\mathbf{c}^K \in \mathbb{R}^{n_{c,K}}$ can be specified.

Odc examples: all

1.3.2 System equations, optimization criterion, and constraints

```
void update(int kk,
const adoublev &x, const adoublev &u,
adoublev &f, adouble &f0, adoublev &c)
```

The function values $\mathbf{f}^{kk} \in \mathbb{R}^{\max(n_{\tilde{x},k}; n_{x,k+1})}$, $f_0^{kk} \in \mathbb{R}^1$, and $\mathbf{c}^{kk} \in \mathbb{R}^{n_{c,k}}$ for the sample periods $kk = 0, \ldots, KK-1$ in the according stages $k = 0, \ldots, K-1$ are evaluated repeatedly during the optimization process. Their calculation has to be defined in the method `update()` depending on the current values of $\mathbf{x}^{kk} \in \mathbb{R}^{n_{\tilde{x},k}}$ and $\mathbf{u}^k \in \mathbb{R}^{n_{u,k}}$. In the first sample period of stage k, the argument $\mathbf{x}^{kk}$ is defined as $x_i^{kk} = x_i^k, 0 \leq i < n_{x,k}$ and $x_i^{kk} = \text{initial}(x_i^k), n_{x,k} \leq i < n_{\tilde{x},k}$. The initial values of the expanded states are specified in `setup()`. The functions $\mathbf{f}^{kk}$, f_0^{kk}, and $\mathbf{c}^{kk}$ define one term of (1.11), (1.12), and (1.13). Starting with the second sample period of stage k, $\mathbf{x}^{kk}$ contains the result $\mathbf{f}^{kk-1}$, respectively.

In the case $kk = KK$, only the criterion $f_0^{KK} \equiv f_0^K \in \mathbb{R}^1$ and the constraints $\mathbf{c}^{KK} \equiv \mathbf{c}^K \in \mathbb{R}^{n_{c,K}}$ can be specified.

The solution $\tilde{\mathbf{x}}^{kk}(t_{kk+1})$ of optional differential equations is passed as default value of $\mathbf{f}^{kk}$ to `update()`. If no differential equations are present, then $\mathbf{f}^{kk}$ defaults to $\mathbf{x}^{kk}$. The default values of f_0^{kk} and $\mathbf{c}^{kk}$ are zeros.

Odc examples: all

1.3.3 Continuous-time equations

```
void continuous(int kk, double t,
const adoublev &x, const adoublev &u,
const adoublev &xp, adoublev &F)
```

The continuous-time equations are defined in each sample period $kk < KK$ in the residual form according to (1.8). Standard numerical integration procedures are applied to the solution of the differential equations and their extension by sensitivity equations. Currently only solvers for non-stiff ODEs are implemented. It is assumed that $\frac{\partial \mathbf{F}^k}{\partial \dot{\mathbf{x}}^k}$ are constant, regular diagonal matrices in each stage k. The matrices are determined internally by Omuses during the problem setup.

Furthermore Omuses determines the first $n_{d,k} \leq n_{\tilde{x},k}$ components of $\mathbf{F}^k$ that are not defined in `continuous()`. They are treated as constants during the integration process. The methods `continuous()` and `consistic()` are not called anymore if $n_{d,k} = n_{\tilde{x},k}$.

The default implementation of `continuous()` is empty.

Odc examples: `HS99omu`, `CranePar`, `Crane`

```
void consistic(int kk, double t,
const Omu_Vector &x, const Omu_Vector &u,
VECP xt, MATP xtx = MNULL, MATP xtu = MNULL)
```

The method `consistic()` can be implemented for the initialization of the continuous-time variables $\tilde{\mathbf{x}}^{kk} \in \mathbb{R}^{n_{\tilde{x},k}}$ from $\mathbf{x}^{kk} \in \mathbb{R}^{n_{\tilde{x},k}}$ and $\mathbf{u}^k \in \mathbb{R}^{n_{u,k}}$ according to (1.9). In the first sample period of stage k, the argument $\mathbf{x}^{kk}$ is defined as $x_i^{kk} = x_i^k, 0 \leq i < n_{x,k}$ and $x_i^{kk} = \text{initial}(x_i^k), n_{x,k} \leq i < n_{\tilde{x},k}$. The initial values of the expanded states are specified in `setup()`. Starting with the second sample period of stage k, $\mathbf{x}^{kk}$ contains the result $\mathbf{f}^{kk-1}$ of `update()`, respectively.

The matrices `xtx`, that is $\frac{\partial \tilde{\mathbf{x}}^{kk}(t_{kk})}{\partial \mathbf{x}^{kk}}$, and `xtu`, that is $\frac{\partial \tilde{\mathbf{x}}^{kk}(t_{kk})}{\partial \mathbf{u}^k}$ are needed for the calculation of sensitivities. They are currently user-specified.

The default implementation copies `x` to `xt`, sets `xtx` to the identity matrix, and fills `xtu` with zeros. The method `consistic()` may be changed in the future, when integration procedures for DAEs will have been implemented. We decided to already document it, because the expansion mechanism can also be useful for ODEs, e.g. to implement fixed initial values of state variables.

Odc examples: none

1.3.4 Simulation of starting values

```
void init_simulation(int k,
Omu_Vector &x, Omu_Vector &u)
```

Often it is convenient to let the optimizer calculate starting values for optimization variables that represent system states. This can be done in Omuses with the help of subsequent initial-value simulations for all stages $k = 0, \ldots, K-1$. Before the simulation starts in stage k, the initial states $\mathbf{x}^k \in \mathbb{R}^{n_{\tilde{x},k}}$, and the control parameters $\mathbf{u}^k \in \mathbb{R}^{n_{u,k}}$ can be specified in `init_simulation()`. The states $\mathbf{x}^k, k > 0$ are set by default to the simulation results of the preceding stage $k-1$, respectively, and to zeros if $k = 0$. The control parameters u^k are set by default to zeros.

In the case $k = K$, only the simulated final states $\mathbf{x}^K \in \mathbb{R}^{n_{x,k}}$ can be modified.

The default implementation copies `x.initial` to `x` if $k = 0$, and `u.initial` to `u` for all $k = 0, \ldots, K-1$.

Odc examples: `Crane`

1.4 The Omuses command interface

Omuses, as well as HQP, contain an interface to the Tool Command Language (Tcl) (Ousterhout, 1994). This allows the flexible problem setup, solver configuration, and treatment of results. A small library of interface element types has been developed in order to encapsulate the creation and deletion of Tcl commands, to support the arrangement of C++ classes in a framework, and to propagate the information about implemented classes to the command interface, see also the discussion in (Franke, 1994b). For a listing of available interface elements see appendix A.

1.5 The Odc demo collection

In this section we discuss simple, illustrative examples. We try to point out several effects that are well known in the field of numerical optimization and that are important for the efficient application of Omuses.

C++ codes of the examples are available with the distribution. They can be compiled to the executable `odc` (Omuses demo collection).

First the multistage front-end Omuses is introduced by treating two small test examples. Each example is solved in two formulations: its original, unstructured formulation and a multistage formulation. The advantage of the multistage formulations for the solution of the optimization problems by HQP is shown.

Afterwards an optimal control problem is discussed. It is motivated by a laboratory for the undergraduate education (Arnold, 3 97). Here it is extended by a parameter and initial states estimation. Furthermore we want to explain the application of Omuses in connection with the modeling and simulation environment OmSim (Mattsson et al., 1993).

More practically oriented applications are stated in section “Applications” on page 39.

1.5.1 Nonlinear test examples

In this subsection the standard test examples `TP383` from the collection (Schittkowski, 1987) and `HS99` from (Hock and Schittkowski, 1981) are treated. The example `<name>` has been implemented in the files `Prg_<name>.[hC]`.

Nonlinear programming example TP383

The problem `TP383` in the 14 optimization variables $x_i,\ i = 0, \ldots, 13$ is

$$\sum_{i=0}^{13} a_i / x_i \quad \rightarrow \quad \min_{x_i},$$

subject to:

$$\begin{aligned} \sum_{i=0}^{13} c_i x_i &= 1, \\ x_i &\geq 0, \quad i = 0, \ldots, 13, \\ x_i &\leq 0.04, \quad i = 0, \ldots, 4, \\ x_i &\leq 0.03, \quad i = 5, \ldots, 13. \end{aligned} \tag{1.17}$$

Values for a_i and c_i, $i = 0, \ldots, 13$ are externally supplied.

In order to obtain an equivalent formulation over $K = 14$ stages, the state $s^k,\ k = 0, \ldots, K$ is introduced and the control parameters $u^k,\ k = 0, \ldots, K-1$ are used for the free optimization variables. The equivalent multistage formulation `TP383omu` is

$$\sum_{k=0}^{13} a_k / u^k \quad \rightarrow \quad \min_{u^k},$$

subject to

$$\begin{aligned} s^K &= 1, \\ s^{k+1} &= s^k + c_k u^k, \quad k = 0, \ldots, 13, \\ s^0 &= 0, \\ u^k &\geq 0, \quad k = 0, \ldots, 13, \\ u^k &\leq 0.04, \quad k = 0, \ldots, 4, \\ u^k &\leq 0.03, \quad k = 5, \ldots, 13. \end{aligned} \tag{1.18}$$

The Omuses implementation of `TP383` and `TP383omu` is listed in appendix B.1.

Optimal control example HS99

The problem `HS99` in the 7 optimization variables $x_i,\ i = 0, \ldots, 6$ is

$$-r_7^2 \ \rightarrow \ \min_{x_i},$$

subject to

$$\begin{aligned} q_7 &= 100000, \\ s_7 &= 1000 \\ 0 &\leq x_i \leq 1.58, \quad i = 0, \ldots, 6, \end{aligned} \tag{1.19}$$

where

$$\begin{aligned} r_i &= a_i(t_i - t_{i-1}) \cos x_{i-1} + r_{i-1}, \\ q_i &= 0.5(t_i - t_{i-1})^2(a_i \sin x_{i-1} - b) + (t_i - t_{i-1})s_{i-1} + q_{i-1}, \\ s_i &= (t_i - t_{i-1})(a_i \sin x_{i-1} - b) + s_{i-1}, \quad i = 1, \ldots, 7, \\ r_0 &= q_0 = s_0 = 0. \end{aligned}$$

Values for a_i, t_i, $i = 0, \ldots, 7$, and b are externally supplied.

The equivalent formulation `HS99omu` over $K = 7$ stages, with the time $t \in [t_0, t_f]$, and using the control parameters $u^k,\ k = 0, \ldots, K-1$ as optimization variables, is

$$-r(t_f)^2 \ \rightarrow \ \min_{u_k},$$

subject to

$$\begin{aligned} q(t_f) &= 100000, \\ s(t_f) &= 1000, \\ \dot{r}(t) &= a(t) \cos u(t), \\ \dot{q}(t) &= s(t), \\ \dot{s}(t) &= a(t) \sin u(t) - b, \quad t \in [t_0, t_f], \\ r(t_0) &= q(t_0) = s(t_0) = 0, \\ 0 &\leq u^k \leq 1.58, \quad k = 0, \ldots, 6, \end{aligned} \tag{1.20}$$

where

$$\begin{aligned} a(t) &= a_{k+1}, \\ u(t) &= u^k, \quad t \in [t^k, t^{k+1}),\ k = 0, \ldots, 6. \end{aligned}$$

A comparison of the formulations underlines the fact that it is normally more user friendly to formulate continuous-time problems in continuous time. Moreover additional flexibility is gained for the control parameterization.

Running the examples

To run an example, execute the Tcl script `run` with the example name as argument, e.g.

```
% run HS99
```

should produce the output

```
 it          obj  ||grdL||   ||inf|| [ qp res]     ||s||       s'Qs stepsize
  0  -7.7636e+08 2.404e+08 1.671e+05 [  5 opt]    0.1504  1.033e+07        1
  1 -8.32583e+08  2.98e+07      5448 [  3 opt]  0.008919  2.563e+04        1
  2 -8.31081e+08 6.837e+05     26.16 [  2 opt]  0.004368       4591        1
  3  -8.3108e+08 2.993e+05      4.44 [  2 opt] 0.0008231      111.9        1
  4  -8.3108e+08 2.542e+04    0.1272 [  2 opt] 0.0001473      5.663        1
  5  -8.3108e+08 2.145e+04  0.007399 [  2 opt] 4.489e-05      1.778        1
  6  -8.3108e+08     328.4  0.001167 [  2 opt] 2.341e-06   0.002231        1
  7  -8.3108e+08     303.2 3.752e-06 [  2 opt] 1.275e-06  0.0008991        1
  8  -8.3108e+08     5.266 8.203e-07 [  2 opt] 4.851e-08  4.588e-07
                                     22 qp-it
Result   : optimal
Objective: -8.3108e+08
Obj-evals: 8
Variables:
0.542468 0.529021 0.508449 0.480269 0.451236 0.409183 0.352788
```

This solution protocol can be interpreted as follows. The problem `HS99` is solved in 9 iterations. During each iteration one local, linear-quadratic approximation to the nonlinear optimization problem is formed, incorporating an initial-value simulation of the system model and a sensitivity analysis. The formed subproblems are treated by the QP-solver. In total 22 subiterations, i.e. solutions of a linear equation system of the dimension of all optimization variables and constraints, are performed. Additional 8 objective function evaluations are done for the step size control, each incorporating an initial-value simulation of the system model and the calculation of the constraints.

Table 1.1 shows the collected results for all examples. One interesting point is the significantly lower number of iterations needed to solve the multistage version of `TP383`. This

Name	n	m_e	m	sbw	f	Evals	Iters	QP-Iters	Time
TP383	14	1	28	26	728594	54	55	149	0.5
TP383omu	29	16	28	4	728594	8	9	79	0.2
HS99	7	2	14	12	-8.3108e8	8	9	22	0.1
HS99omu	31	26	14	9	-8.3108e8	6	6	30	0.5 (0.2)

Table 1.1: Results for the nonlinear test examples. Each row shows the numbers of optimization variables n, equality constraints m_e, inequality constraints (bounds) m, the semibandwidth of the factorized matrix, the obtained optimal objective function value f, the numbers of objective function evaluations, iterations and subiterations, and the consumed CPU-time $[s]$ using a Sun UltraSPARC 1/140 workstation.

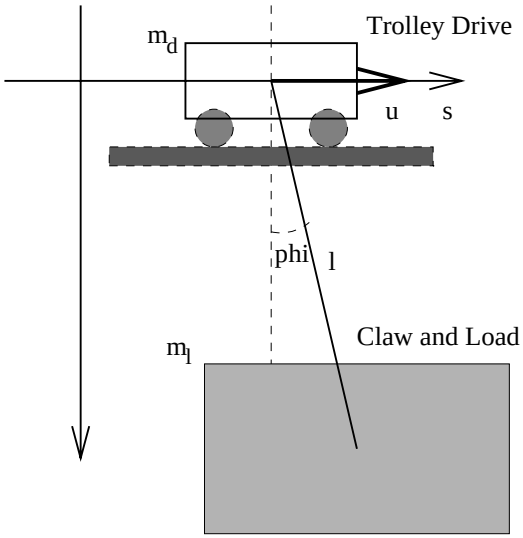

Figure 1.2: Container crane.

is caused by the better approximation of the Lagrangian Hessian of the problem due to partitioned BFGS update. In the unstructured formulation `TP383`, one matrix is updated for all optimization variables. In the multistage formulation `TP383omu`, one separate block that contains only one intrinsic optimization variable is updated for each stage.

The increase of the computational time for `HS99omu`, compared to `HS99`, is mainly caused by the numerical solution of the differential equations. The computational time can be reduced significantly by replacing the default integration routine RKsuite with the fixed stepsize method RK4 (value in parentheses). This can be done for this example without drawbacks, because the exact solutions with the applied piecewise constant control parameterization are piecewise low order polynomials.

1.5.2 Container crane

Model description

Figure 1.2 shows the schema of a crane used to carry container loads. It can be modeled by the following system of ordinary differential equations:

$$\begin{aligned} \dot{s} &= v \\ \dot{v} &= \frac{\frac{1}{2} m_l g \sin(2\phi) + m_l l \omega^2 \sin\phi + u F_{\text{scale}}}{m_d + m_l \sin^2\phi} \\ \dot{\phi} &= \omega \\ \dot{\omega} &= \frac{-(m_d + m_l) g \sin\phi - \frac{1}{2} m_l l \omega^2 \sin(2\phi) - u F_{\text{scale}} \cos\phi}{l(m_d + m_l \sin^2\phi)} \end{aligned} \tag{1.21}$$

where

s	position of the trolley drive	$[m]$

v	velocity of the trolley drive	$[ms^{-1}]$
m_d	mass of the trolley drive	$[kg]$
m_l	mass of the container load, the claw, and the connecting cable	$[kg]$
l	length of the connecting cable	$[m]$
ϕ	angle of the cable	[rad]
ω	angular velocity of the cable	$[\text{rad}s^{-1}]$
g	acceleration of gravity	$[ms^{-2}]$
u	control of the trolley drive	$[-]$
F_{scale}	scaling factor	$[N]$

The model is formulated in the modeling language Omola in the file `crane.om` (see appendix B.2.1). The modeling and simulation environment OmSim can be used to interactively perform initial value simulations. OmSim is freely available in binary form for several platforms, see `http://www.control.lth.se/~cace/omsim.html`. It is not required by Omuses.

Transferring the Model from OmSim to Omuses

The procedure described in this subsection should be seen as a rapid prototype. It was developed in order to study the possibility of the connection of Omuses with an advanced modeling and simulation environment. We decided to include it here, because it has already been successfully applied to quite complex system models that practically can not be written down in state-space form by hand, e.g. (Franke and Hellström, 1997).

OmSim has the ability to output the compiled model equations and the initialization of the model variables symbolically. Based on this flat model output, a simple lexical translator to C has been implemented in the file `fla2c.l`, which is included in the distribution. An executable can be generated using `flex` and a C compiler.

To transfer the model, first the file `crane.fla` with the model equations and variables is generated.

```
% omsim crane.om init.ocl > crane.fla
```

After the startup of OmSim and the execution of the script file `init.ocl`, a simulator panel should appear that shows the state *Init* in its upper right corner. Choose the menu *Debug* → *Flat model* and exit OmSim.

The Omola-like syntax in `crane.fla` is translated to C and written into the file `crane.c`.

```
% fla2c < crane.fla > crane.c
```

The file `crane.c` (see appendix B.2.2) contains the following code fragments that can be inserted into a template for an `Omu_Program`.

- The bound parameters and the implicit discrete part needs to be calculated once during the problem setup.

- The dynamic model equations should be calculated in `continuous()`.
- The continuous part of the Omuses state vector is assigned to the according model variables.
- The initial values of the states can be used in two ways in `setup()`: to define initial state constraints and to specify starting values for the initialization of the problem.
- The setting of parameter default values and the creation of interface elements should be done in the constructor.
- The parameters and time-independent variables are declared of type `double`, the time-dependent model variables are declared of type `adouble`.
- Finally, a list of the Tcl names of all unresolved model variables is written. Only external model inputs should appear here.

Known insufficiencies of `fla2c` are:

- The Flamola input should be replaced by a well defined language for flat models and the grammar should be analyzed.
- The translator does not support implicit equations and multiple derivatives of the form `x''`.
- It can be neccessary to manually modify the generated C code.
  - Function names are not translated from Omola to C (e.g. `ln` is used for the natural logarithm).
  - Powers of the form `x^y` are only translated to `pow(x,y)`, if `x` is a variable name and `y` a constant number.
  - Conditional assignments (that should be avoided if possible!) may lack a type cast to `adouble` for compound expressions in the alternative branch. This can be solved by enclosing the expression into parentheses.

These insufficiencies are detected by a C++ compiler, so that they do not result in wrong executables. The correctness of the transferred model can be checked by observing the results of the initial value simulation (`init_solution()`) after the problem setup.

For the following explanations we introduce the writing

$$\dot{\mathbf{x}} = \mathbf{f}_{\mathrm{m}}(t, \mathbf{x}, u, \mathbf{p}) \tag{1.22}$$

for the model equations (1.21) using the continuous-time state vector

$$\mathbf{x} = (\phi, \omega, v, s)^T \tag{1.23}$$

and the parameters $\mathbf{p}$.

Parameter and initial states estimation problem

The first problem we want to treat for the container crane is the estimation of the mass of the load m_l and of initial states $\mathbf{x}_0 = \mathbf{x}(t_0)$. The applied control force $u(t),\ t \in [t_0, t_f]$ is assumed to be known and measurements of the resulting trajectory of the trolley position $s(t)$ are available. The basic task for the solution of this problem is the minimization of a least squares criterion for the deviation between the set of measurement data $\{\bar{s}(t^{kk}), t^{kk} \in [t_0, t_f], kk = 0, \ldots, KK\}$ and the simulated model variables $s(t^{kk})$ at the measurement times.

$$\sum_{kk=0}^{KK} \|\bar{s}(t^{kk}) - s(t^{kk})\|^2 \rightarrow \min_{\mathbf{x}_0, m_l}, \tag{1.24}$$

subject to

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}_{\mathrm{m}}(t, \mathbf{x}, u, m_l), \quad t \in [t_0, t_f], \\ \mathbf{x}(t_0) &= \mathbf{x}_0. \end{aligned} \tag{1.25}$$

where

$$u(t) = \hat{u}, \quad t \in [t_0, t_f). \tag{1.26}$$

The according Omuses problem formulation can be found in the files `Prg_CranePar.[hC]`. The method `model_eq()` is defined for the calculation of the dynamic model equations. In addition to the four continuous-time states, one constant discrete-time state `x[0]` is introduced for the unknown load m_l. As Omuses needs to calculate the sensitivities of the model equations with respect to this parameter, the variable `ml` and its descendant `mdl` are redefined of type `adouble`. The state `x[0]` is assigned to `ml`, and `mdl` is recalculated. Furthermore the predefined control $\hat{u}$ is assigned to `u`.

The least squares criterion (1.24) is implemented in the method `update()` in two ways – multistage and singlestage. In both cases there are KK sample periods.

In the multistage case, one stage is used for each sample period ($K = KK$). This results in $5(KK + 1)$ optimization variables. The criterion is formulated directly as function of the optimization variables.

In the singlestage case ($K = 1$), there are 5 optimization variables for the initial states of the model. The criterion is defined for the first measurement point as function of the initial state variable $s(t_0)$, and for the following measurement points as function of intermediate results of the numerical integration.

No constraints are defined. Normally at least something should be known about initial states and valid ranges of free variables. The careful specification of bounds is often useful for the solution process. The values $\mathbf{x}_0 = (0, 0, 0, 25)^T$ and the (scaled) default value of the load $m_l = 4000kg$ are used as initial guess in the method `setup()`.

The method `disturb()` and the according Tcl command `prg_disturb` are defined to apply a simple disturbance to the measurement data. This makes the parameter estimation more

K	ad	f	Evals	Iters	Time	m_l	$\phi(t_0)$	$\omega(t_0)$	$v(t_0)$	$s(t_0)$
20	yes	0.01716	12	11	3	5984	0.1030	0.1986	-1.987	19.97
1	yes	0.01716	47	48	9	5984	0.1030	0.1986	-1.987	19.97
20	no	0.01718	22	11	5	5984	0.1030	0.1986	-1.987	19.97
1	no	0.01727	47	46	15	5977	0.1035	0.1976	-1.977	19.96
Reference with maxdev on $\bar{s}(t^{kk})$ $\pm 0.05m$						6000	0.1000	0.2000	-2.000	20.00

Table 1.2: Results for the parameter and initial states estimation problem. The table shows the number of stages K, the usage of automatic differentiation (ad), the final residual value f, the numbers of calculated residual values and iterations, the consumed CPU-time $[s]$ using a Sun UltraSPARC 1/140 workstation, and the obtained values.

realistic, as the measurements are simulated. The OmSim simulator can be used to record data by invoking

```
% omsim crane.om record.ocl
```

The script file `record.ocl` initializes the crane model, sets values for the container mass and the initial states, performs an initial-value simulation over the time horizon $t \in [0s, 5s]$, and writes data samples for the trolley position $s(t)$ into the file `record.plt`.

The parameter and initial states estimation, based on the data in `record.plt`, is initialized and driven from the script file `cranepar.tcl`. For its execution type

```
% run CranePar
```

Table 1.2 lists calculated results for four different option settings done in the file `cranepar.tcl`. Again we make the observation that the problem can be solved with significantly less iterations in its multistage formulation, resulting in a faster solution. If the automatic differentiation is switched off, then the solver is not able to obtain the optimum with the same accuracy anymore. The sensitivities are then calculated by using finite forward differences (the extra function evaluations are not listed in table 1.2). The optimization algorithm fails close to the optimum in finding further improvements. This can also lead to a stall error because the step size breaks down before the solution has been calculated with the required accuracy. The stopping criterion could be relexed in these cases. However, especially when many iterations are performed, slow convergence might suggest a bad result.

The calculation of sensitivities by using finite forward differences was implemented for testing purposes prior to the introduction of automatic differentiation into Omuses. It should not be needed anymore.

Optimal control problem

In this subsection we want to solve an optimal control problem numerically. The unknown continuous-time control trajectory is parameterized and a multistage problem is formulated and solved for the parameters. The resulting approximation of the optimal control trajectory is used to perform an initial-value simulation with OmSim.

The optimal control problem is

$$t_f \rightarrow \min_{u(t)}, \quad t \in [0, t_f] \tag{1.27}$$

subject to

$$\begin{aligned}
\dot{\mathbf{x}} &= \mathbf{f}_{\mathrm{m}}(t, \mathbf{x}, u, \mathbf{p}), && (1.28)\\
\mathbf{x}(0) &= (0,0,0,25)^T, \quad \mathbf{x}(t_f) = (0,0,0,0)^T, && (1.29)\\
|u(t)| &\leq 5, && (1.30)\\
|\phi(t)| &\leq 5\frac{\pi}{180}, \quad 0 \leq s(t) \leq 25. && (1.31)
\end{aligned}$$

Here we approximately describe $u(t)$ with the continuous, piecewise linear function u($\mathbf{p}$) with $K+1$ degrees of freedom for the initial value u(0) and for the slopes u^k, $k = 0, \ldots, K-1$. The Omuses problem definition can be found in the files `Prg_Crane.[hC]`. It is based on the automatically generated C code fragments listed in appendix B.2.2.

The bound parameters and implicit discrete part is calculated once in `setup()`.

The method `model_eq()` is defined for the calculation of the dynamic model equations. Instead of applying the control parameter `u[0]` directly, the additional continuous-time state `x[5]` is introduced for the control trajectory, and `u[0]` is used to describe its slope. Furthermore the constant discrete-time state `x[0]` with free initial and final value is introduced for the unknown final time t_f. The result of `model_eq()` is scaled with the final time in `continuous()`.

The dimensionless horizon $[0, 1]$ with `_K` stages and one sample period per stage is allocated in the method `setup_stages()`.

The initial and final state constraints (1.29) and the control bounds (1.30) are specified in `setup()`. The state bounds (1.31) are applied to the initial state values of each stage (except for the first stage where all states are fixed). The impact of this simplified implementation of the trajectory constraints will be discussed below. Furthermore, a reasonable lower bound for `x[0]` is specified.

The method `update()` defines the state `x[0]` to be constant and implements the criterion (1.27).

The control law

$$\begin{aligned}
u(t) &= \begin{cases} -\bar{u}(t_f), & t < t_f/2 \\ \bar{u}(t_f), & \text{else} \end{cases}\\
\bar{u}(t_f) &= \frac{4(m_d + m_l)s(t_0)}{{t_f}^2 F_{\mathrm{scale}}} && (1.32)
\end{aligned}$$

is implemented in `setup()`. This control is a first approximation of the solution. It can be derived under the assumption that all moving mass is concentrated in the trolley drive, and that the control task should be fulfilled for $s(t_f)$ and $v(t_f)$. The initial guess $t_f = 10s$ can be obtained with the help of equation (1.32), using the given mass parameters, the initial position, and the control bounds of the problem.

The setup and execution of the optimization solver is coded in the script file `crane.tcl`. When invoking

```
% run Crane
```

the following solution protocol should be written for $K = 50$ stages.

```
 it          obj  ||grdL||   ||inf|| [ qp res]     ||s||       s'Qs stepsize
  0           10         1      3.47 [ 23 sub]     52.87  0.0001516        1
  1      11.4934         1     1.579 [ 28 opt]     53.09  8.581e-05        1
  2      11.7757    0.2979    0.9662 [ 30 opt]     52.06   0.003439        1
  3      11.6746   0.05689    0.1052 [  5 opt]    0.6624  2.054e-07        1
  4      11.6751   0.00149 6.486e-06 [  5 opt] 0.0001333  1.185e-14        1
  5      11.6751 1.879e-07 1.541e-13
                                     91 qp-it
Result: optimal
```

The QP-solver result `sub` in the first iteration means that no feasible solution could be found based on the linear approximation of the nonlinear constraints at the simulated initial trajectories. Nevertheless the returned 'suboptimal' solution is useful for the SQP-solver in this case. Starting with the second iteration, the problem is solved without problems. Figure 1.3 shows the calculated trajectories for u, s, and ϕ with $K = 250$ stages. The initial solution for ϕ (grey trajectory) shows a strong oscillation. This is the reason for the infeasible first iteration. The trouble can be circumvented by changing the initial solution in a way that reduces the oscillation, e.g.

- restrict the initial state value in each stage by defining the method `init_simulation()`,
- relax the initial time guess, e.g. to $t_f = 15s$,
- apply the constant initial control $u(t) = -\bar{u}/2,\ t \in [0, t_f]$, not considering the final state constraint for $v(t_f)$.

Finally the solution has to be validated. In particular it should be reminded that the continuous-time trajectory constraints for the states ϕ and s have only been defined at $K-1$ communication time points so far. This simplifies the numerical solution of the problem. But the continuous-time state trajectories could violate their bounds between the communication points. Here we check the solution with the help of an initial-value simulation. The script `crane.tcl` writes the calculated optimal values into the file `control.plt`. The piecewise linear control trajectory is used as input for OmSim. The according setup is coded in the file `control.ocl`.

```
% omsim crane.om control.ocl
```

Please check the simulation stop time (it should be equal to the calculated optimal value) and press the *Start* button. According to the simulation, the trajectory constraints are practically fulfilled for $K = 50$. The small violations of the bounds for $\phi(t)$ completely disappear for $K = 250$. A serious violation can be seen for $K = 10$. Already the optimization result suggests that the approximation of $u(t)$ is insufficient in this case.

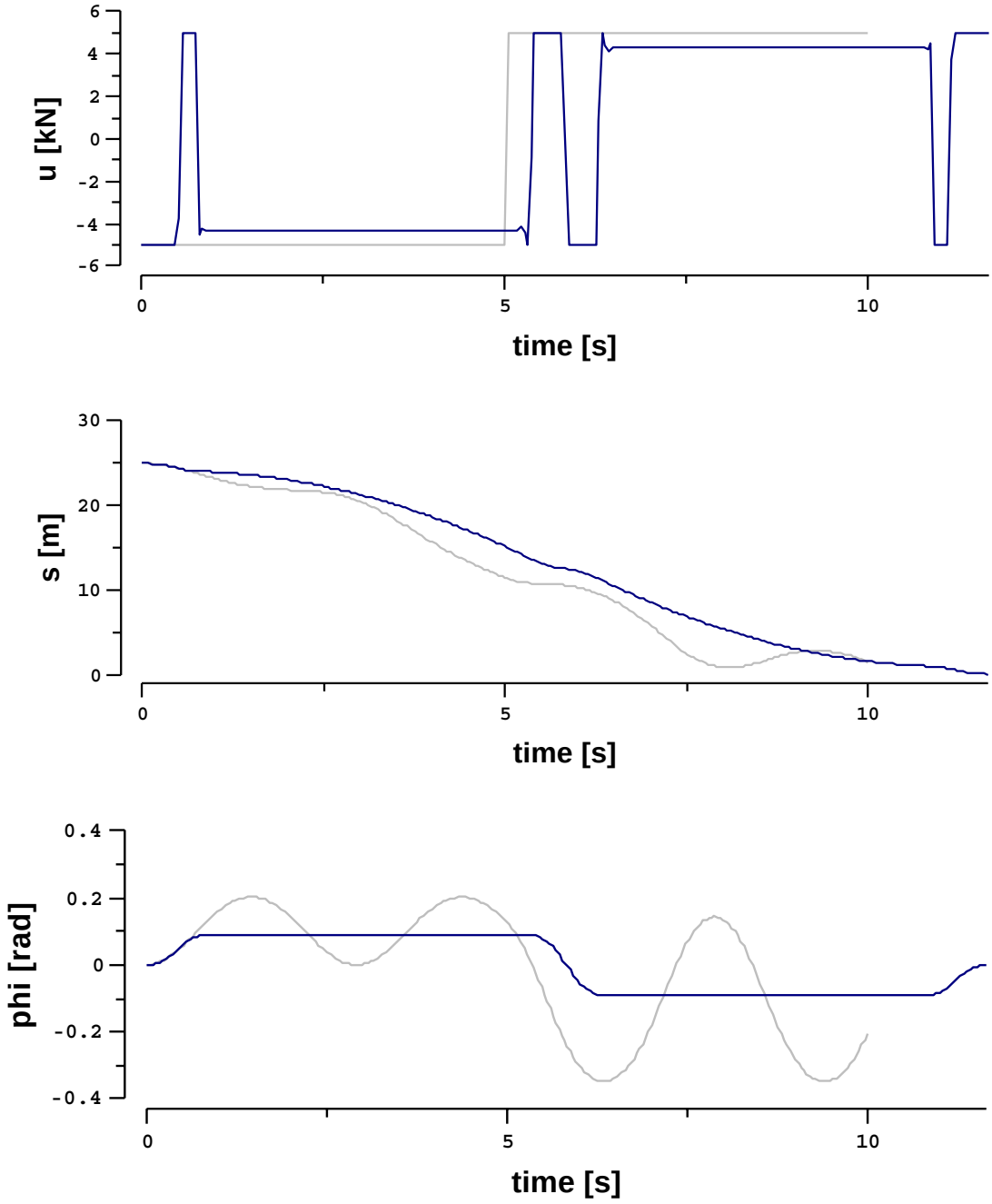

Figure 1.3: Unbounded initial values (grey) and calculated optimal trajectories (black) for the crane control problem with $K = 250$ equally sized stages and piecewise linear control parameterization.

Alternatively to the increase of K, the sample time points could also be redistributed (the current vector of time points `ts` can be accessed through the interface element `prg_ts`, see e.g. `cranepar.tcl`).

However, sometimes the proper choice of communication time points allone does not give satisfactory results. (In the crane example, the control parameter of each stage could dirictly be used to approximate $u(t)$ piecewise constant, see `Prg_Crane.uconst.C`. But then the solution for $\phi(t)$ would have steady oscillations, crossing the bounds at the communication points.) In such cases the problem formulation must be extented, e.g.:

- Modify the control parameterization. (In the crane example, the oscillations do not occur with the chosen continuous, piecewise linear control parameterization.)
- Introduce additional constraints for intermediate values of the state trajectory. The difference to the introduction of additional stages is, that the control parameters are the same for all sample periods of a stage. (In the crane example with piecewise constant control parameterization, the problem can be solved by defining two sample periods per stage and a constraint for the intermediate value of ϕ.)
- Define a trajectory constraint by using a continuous-time state variable.

In a practical application, the calculated optimal control normally can not be applied directly to the modeled process, caused by model uncertainties and unknown disturbances. But at least it can be used as a reference. The optimal objective function value gives us a limit for potential improvements of an implemetation.

An important practical approach is the iterative refinement of the optimal control based on new knowledge about the process. The two examples explained for the container crane, the adaptation of a system model to measurements and the calculation of the optimal system control subject to constraints, are basic algorithmic steps in Model-based Predictive Control (MPC) (Clarke, 1994).

Chapter 2

HQP

2.1 Introduction

During the last years, we have been successfully applying the method of sequential quadratic programming (SQP) to the solution of optimal control problems. Thereby, continuous time problems are discretized and treated as structured, large-scale nonlinear programming problems (Arnold et al., 1994).

The decision for the development of HQP was made because of several reasons. First of all, theoretical properties and practical results reported for interior point methods in the literature, e.g. (Wright, 1993), inspired us to apply them to our problems as well. Moreover, we wanted to exploit the powerful computing facilities widely available nowadays. The optimization solver HQP has been designed using general purpose algorithms and data structures for nonlinear programming and matrix algebra. That is why it is not only applicable to discrete-time optimal control problems, but also to other optimization problems.

The solver is based on sparse matrix codes of the Meschach library for matrix computations in C (Steward and Leyk, 1994). The tool command language Tcl (Ousterhout, 1994) was chosen for setting parameters and for controlling the execution.

Two different interfaces are currently available for the specification of optimization problems. During applications to several engineering problems, we have been developing the interface DOCP for discrete-time optimal control problems. Such multistage problems can be formulated in the programming language C++. The DOCP interface is accessed by Omuses (see chapter 1).

General large-scale optimization problems, formulated in the standard input format (SIF), can be passed trough a low-level interface to the constrained and unconstrained testing environment (CUTE) (Bongartz et al., 1995). The treatment of several available test problems has been very helpful for the development of the HQP solver. Furthermore, the low-level interface can be used for testing purposes. We have installed an online compute service that can be accessed through the Internet. All examples discussed in this chapter are taken from the CUTE collection.

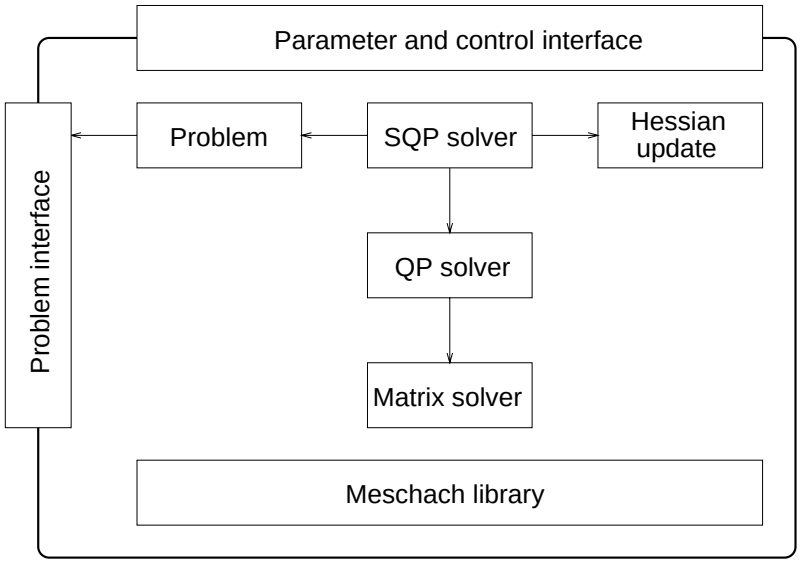

Figure 2.1: Modular structure of HQP. The arrows symbolize accesses. All modules are connected to the parameter and control interface and use the Meschach library for matrix computations.

2.2 Outline of the algorithm

This section gives a short overview of the overall structure of HQP and the implemented numerical methods. For more details is referred to Franke (1994a).

Figure 2.1 shows the component framework of HQP. It is implemented in the programming language C++. The scripting language Tcl is used for configuration and for control of execution.

2.2.1 Problem formulation

HQP addresses the general constrained nonlinear optimization problem in the variable vector $\mathbf{x}$, stated by the criterion $f(\mathbf{x})$, the equality constraints $\mathbf{h}(\mathbf{x})$, and the inequality constraints $\mathbf{g}(\mathbf{x})$. The problem is staged as

$$f(\mathbf{x}) \to \min_{\mathbf{x}}, \quad f : \mathbb{R}^n \to \mathbb{R}^1 \tag{2.1}$$

subject to:

$$\mathbf{h}(\mathbf{x}) = \mathbf{0}, \quad \mathbf{h} : \mathbb{R}^n \to \mathbb{R}^{m_e}, \tag{2.2}$$

$$\mathbf{g}(\mathbf{x}) \geq \mathbf{0}, \quad \mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m. \tag{2.3}$$

$f(\mathbf{x})$, $\mathbf{h}(\mathbf{x})$, and $\mathbf{g}(\mathbf{x})$ are assumed to be sufficiently smooth.

2.2.2 The SQP solver

A Lagrange-Newton-type SQP algorithm treats the problem (2.1) – (2.3) with the Lagrangian function

$$L(\mathbf{x}, \mathbf{y}, \mathbf{z}) = f(\mathbf{x}) - \mathbf{y}^T \mathbf{h}(\mathbf{x}) - \mathbf{z}^T \mathbf{g}(\mathbf{x}). \tag{2.4}$$

The Lagrangian multiplier vectors $\mathbf{y} \in \mathbb{R}^{m_e}$ and $\mathbf{z} \in \mathbb{R}^m$ are introduced for the equality and inequality constraints, respectively. The algorithm attempts to find a stationary point of the Lagrangian with the help of solutions of local, linear-quadratic approximations to the nonlinear optimization problem (2.1) – (2.3).

In each SQP iteration i, a quadratic approximation to the Lagrangian (2.4) and local linearizations of (2.1) – (2.3) at a given iterate $\mathbf{x}^i$ of the variables $\mathbf{x}$ are used to formulate the convex quadratic optimization problem

$$\frac{1}{2}\mathbf{s}^T \mathbf{Q}_i \mathbf{s} \; + \; \left(\frac{\partial f(\mathbf{x}^i)}{\partial \mathbf{x}} \right)^T \mathbf{s} \; \rightarrow \; \min_{\mathbf{s}} \tag{2.5}$$

subject to:

$$\frac{\partial \mathbf{h}(\mathbf{x}^i)}{\partial \mathbf{x}} \mathbf{s} \; + \; \mathbf{h}(\mathbf{x}^i) \; = \; \mathbf{0}, \tag{2.6}$$

$$\frac{\partial \mathbf{g}(\mathbf{x}^i)}{\partial \mathbf{x}} \mathbf{s} \; + \; \mathbf{g}(\mathbf{x}^i) \; \geq \; \mathbf{0} \tag{2.7}$$

in the variable vector $\mathbf{s}$. The solution $\mathbf{s}^i$ of (2.5) – (2.7) is taken as descending direction for a suitable line-search function, in order to obtain an improved iterate $\mathbf{x}^{i+1}$.

Two SQP modules were written for HQP based on the algorithms of Powell (Powell, 1978), and Schittkowski (Schittkowski, 1983). In general, we would recommend Powell's algorithm, which behaves in our applications very robust for the solution of large-scale problems. The implementation of Powell's algorithm has been extended with a watchdog strategy (Chamberlain et al., 1982) to further improve the robustness.

The linear approximation of nonlinear constraints may give subproblems without feasible solution, even though a solution of the nonlinear problem exists. Normally, additional slack variables are introduced in an SQP algorithm (Powell, 1978), (Tone, 1983). In HQP, this effect is accounted for by the QP solver, where we exploit the properties of the interior point algorithm (see subsection 2.2.3).

The quadratic approximation of the Lagrangian (2.4) is probably most crucial to the advantageous application of SQP-type solvers to large-scale problems. First of all, the matrix $\mathbf{Q}_i$ must be positive definite, to get a convex quadratic subproblem, which can be treated efficiently by the QP solver. Secondly, $\mathbf{Q}_i$ should be sufficiently sparse, in order to allow the efficient application of sparse matrix solvers.

The sparsity requirement is often fulfilled by the analytical Lagrangian Hessian

$$\mathbf{Q}_{i+1} = \frac{\partial^2 L(\mathbf{x}^{i+1}, \mathbf{y}^i, \mathbf{z}^i)}{\partial^2 \mathbf{x}} \mathbf{x}, \tag{2.8}$$

whose sparsity is inherent to many large-scale problems, see e.g. (Conn et al., 1992). Positive definiteness is guaranteed by HQP with the diagonal offsets

$$q_{i,i} := \max(\epsilon + \sum_{j \neq i} |q_{i,j}|,\ q_{i,i});\ i, j = 1, \ldots, n \tag{2.9}$$

(Gerschgorin modification). However, often this modification results in very poor convergence. Other authors add a constant value, which is based on the Gerschgorin bound of the most negative eigenvalue of the Lagrangian Hessian and an additional Levenberg parameter, to all diagonal elements (Betts and Frank, 1994).

In general, numerical Hessian updates are preferred in nonlinear programming. The positive definite update for sparse Hessians has been an open area of research for many years (Fletcher, 1995). In the current version of HQP, we have implemented dense BFGS updates for separate diagonal blocks of the Lagrangian Hessian. This partitioned variable metric update has proven to be very useful for the solution of multistage problems, the actual intension of the HQP solver. In order to be able to apply the partitioned BFGS update to general nonlinear programming problems, we are experimenting in the CUTE interface with the automatic introduction of dummy variables. The current implementation of this feature is on an early stage and can probably not be completed without reimplementing some CUTE procedures (see also subsection 2.7.2).

Thirdly, HQP provides a simple diagonal scaling procedure for the Lagrangian Hessian. According to our experience, this is the most reliable option for the solution of general large-scale nonlinear programming problems, even though the Newton-type solution method actually degenerates to a scaled gradient method in this case.

2.2.3 The QP solver

Most expenditure for solving the nonlinear problem (2.1) – (2.3) is propagated by the SQP solver in the form of convex quadratic subproblems (2.5) – (2.7) to the QP solver.

"It (the SQP solver) can be programmed in an afternoon if one has a quadratic programming subroutine available ..." (Powell, 1978).

Because of the importance of the QP solver, the name HQP was chosen for the whole nonlinear optimization tool. H stands for huge, a synonym for large-scale, as the letters L and S are already widely stressed in the optimization community.

Two different implementations are currently available with HQP: `Mehrotra` and `Franke`. Mehrotra's primal-dual predictor-corrector method (Mehrotra, 1992) was implemented because of its good reputation in the literature for linear programming, e.g. (Wright, 1996), by E. Arnold for HQP.

The other QP solver is derived from several interior point algorithms known from the literature, with most impact from (Wright, 1993). Some modifications were introduced for HQP with the aim to improve its performance and robustness (Franke, 1994a).

According to our experience the overall performance of both QP solvers is about the same. However, for a specific problem one of them may perform significantly better than the other and vice versa.

With variable substitution in (2.5) – (2.7) the quadratic programming subproblem can be rewritten as:

$$\frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \;+\; \mathbf{c}^T\mathbf{x} \;\rightarrow\; \min_{\mathbf{x}} \tag{2.10}$$

subject to:

$$\mathbf{A}\mathbf{x} \;+\; \mathbf{b} \;=\; \mathbf{0} \tag{2.11}$$

$$\mathbf{C}\mathbf{x} \;+\; \mathbf{d} \;\geq\; \mathbf{0} \tag{2.12}$$

$\mathbf{Q}$ is supposed to be symmetric positive semidefinite and $\mathbf{A}$ must be of full rank.

A barrier parameter $\mu > 0$ and a slack vector $\mathbf{w} = \mathbf{C}\mathbf{x} + \mathbf{d}$ are introduced for the treatment of inequality constraints. The objective function (2.10) is extended to:

$$\frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \;+\; \mathbf{c}^T\mathbf{x} \;-\; \mu\sum_{j=1}^{m}\ln w_j \rightarrow \min_{\mathbf{x}} \tag{2.13}$$

The solution of the quadratic subproblem is characterized by the extended Karush-Kuhn-Tucker conditions ($\mathbf{Z} = \mathrm{diag}(\mathbf{z}),\ \mathbf{W} = \mathrm{diag}(\mathbf{w}),\ \mathbf{e} = (1\ldots1)^T$):

$$\mathbf{Q}\mathbf{x} \;+\; \mathbf{c} \;-\; \mathbf{A}^T\mathbf{y} \;-\; \mathbf{C}^T\mathbf{z} \;=\; \mathbf{0} \tag{2.14}$$

$$\mathbf{A}\mathbf{x} \;+\; \mathbf{b} \;=\; \mathbf{0} \tag{2.15}$$

$$\mathbf{w} \;=\; \mathbf{C}\mathbf{x} \;+\; \mathbf{d} \;>\; \mathbf{0} \tag{2.16}$$

$$\mathbf{z} \;>\; \mathbf{0} \tag{2.17}$$

$$\mathbf{Z}\mathbf{W}\mathbf{e} \;-\; \mu\mathbf{e} \;=\; \mathbf{0} \tag{2.18}$$

$$\mu \;\rightarrow\; 0 \tag{2.19}$$

For each $\mu > 0$, equations (2.14) – (2.18) characterize a unique solution $(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w})$. For $\mu \rightarrow 0$, equation (2.18) reduces to the complementary condition $\mathbf{z}^T\mathbf{w} = 0$; (2.14) – (2.18) describe then the Karush-Kuhn-Tucker conditions of the original quadratic subproblem (2.10) – (2.12). The result vectors $\mathbf{y}$ and $\mathbf{z}$ are used by the SQP solver as approximations for the Lagrange multipliers.

The numerical solution of the nonlinear equation system (2.14) – (2.19) is obtained with an iterative Newton procedure, starting with suitable values $\mathbf{x}^0$, $\mathbf{y}^0$, $\mathbf{z}^0$, $\mathbf{w}^0$, and μ_0 (a further augmentation of (2.14) – (2.19) for obtaining a trivial starting point is discussed in (Franke, 1994a)). The following linear equation system is passed to the matrix solver during each QP iteration j:

$$\begin{pmatrix} -\mathbf{Q} & \mathbf{A}^T & \mathbf{C}^T & \mathbf{0} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{W}_j & \mathbf{Z}_j \end{pmatrix} \begin{pmatrix} \delta\mathbf{x}^j \\ \delta\mathbf{y}^j \\ \delta\mathbf{z}^j \\ \delta\mathbf{w}^j \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{Z}_j\mathbf{W}_j\mathbf{e} - \mu_j\mathbf{e} \end{pmatrix} \tag{2.20}$$

A very interesting property of interior point algorithms is that the failure of the complementary condition, i.e. $\mathbf{z}^T\mathbf{w} = 0$, is continuously decreased over the iterations. In this way, one has a kind of measure for the distance of a current iterate from the optimum. This is the basis for convergence proofs, e.g. (Monteiro and Adler, 1989), and motivates further developments, e.g. inexact Newton methods (Pang, 1986). In HQP, we exploit this property for the treatment of infeasible subproblem approximations.

2.2.4 The matrix solver

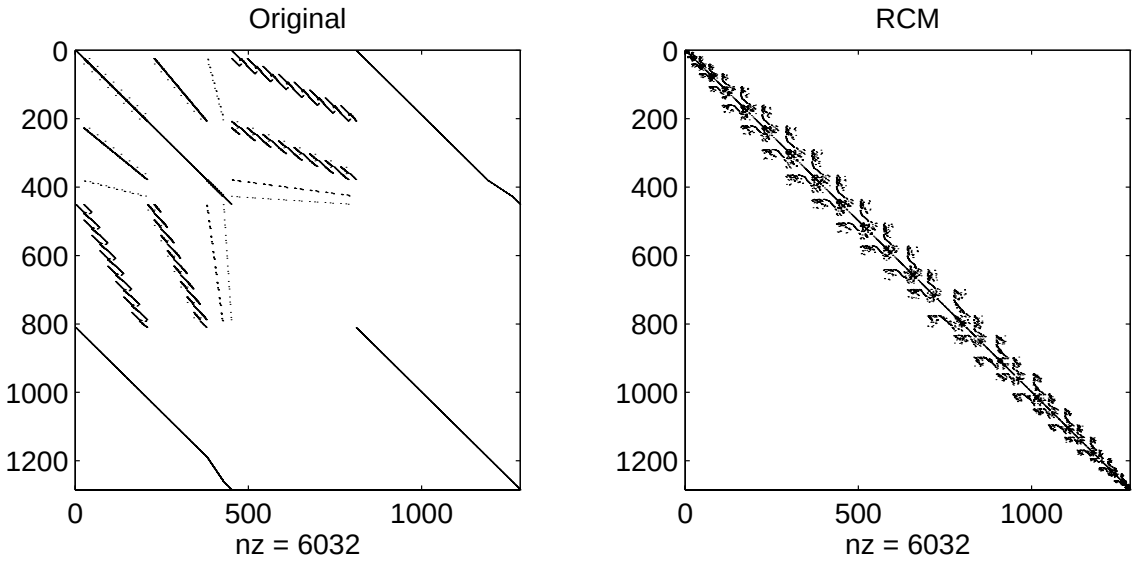

Figure 2.2: Sparsity pattern of the coefficient matrix (2.21) for the example BRITGAS of the CUTE collection. In the original matrix one can see in the upper left part the Hessian $\mathbf{Q}$ for the 450 problem variables. Below and right there are matrices $\mathbf{A}$ and $\mathbf{A}^T$, respectively, for 360 nonlinear equality constraints. In the lower left and the upper right parts there are $\mathbf{C}$ and $\mathbf{C}^T$, resulting from at all 474 variable bounds. Last but not least the diagonal matrix $\mathbf{Z}^{-1}\mathbf{W}$ finds in the lower right part. The profile of the whole coefficient matrix can be reduced considerably with symmetric Reverse-Cuthill-McKee (RCM) ordering.

The equation system (2.20) has to be solved once in each QP iteration. Currently four different matrix solvers are available with HQP.

The coefficient matrix of (2.20) can be made symmetric by eliminating $\delta\mathbf{w}^j = \mathbf{C}\delta\mathbf{x}^j$. This results in the symmetric, indefinite system

$$\begin{pmatrix} -\mathbf{Q} & \mathbf{A}^T & \mathbf{C}^T \\ \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{C} & \mathbf{0} & \mathbf{Z}_j^{-1}\mathbf{W}_j \end{pmatrix} \begin{pmatrix} \delta\mathbf{x}^j \\ \delta\mathbf{y}^j \\ \delta\mathbf{z}^j \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{W}_j\mathbf{e} - \mu_j\mathbf{Z}_j^{-1}\mathbf{e} \end{pmatrix}. \tag{2.21}$$

We apply a diagonal scaling as proposed by Wright (1993) in order to improve the numerical stability.

In many cases it is advantageous to further reduce the equation system prior to its factorization. This can be done by eliminating the inequality constraints from (2.21). This results in the reduced system

$$\begin{aligned} \begin{pmatrix} -\mathbf{Q} - \mathbf{C}^T\mathbf{W}_j^{-1}\mathbf{Z}_j\mathbf{C} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \delta\mathbf{x}^j \\ \delta\mathbf{y}^j \end{pmatrix} &= \begin{pmatrix} -\mathbf{C}^T\mathbf{W}_j^{-1}\mathbf{Z}_j\mathbf{r}_3^j \\ \mathbf{0} \end{pmatrix}, \qquad (2.22) \\ \delta\mathbf{z}^j &= \mathbf{W}_j^{-1}\mathbf{Z}_j(\mathbf{r}_3^j - \mathbf{C}\delta\mathbf{x}^j), \\ \mathbf{r}_3^j &= \mathbf{W}_j\mathbf{e} - \mu_j\mathbf{Z}_j^{-1}\mathbf{e}. \end{aligned}$$

The row and column order of the coefficient matrices in (2.21) and (2.22) is globally analyzed and permuted with a Reverse-Cuthill-McKee algorithm (Duff et al., 1992). Figure 2.2 shows sparsity patterns of an exemplary coefficient matrix of the form (2.21) and its RCM permutation. A sparse implementation of the factorization for symmetric, indefinite matrices by Bunch, Kaufman, and Parlett (Bunch et al., 1976) is applied afterwards. The implementations of the BKP matrix factorization and the RCM ordering were decoupled from the HQP framework and made as extensions to the Meschach library for matrix computations.

With the so called Schur complement method, see e.g. (Steinbach, 1995), the indefinite system (2.22) can be further reduced to two positiv definite equation systems of smaller dimension. An implementation for HQP has been made by H. Linke. In general the solution of definite systems is advantageous. Furthermore this method treats overdetermined contstraints implicitly (currently all other matrix solvers raise an error). However, drawbacks of this method are that the condition numbers of the factorized matrices become worse with the reduction and that the sparsity structure may be destroyed.

The fourth matrix solver developed by E. Arnold performs a block-wise elimination. It takes advantage of the stage-wise formulation of problems specified using Omuses. The algorithm is an extension of the Ricatti recursion for unconstrained linear-quadratic optimal control problems, see (Arnold, 1987), (Arnold et al., 1994). Applied to an example in (Arnold et al., 1998), the specialized method outperforms clearly the other matrix solvers with respect to computational time. An additional advantage of the method is that no dynamic allocation of memory is needed during the iterations if dense submatrices are used.

2.3 The parameter and control interface

All modules of HQP may be accessed through an interface, implemented with the tool command language Tcl (Ousterhout, 1994). The provided interface elements may be used in combination with standard Tcl commands, e.g. for conditional execution or for file accesses. For a listing of available interface elements see appendix A.

2.4 The DOCP problem interface

The DOCP (Discrete-time Optimal Control Problem) interface has been developed for multistage problems of the form (1.4)–(1.7). The multistage problem is transformed to a large scale nonlinear programming problem of the form (2.1)–(2.3) as follows.

The state variables $\mathbf{x}^k$, the control variables $\mathbf{u}^k$ of all stages k, and the final states $\mathbf{x}^K$ are

assembled to one large vector $\mathbf{x} \in \mathbb{R}^n$ of optimization variables

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}^0 \\ \mathbf{u}^0 \\ \vdots \\ \mathbf{x}^{K-1} \\ \mathbf{u}^{K-1} \\ \mathbf{x}^K \end{pmatrix}. \tag{2.23}$$

The optimization criterion (1.5), the system equations (1.4), the bounds (1.6), and the constraints (1.7) are assigned to the high-dimensional algebraic functions

$$J(\mathbf{x}) = F^K + \sum_k f_0^k, \tag{2.24}$$

$$\mathbf{h}(\mathbf{x}) = \begin{pmatrix} \mathbf{f}^0 - \mathbf{x}^1 \\ \vdots \\ \mathbf{f}^{K-1} - \mathbf{x}^K \\ \mathbf{h}^0 \\ \vdots \\ \mathbf{h}^K \end{pmatrix}, \; \mathbf{g}(\mathbf{x}) = \begin{pmatrix} \mathbf{x} - \mathbf{x}_\mathrm{l} \\ \mathbf{x}_\mathrm{u} - \mathbf{x} \\ \mathbf{c}^0 - \mathbf{c}_\mathrm{l}^0 \\ \vdots \\ \mathbf{c}^K - \mathbf{c}_\mathrm{l}^K \\ \mathbf{c}_\mathrm{u}^0 - \mathbf{c}^0 \\ \vdots \\ \mathbf{c}_\mathrm{u}^K - \mathbf{c}^K \end{pmatrix}. \tag{2.25}$$

The equality constraints $\mathbf{h}^0 \ldots \mathbf{h}^K$ result from those components of (1.6) and (1.7) that have equal lower and upper bounds, e.g. initial or final state constraints.

Please see the simple example provided with the distribution for details about using the DOCP interface.

2.5 The CUTE problem interface

The constrained and unconstrained testing environment (CUTE) (Bongartz et al., 1995) is developed in conjunction with the solver LANCELOT for large-scale nonlinear optimization (Conn et al., 1992). It is freely available, including an extensive collection of interesting nonlinear programming examples and a set of Fortran procedures for their transformation to standard formulations.

For the following explanations is assumed that CUTE is installed on your machine and that the environment variable `CUTEDIR` is set to the installation directory, see (Bongartz et al., 1995). See the distribution's README file for the additional installation of HQP.

After successful installation you can run the example HS11 as follows.

- Go to the problems directory.

  ```
  cd $CUTEDIR/problems
  ```

- Run HQP on HS11. (You should adapt your environment for repeated applications.)

  `$CUTEDIR/interfaces/sdhqp HS11`

  The shell script `$CUTEDIR/interfaces/sdhqp` executes the SIF decoder on the file `HS11.SIF`. The SIF specification is translated and written into a number of Fortran files. Afterwards the script file `$CUTEDIR/interfaces/hqp` compiles the Fortran modules. The executable program `hqpmin` is created and executed.

- You should get the following output:

```
 Problem name: HS11

 Double precision version will be formed.

 The objective function uses      1 nonlinear group

 There  is      1 nonlinear inequality constraint

 There are      2 free variables


 Semibandwidth:      1

 it          obj  ||grdL||   ||inf|| [ qp res]     ||s||       s'Qs stepsize
  0       -24.98       0.2     23.91 [ 10 opt]     2.424       11.8         1
  1     -18.5645     2.497     5.874 [  4 opt]     1.069      4.135         1
  2     -11.3907     2.476     1.143 [  4 opt]     0.647      0.948         1
  3     -8.59145    0.4338   0.03104 [  4 opt]   0.04298   0.003613         1
  4     -8.49852  0.009578 2.351e-05 [  3 opt]  0.003029  2.771e-05         1
  5     -8.49847  0.002249 1.525e-06 [  3 opt] 0.0003184  3.117e-07         1
  6     -8.49846 9.403e-05 1.646e-08 [  3 opt] 2.398e-05  1.842e-09         1
  7     -8.49846 3.747e-06 9.041e-11 [  3 opt] 9.256e-07  2.794e-12
                                      34 qp-it
HQP 1.5: optimal solution
f      : -8.49846
f_evals: 8
0.0u 0.0s 0:00 0% 0+0k 0+0io 0pf+0w
```

  The first five lines (output coded in the SIF decoder) characterize the problem. Note that HQP is only provided in a double precision version. The following lines (output coded in `$CUTEDIR/hqp/hqp_cute.tcl`) give a brief overview about the solution process. First, the semibandwidth of the permuted matrix (2.22) is printed. Each line of the following table shows:

  1. the SQP iteration,
  2. the objective function value,
  3. the norm of the gradient of the Lagrangian (2.4),
  4. the infeasibility norm (i.e. maximal violation) of a constraint,
  5. the number of QP iterations and the first three letters of the QP result,
  6. the maximum norm of $\mathbf{s}$,
  7. the value of $\mathbf{s}^T\mathbf{Q}\mathbf{s}$,

8. the obtained stepsize for the SQP iteration

Afterwards, the cumulative number of QP iterations, the HQP version and the result, the obtained function value, and the number of function evaluations are shown (output coded in `$CUTEDIR/hqp/hqp_cute.tcl`). The last line (output coded in `$CUTEDIR/interfaces/hqp`) summarizes execution times of HQP, as provided by the Unix command `time`.

- The solution is written in SIF format to the file `SOLUTION.d` in the current working directory (output coded in `$CUTEDIR/tools/sources/hqpma.f`).

The solver parameters can be changed in the file `$CUTEDIR/hqp/hqp_cute.tcl`.

2.6 Using HQP through the Internet

The easiest way for using HQP is to exploit our optimization service. It is built upon a batch compute service of the Computing Center of the Technical University of Ilmenau and runs on a pool of cooperating workstations.

To the outer world, the service appears like an ordinary user. Newest information can be obtained in the World Wide Web from `http://www.rz.tu-ilmenau.de/~hqp/`.

Requests can be submitted via the Network-Enabled Optimization System (NEOS Server) at Argonne National Laboratory (see `http://www.mcs.anl.gov/home/otc/`).

The currently supported format for describing an optimization problem is SIF (Standard Input Format) (Conn et al., 1992). It is backward compatible with the MPS format for linear programming.

2.7 Computational examples

The examples discussed in this section are taken from the small CUTE collection (Bongartz et al., 1995). The listed tables show

1. problem name in the CUTE collection
2. number of variables, equality constraints (including fixed variables), and inequality constraints (including bounds), respectively
3. semibandwidth of the RCM-ordered coefficient matrix in (2.22)
4. the obtained objective value
5. number of objective calculations
6. number of SQP iterations (the number of gradient and Hessian calculations is by one higher)
7. cumulative number of QP iterations
8. CPU time of HQP in seconds using a Sun UltraSPARC station 1/140

Table 2.1: Solved examples from different areas of application using default options.

Name	n	m_e	m	sbw	f	Evals	Iters	QP-Iters	Time
AUG2D	3280	1600	0	118	40565.4	2	1	1	7
AUG2DQP	3280	1600	3280	118	132458	2	1	66	134
BRITGAS	450	360	474	73	0.00000	11	10	384	47
CHEMRCTA	500	500	500	5	—	7	3	22	1
ROTDISC	905	207	175	353	7.87207	258	124	3199	272
ZAMB2	3966	1440	7920	13	-11.1312	20	19	270	112

2.7.1 Solved examples

All examples in this subsection were treated with default solver options. Table 2.1 summarizes the results.

AUG2D is the formulation of a boundary value problem for a two-dimensional partial differential equation (PDE) as convex quadratic programming problem. It can be solved in one QP-iteration because no inequality constraints are present.

AUG2DQP is a variant of AUG2D with lower bounds on the variables. 66 QP-iterations are needed in order to find the bound-constrained solution. The convex, linear-quadratic problem still can be solved in one major iteration.

BRITGAS describes a high pressure gas network. The network has 23 nodes and is observed over 8 hours with a time-step of one hour. The sparsity pattern of the internally solved Karush-Kuhn-Tucker system was shown in figure 2.2.

CHEMRCTA is a nonlinear equation system with lower bounds on the variables describing a chemical reactor model.

ROTDISC is an example for the design of a rotating disc of minimal weight subject to constraints for the engine of a small civil jet.

ZAMB2 is a discrete-time optimal control problem for the Zambezi multi-reservoir hydropower system. The problem is explained in (Arnold et al., 1994). In (Franke and Arnold, 1997), we used a variant of this problem as example to demonstrate the moderate increase of the computational time needed by HQP to solve inequality constrained problems of increasing dimension. In the variant discussed there, we neglected the reference control trajectories, which are considered with positive quadratic terms in the objective function (parameter PSI), so that the control and state bounds become more important at the solution.

2.7.2 Experiments with stretching problems

Currently, the Lagrangian Hessian of a nonlinear programming problem passed through the CUTE interface is approximated by default by a diagonal matrix. Of course this is not satisfactory and there finds many examples in the CUTE collection, where the number of performed SQP-iterations is very high. Often exact second order derivatives

Table 2.2: Improvements with stretching. The upper lines show results obtained with default solver options; the lower lines show results obtained with CUTE_ST and BFGS.

Name	n	m_e	m	sbw	f	Evals	Iters	QP-Iters	Time
ROSENBR	2	0	0	0	0.00000	7278	7268	7269	29
ROSENBR	2	0	0	1	0.00000	46	37	38	0
CATENA	501	170	0	5	-348530	24949	24730	24738	593
CATENA	996	665	0	6	-348530	58	21	24	2
ORBIT2	268	207	175	353	312.350	82	44	963	551
ORBIT2	464	379	233	22	312.350	53	32	714	54

and the Gerschgorin modification (2.9) do neither improve the performance in those cases. For problems passed through the high-level interface Omuses (see chapter 1), we have made best experiences with positive definite BFGS updates for separate diagonal blocks. Unfortunately, this update can not be applied to problems passed through the CUTE interface, even if the underlying optimization problem would allow it. That is why we made the alternative implementation CUTE_ST, which analyzes the nonlinear groups of the SIF formulation and introduces equality constrained dummy variables, in order to stretch the Lagrangian Hessian.

Table 2.2 summarizes results obtained with the solver options `CUTE_ST` and `BFGS`, compared to default settings.

ROSENBR is the well known Rosenbrock banana valley. Due to the small problem size, it can be solved using dense BFGS update anyway.

CATENA models a hanging catenary that consists of $N = 166$ rigid beams. The beams are concatenated to a freely hanging chain, which has fixed end positions. The catenary is described by using $N + 1$ coordinate points in the three-dimensional Cartesian space, which results in $3(N + 1) = 501$ optimization variables. The optimization problem is to find the beam positions by extremizing the total potential energy. After stretching the problem, each beam is described by 6 variables for its two end points, which results in $6N = 996$ optimization variables. Additional linear equality constraints are introduced for the junction conditions between the beams. In this way, separate BFGS updates can be made for the Hessian blocks that correspond to the beams. The solution of the stretched version of this non-convex problem is obtained significantly faster due to the better Lagrangian Hessian approximation.

ORBIT2 is an other typical example for the usefulness of stretching. It is a discrete-time optimal control problem with the objective to minimize the time needed for a spacecraft to change its circular orbit around the earth subject to a constrained control magnitude. Besides the discrete-time state and control variables, one additional variable is introduced for the free final time. This additional variable ingoes into all nonlinear equality constraints that result from the physical model. When applying stretching, the time variable is multiplied. Besides the possible application of partitioned BFGS update to the resulting Lagrangian Hessian, the sparsity structure of the problem is improved (semibandwidth 22 instead of 353). That is why the stretched version of the problem can be solved about 10 times faster with HQP.

Unfortunately, not much information about the separability structure of large-scale nonlinear programming problems formulated in SIF is available with the CUTE procedures. The following information is missed.

- Only the nonlinear groups, but not the elements are known. There finds many examples (e.g. DALLASL, HUESTIS, TRAINF, UBH1), where only one nonlinear group is defined for the whole objective function. This results in a full Lagrangian Hessian and often in memory overflows in the CUTE procedures using the finite-element data structure.

- For groups that contain linear and nonlinear variables, the according Lagrangian Hessian block contains not only the nonlinear variables, but also the linear variables. This is the reason, why the partitioned BFGS update of HQP practically does not work without stretching. Often too much dummy variables need to be introduced and the variable metric update is complicated numerically.

- The gradient of the objective function can not be calculated for separate groups. That is why the gradient of the Lagrangian can not be determined for some stretched problems.

- User-defined variable scaling is not known.

In order to further develop the treatment of problems formulated in SIF, we probably needed to directly access the Fortran procedures generated by the SIF decoder. Instead of doing this, we have been concentrating the recent development on the alternative high-level interface Omuses (see chapter 1). Nevertheless, the availability of CUTE helps considerably to further develop HQP and to improve the author's understanding for large-scale nonlinear programming.

Applications

The optimization software Omuses/HQP has been proving efficiency in a number of applications. Most important is its incorporation into a real-time process control system for a large water canal system in Northern Germany. The operational water management task is to maintain navigable water levels with minimum electrical energy (pump) costs. Optimal pump and discharge strategies are calculated based on predictions for the operation of the locks, wind stress, and natural inflows, using a sophisticated process model. The calculated decision proposals are updated repeatedly during a day within a receding horizon or model-based predictive control scheme, see e.g. (Linke et al., 1997) and (Arnold et al., 1998).

An other interesting application is the planning and management of water reservoir systems due to expected climate changes (CHEWS, 1998), (Fritsch et al., 1998).

The integration of Omuses with the object-oriented continuous systems modeling and simulation environment OmSim allows the efficient application to complex dynamical sytem models. The modeling tool is used for the automated compilation of system models that are imported by Omuses afterwards. Examples are studies to the optimal design of a solar heating system with ground heat store (Franke and Hellström, 1997) and the optimal control of a waste water treatment plant (Reichl, 1998), (Reichl et al., 1998).

The development of Omuses within the PhD thesis project (Franke, 1998) was driven by the requirements of its applications. We hope that the release under the GNU Library General Public License (LGPL) will allow both: more interesting solutions to optimization problems and further developments of the solution procedures.

Acknoledgements

The projects, this software has mainly been developed for, were supported by the *Deutsche Bundesstiftung Umwelt*, Osnabrück (grant 1000/276), *Deutsche Bundesanstalt für Wasserbau*, Karlsruhe (project "Optimierte Wasserbewirtschaftung des Mittellandkanals und des Elbe-Seiten-Kanals"), and the *Thüringer Ministerium für Wissenschaft, Forschung und Kultur*, Erfurt (project "VakuSol").

Bibliography

Arnold, E. (1987). *Zur optimalen Steuerung zeitdiskreter dynamischer Prozesse mittels nichtlinearer Optimierung mit Anwendungen auf die Klimasteuerung von Gewächshäusern.* PhD thesis, Technical University of Ilmenau.

Arnold, E. (1993-97). Numerische Lösung von Optimalsteuerungsaufgaben. Praktikumsanleitung (in German), Technical University of Ilmenau.

Arnold, E., Linke, H., and Franke, R. (1998). Optimal control application for operational water management of a canal system. In Koussoulas, N. and Groumpos, P., editors, *8th IFAC Symposium on Large Scale Systems LSS'98*, volume II, pages 810–815, Patras, Greece.

Arnold, E., Tatjewski, P., and Wołochowicz, P. (1994). Two methods for large-scale nonlinear optimization and their comparison on a case study of hydropower optimization. *J. Optim. Theory and Applics.*, 81(2):221–248.

Betts, J. and Frank, P. (1994). A sparse nonlinear optimization algorithm. *J. Optim. Theory and Applics.*, 82(3):519–541.

Bongartz, I., Conn, A., Gould, N., and Toint, P. (1995). CUTE: Constrained and unconstrained testing environment. *ACM TOMS*, 21(1):123–160.

Bunch, J. R., Kaufman, L., and Parlett, B. N. (1976). Decomposition of a symmetric matrix. *Numerical Mathematics*, 27:95–109.

Chamberlain, R., Powell, M., Lemarechal, C., and Pedersen, H. (1982). The watchdog technique for forcing convergence in algorithms for constrained optimization. *Mathematical Programming Study*, 16:1–17.

CHEWS (1998). CHEWS - The impact of climate change and other hydrological events on European water supply planning and management. Final report ENV4-CT95-0138, European Commission, DG XII.

Clarke, D., editor (1994). *Advances in Model-Based Predictive Control.* Oxford Science Publications. Oxford University Press.

Conn, A., Gould, N., and Toint, P. (1992). *LANCELOT – A Fortran Package for Large-Scale Nonlinear Optimization (Release A).* Number 17 in Springer Series in Computational Mathematics. Springer-Verlag, Berlin.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1992). *Direct methods for sparse matrices*. Oxford University Press, 3rd edition.

Fletcher, R. (1995). An optimal positive definite update for sparse Hessian matrices. *SIAM J. Optimization*, 5(1):192–218.

Fourer, R., Gay, D., and Kernighan, B. (1993). *AMPL: A Modeling Language for Mathematical Programming*. Boyd & Fraser Publishing Company, Danvers, Massachusetts.

Franke, R. (1994a). Anwendung von Interior-Point-Methoden zur Lösung zeitdiskreter Optimalsteuerungsprobleme. Diploma Thesis 200–94 D–14 (in German), Technical University of Ilmenau.

Franke, R. (1994b). User shell design with components for Tcl and Tk. In *2nd Tcl/Tk Workshop*, pages 177–182, New Orleans, Louisiana.

Franke, R. (1996). Internet enabled HQP optimization service – software demonstration. In *2nd IEEE European Workshop on Computer Intensive Methods in Control and Signal Processing*, pages 245–246, Prague, Czech Republik.

Franke, R. (1997). Object-oriented modeling of solar heating systems. *Solar Energy*, 60(3/4):171–180.

Franke, R. (1998). *Integrierte dynamische Modellierung und Optimierung von Systemen mit saisonaler Wärmespeicherung*, volume 394 of *Fortschritt-Berichte VDI, Reihe 6 (in German)*. VDI-Verlag, Düsseldorf.

Franke, R. and Arnold, E. (1996). On the integration of a large-scale nonlinear optimization tool with open modeling and simulation environments for dynamic systems. In Javor, A., Lehmann, A., and Molnar, I., editors, *10th European Simulation Multiconference*, Budapest, Hungary. The Society for Computer Simulation International (SCS).

Franke, R. and Arnold, E. (1997). Applying new numerical algorithms to the solution of discrete-time optimal control problems. In Warwick, K. and Kárný, M., editors, *Computer-Intensive Methods in Control and Signal Processing: The Curse of Dimensionality*, pages 105–118. Birkhäuser Verlag, Basel.

Franke, R. and Hellström, G. (1997). Optimization of solar heating systems with seasonal storage in the ground. In *Megastock '97, 7th Int. Conference on Thermal Energy Storage*, volume 1, pages 527–532, Sapporo, Japan.

Fritsch, P., Hopfgarten, S., and Puta, H. (1998). Planning and management of a water reservoir system due to expected climate changes. In Koussoulas, N. and Groumpos, P., editors, *8th IFAC Symposium on Large Scale Systems LSS'98*, volume II, pages 804–809, Patras, Greece.

Griewank, A., Juedes, D., and Utke, J. (1996). ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167. Algor. 755.

Griewank, A. and Toint, P. (1982). Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137.

Grupp, F. (1996). *Parameteridentifizierung nichtlinearer mechanischer Deskriptorsysteme mit Anwendungen in der Rad-Schiene-Dynamik*, volume 550 of *Fortschritt-Berichte VDI, Reihe 8 (in German)*. VDI-Verlag, Düsseldorf.

Hock, W. and Schittkowski, K. (1981). *Test Examples for Nonlinear Programming Codes*. Number 187 in Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin.

Linke, H., Arnold, E., and Franke, R. (1997). Optimal water management of a canal system. In Refsgaard, J. and Karalis, E., editors, *Operational Water Management*, pages 211–218. Balkema, Rotterdam.

Maly, T. and Petzold, L. (1996). Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20:57–79.

Mattsson, S., Andersson, M., and Åström, K. (1993). Object-oriented modeling and simulation. In Linkens, D. A., editor, *CAD for Control Systems*, pages 31–69. Marcel Dekker, Inc.

Mattsson, S. and Söderlind, G. (1993). Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific and Statistical Computing (SISSC)*, 14(3):677–692.

Mehrotra, S. (1992). On the implementation of a primal-dual interior point method. *SIAM J. Optimization*, 2(4):575–601.

Monteiro, R. D. C. and Adler, I. (1989). Interior path following primal-dual algorithms. part 2: Convex quadratic programming. *Mathematical Programming*, 44:43–66.

Ousterhout, J. K. (1994). *Tcl and the Tk toolkit*. Addison-Wesley, Reading, Massachusetts.

Pang, J.-S. (1986). Inexact Newton methods for the nonlinear complementarity problem. *Mathematical Programming*, 36:54–71.

Pantelides, C., Sargent, R., and Vassiliadis, V. (1994). Optimal control of multistage systems described by high-index differential-algebraic equations. *International Series of Numerical Mathematics*, 115:177–191.

Powell, M. (1978). A fast algorithm for nonlinearly constrained optimization calculations. In Watson, G. A., editor, *Proceedings of the Dundee Conference on Numerical Analysis, 1977*. Springer-Verlag, Berlin.

Reichl, G. (1998). Dynamische Simulation und Optimierung einer Kläranlage nach dem Belebungsverfahren. Diploma Thesis 200–97 D–035 (in German), Technical University of Ilmenau.

Reichl, G., Franke, R., and Puta, H. (1998). Object-oriented modeling, simulation and optimization of a waste water treatment plant. In *Proceedings ESS'98, European Simulation Symposium*, Nottingham, Great Britain.

Reinisch, K. (1986). Systemanalyse und Steuerung komplexer Prozesse: Probleme, Lösungswege, industrielle und nichtindustrielle Anwendungen. *MSR*, 29(5):194–207. (Plenary presentation at the 30. International Scientific Colloquium at TH Ilmenau, IWK'85).

Schittkowski, K. (1983). On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function. *Math. Operationsforschung u. Statist., Ser. Optimization*, 14(2):197–216.

Schittkowski, K. (1987). *More Test Examples for Nonlinear Programming Codes*. Number 282 in Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin.

Steinbach, M. (1995). *Fast Recursive SQP Methods for Large-Scale Optimal Control Problems*. PhD thesis, University of Heidelberg, Germany.

Steward, D. E. and Leyk, Z. (1994). *Meschach: Matrix Computations in C*, volume 32 of *The Centre for Mathematics and its Application*. The Australian National University.

Tone, K. (1983). Revisions of constraint approximations in the successive QP method for nonlinear programming problems. *Mathematical Programming*, 26:144–152.

Wright, S. (1996). *Primal-Dual Interior-Point Methods*. SIAM.

Wright, S. J. (1993). Interior point methods for optimal control of discrete time systems. *J. Optim. Theory and Applics.*, 77(1):161–187.

Appendix A

Interface Elements

Table A.1 shows the prepositions used to distinguish between several modules.

Table A.1: Prepositions for names of interface elements and Tcl procedures

Preposition	Addressed module or library
`prg_`	program to solve
`sqp_`	SQP solver
`qp_`	QP solver
`mat_`	matrix solver
`omu_`	Omuses library
`hqp_`	HQP library
`m_`	Meschach library

It is important to access interface elements for solver configuration in the right order. A module must be selected before modifying its local settings. To choose, for instance, Powell's SQP algorithm with Mehrotras QP algorithm and block-wise matrix factorization for a multistage problem, one has to specify exactly in this order:

```
sqp_solver    Powell
sqp_qp_solver Mehrotra
qp_mat_solver LQDOCP
```

(see also the access paths in Figure 2.1 on page 27)

A.1 Omuses interface elements

The following interface elements are introduced by Omuses:

Name	Type	Explanation
`prg_name`	Module	the programming problem to treat, i.e. an implementation of `Omu_Program`
`prg_K`	Int	number of stages (default: 0)
`prg_KK`	Int	number of sample periods (default: 0)
`prg_ts`	FloatVec	vector of sample time points (default: $(0.0)^T$)
`prg_ks`	IntVec	index vector into `ts` to determine communication time points (default: $(0)^T$)
`prg_nxs`	IntVec	number of optimization variables that represent states per stage
`prg_nus`	IntVec	number of optimization variables that represent controls per stage
`prg_fscale`	Float	scaling factor for the objective function (default: 1.0)
`prg_ad`	Bool	use of automatic differentiation for the calculation of Jacobians and sensitivity matrices (default: 1)
`prg_integrator`	Module	`RKsuite` – variable stepsize method `RK4` – fixed stepsize method (default: `RKsuite`, method 2)
`prg_int_rtol`	Float	relative integration error for variable stepsize methods (default: 1e-6)
`prg_int_atol`	Float	absolute integration error for variable stepsize methods (default: 1e-12)
`prg_res_evals`	Int	total number of residuum evaluations, that is calls to `continuous()`
`prg_setup_stages`	Method	setup the stages of the optimization problem – may be called before `prg_setup` – (see subsection 1.3.1 and example `CranePar`, subsection 1.5.2)
`prg_simulate`	Method	perform an initial-value simulation and store the results in the vector of optimization variables – may be called after `prg_setup` – (see subsection 1.3.4 and example `Crane`, subsection 1.5.2)
`omu_version`	String	contains "X.Y", with X and Y are the major and the minor version numbers, respectively

A.2 CUTE interface elements

An interface to the Constrained and Unconstrained Testing Environment (CUTE) is part of the HQP library. The optimization problem under consideration can be configured with:

Name	Type	Explanation
`prg_name`	Module	The program to solve, i.e. an implementation of `Hqp_SqpProgram`. Available modules are: `CUTE` – a program formulated in the standard input format SIF `CUTE_ST` – a program formulated in the standard input format SIF, which may be stretched by the introduction of dummy variables in order to allow partitioned BFGS update
`prg_stretch`	Bool	– only available with `CUTE_ST` – stretch a program by introducing dummy variables (default: 1)
`prg_hela_init`	Bool	initialize the sparsity structure and starting values for the Lagrangian Hessian during the problem setup (default: 1)
`prg_hela`	Bool	use the analytical Lagrangian Hessian, as provided with the problem (default: 0)
`prg_write_soln`	Command	Write the solution file. The option `-nosoln` exhibits the output of variable values. An optional argument string is written on the top of the file (e.g. the solver return status).

A.3 HQP solver interface elements

A.3.1 Solver configuration

Interface elements for solver parameters are:

Name	Type	Explanation
`sqp_solver`	Module	SQP algorithm; available solvers are: `Powell` – Powell's algorithm `Schittkowski` – Schittkowski's algorithm (default: `Powell`)
`sqp_eps`	Float	limit for the stopping criterion (default: 1e-6)
`sqp_max_iters`	Int	limit for the iteration counter (default: 500, overwritten to 9999 for CUTE)
`sqp_max_inf_iters`	Int	maximal accepted number of infeasible iterations (default: 10)
`sqp_min_alpha`	Float	lower limit for the step length (default: 1e-10)
`sqp_watchdog_start`	Int	– only available with `Powell` – iteration to start with watchdog algorithm (default: 10)
`sqp_watchdog_credit`	Int	– only available with `Powell` – number of "bad" iterations until backtracking and regular step are performed (default: 0, i.e. watchdog is disabled)
`sqp_watchdog_logging`	Bool	write watchdog log to stdout (default: 0)
`sqp_hela`	Module	Method for the Lagrangian Hessian approximation. Available modules are: `BFGS` – partitioned BFGS update with Powell's damping `DScale` – numerical approximation with a diagonal matrix `Gerschgorin` – modification of a user-provided Lagrangian Hessian for positive definiteness according (2.9) (default: `BFGS`, overwritten to `DScale` for CUTE)
`sqp_hela_scale`	Bool	initialize Hessian with diagonal matrix based on finite differences at starting point (else initialize with identity matrix) (default: 1)
`sqp_hela_eps`	Float	ϵ in (2.9) (default: 1e-8)

sqp_hela_eigen_control	Bool	– only available with BFGS – control of positiv definite Hessian blocks based on eigenvalues (default: 1)
sqp_qp_solver	Module	the QP solver to use; available solvers are: Mehrotra – Mehrotra's algorithm Franke – original HQP algorithm (default: Franke)
qp_eps	Float	limit for the stopping criterion (default: 1e-10)
qp_max_iters	Int	limit for the iteration counter (default: 250, overwritten to 999 for CUTE)
qp_mat_solver	Module	the used matrix solver; available are: SpBKP – solution of the system (2.21) RedSpBKP – solution of the reduced system (2.21) SpSC – sparce Schur complement method LQDOCP – block-wise solution of multistage problems (default: RedSpBKP)

A.3.2 Retrieving variable values and controlling execution

Interface elements for retrieving variable values are:

Name	Type	Explanation
prg_f	Float	objective value f
prg_x	FloatVec	vector $\mathbf{x}$ of variables
sqp_y	FloatVec	vector $\mathbf{y}$ of Lagrange multipliers
sqp_z	FloatVec	vector $\mathbf{z}$ of Lagrange multipliers
sqp_iter	Int	iteration counter
sqp_inf_iters	Int	number of infeasible iterations
sqp_alpha	Float	step length determined by line-search
sqp_norm_s	Float	maximum norm of the current solution of the quadratic subproblem
sqp_norm_inf	Float	maximal violation of a constraint
sqp_norm_grd_L	Float	maximum norm of the gradient of the Lagrangian

`sqp_sQs`	Float	the product $\mathbf{s}^T\mathbf{Q}\mathbf{s}$
`sqp_xQx`	Float	the product $\mathbf{x}^T\mathbf{Q}\mathbf{x}$
`qp_iter`	Int	iteration counter
`qp_result`	String	subproblem optimization result; possible values are: `optimal` – Constraints and the optimality criterion, i.e. $\mathbf{z}^T\mathbf{w} < \epsilon$, are fulfilled. `suboptimal` – Constraints are violated, but the solver can't find a better solution (see the discussion in the text). `degenerate` – The problem can not be solved numerically, which may happen, e.g., for linearly dependent equality constraints. `feasible` – Constraints are fulfilled, but not the optimality criterion. `infeasible` Constraints are violated. The results `feasible` and `infeasible` are only returned, if the iteration counter limit is reached.
`mat_sbw`	Int	the semibandwidth of the RCM-ordered coefficient matrix in (2.21) or (2.22)

The QP result '`suboptimal`' was introduced, because linear approximations of nonlinear constraints may give a quadratic subproblem without feasible solution, even though a solution of the nonlinear problem exists. The currently implemented SQP solvers perform those infeasible steps, until the limit `sqp_max_inf_iters` is reached. From none of the results can be concluded that the nonlinear problem has no solution.

Interface elements for controlling the solution process are:

Name	Type	Explanation
`prg_setup`	Command	allocate data structures for a program and initialize variable values
`sqp_init`	Command	initialize the solver
`sqp_qp_update`	Command	approximate a subproblem for the current iterate $\mathbf{x}^k$
`sqp_qp_solve`	Command	solve the subproblem
`sqp_step`	Command	perform a step based on the solution of the subproblem
`prg_qp_dump`	Command	The current quadratic subproblem is written to a file. An argument is taken for the file name.

Further interface elements are:

Name	Type	Explanation
`m_version`	Command	print a summary of the Meschach library version and modifications to stdout
`hqp_version`	String	contains "X.Y", with X and Y are the major and the minor version numbers, respectively

A.4 Tcl procedures

The following Tcl procedures are provided with the CUTE interface to HQP in the file `hqp_cute.tcl`:

Name	Type	Explanation
`hqp_config`	Procedure	Configure HQP for the solution of a specific problem.
`hqp_options`	Procedure	Read the environment variable `HQP_OPTIONS`, that is supposed to contain a list of the form (in EBNF): *name*`=`*value*`(,`*name*`=`*value*`)*` Apply and report user-specified settings.
`hqp_exit`	Procedure	Write the solution file and exit HQP (see also the `hqp_exit` provided with the HQP library).

The following Tcl procedures are provided with the the HQP library in the file `hqp.tcl`:

Name	Type	Explanation
`hqp_solve`	Procedure	Solve a problem and generate a short output on the solution process.
`hqp_exit`	Procedure	Exit HQP. This procedure is evaluated in the events of successful termination or a received signal interrupt (`kill -2`, `^C`). The procedure may be overwritten to perform something before termination.

The following Tcl procedures are provided with the the Omuses demo collection in the file `omu.tcl`:

Name	Type	Explanation
`omu_k_times`	Procedure	generate a vector of start/end times of all stages
`omu_read_plt`	Procedure	read an OmSim-like data file into the global array `data`
`omu_write_plt`	Procedure	write current `prg_x` into an OmSim-like data file (this procedure only works for a constant number of variables per stage)
`omu_plot`	Procedure	plot results into a BLT graph (this procedure only works for a constant number of variables per stage)

Appendix B

Omuses examples

B.1 Implementations of TP383 and TP383omu

```
#include "Prg_TP383.h"
#include <If_Class.h>

// propagate the class to the command interface
IF_CLASS_DEFINE("TP383", Prg_TP383, Omu_Program);

//------------------------------------------------------------------------
static double a[] = {
  12842.275, 634.25, 634.25, 634.125, 1268.0, 633.875, 633.75,
  1267.0, 760.05, 633.25, 1266.25, 632.875, 394.46, 940.838
};
static double c[] = {
  5.47934, 0.83234, 0.94749, 1.11082, 2.64824, 1.55868, 1.73215,
  3.90896, 2.74284, 2.60541, 5.96184, 3.29522, 1.83517, 2.81372
};

//------------------------------------------------------------------------
void Prg_TP383::setup(int k, Omu_Vector &x, Omu_Vector &u, Omu_Vector &cns)
{
  // allocate optimization variables
  x.alloc(14);

  // bounds on variables and initial values
  for (int i = 0; i < 5; i++) {
    x.min[i] = 0.0;
    x.max[i] = 0.04;
    x.initial[i] = 0.01;
  }
  for (int i = 5; i < 14; i++) {
    x.min[i] = 0.0;
    x.max[i] = 0.03;
    x.initial[i] = 0.01;
  }

  // one equality constraint
  cns.alloc(1);
  cns.min[0] = cns.max[0] = 1.0;
}

//------------------------------------------------------------------------
void Prg_TP383::update(int kk, const adoublev &x, const adoublev &u,
                       adoublev &f, adouble &f0, adoublev &cns)
{
  for (int i = 0; i < 14; i++) {
    f0 += ::a[i] / x[i];
    cns[0] += ::c[i] * x[i];
  }
}
```

```
#include "Prg_TP383omu.h"

IF_CLASS_DEFINE("TP383omu", Prg_TP383omu, Omu_Program);

//------------------------------------------------------------------------
void Prg_TP383omu::setup_stages(IVECP ks, VECP ts)
{
  // initialize ks and ts and set _K = _KK = 14
  stages_alloc(ks, ts, 14, 1);
}

//------------------------------------------------------------------------
void Prg_TP383omu::setup(int k, Omu_Vector &x, Omu_Vector &u, Omu_Vector &)
{
  // allocate optimization variables
  x.alloc(1);
  if (k < _K)
    u.alloc(1);

  // bounds on variables and initial values
  if (k < 5) {
    u.min[0] = 0.0;
    u.max[0] = 0.04;
    u.initial[0] = 0.01;
  }
  else if (k < _K) {
    u.min[0] = 0.0;
    u.max[0] = 0.03;
    u.initial[0] = 0.01;
  }

  // initial and final state constraints
  if (k == 0) {
    x.min[0] = x.max[0] = 0.0; // s^0
    x.initial[0] = 0.0;
  }
  else if (k == _K) {
    x.min[0] = x.max[0] = 1.0; // s^K
  }
}

//------------------------------------------------------------------------
void Prg_TP383omu::update(int kk, const adoublev &x, const adoublev &u,
                          adoublev &f, adouble &f0, adoublev &)
{
  if (kk < _KK) {
    f0 = ::a[kk] / u[0];
    f[0] = x[0] + ::c[kk] * u[0];
  }
}
```

B.2 Container crane

B.2.1 Omola code for the model

```
Crane isa Model with
  u isa ControlInput with control.default := -1.0; end;

  Fscale isa Parameter with default := 1000.0; end;     % [N]
  ml isa Parameter with default := 4000.0; end;         % [kg]
  md isa Parameter with default := 1000.0; end;         % [kg]
  g isa Parameter with default := 9.81; end;            % [m/s2]
  l isa Parameter with default := 10.0; end;            % [m]

  s, v, phi, omega isa Variable;

  mdl, sinphi, den type real;

  mdl = md + ml;
  sinphi = sin(phi);
  den = md + ml * sinphi^2;

  s' = v;
  v' = (0.5 * ml * g * sin(2*phi)
        + ml * l * omega^2 * sinphi
        + u * Fscale) / den;
  phi' = omega;
  omega' = (-mdl * g * sinphi
            - 0.5 * ml * l * omega^2 * sin(2*phi)
            - u * Fscale * cos(phi)) / (l * den);
end;
```

B.2.2 Automatically generated C code fragments

```
    // bound parameters and implicit discrete part
    mdl = md + ml;

  // dynamic model equations
  u_control = u[0];
  sinphi = sin(phi);
  den = md + ml*pow(sinphi, 2);
  xp[offs+0] = omega;
  xp[offs+1] = -(mdl*g*sinphi + 0.5*ml*l*pow(omega, 2)*sin(2*phi)
 + u_control*Fscale*cos(phi))/(l*den);
  xp[offs+2] = (0.5*ml*g*sin(2*phi) + ml*l*pow(omega, 2)*sinphi
 + u_control*Fscale)/den;
  xp[offs+3] = v;

  // state assignments
  phi = x[offs+0];
```

```
omega = x[offs+1];
v = x[offs+2];
s = x[offs+3];

  // initial state constraints
  x.min[offs+0] = x.max[offs+0] = 0.0;        // phi
  x.min[offs+1] = x.max[offs+1] = 0.0;        // omega
  x.min[offs+2] = x.max[offs+2] = 0.0;        // v
  x.min[offs+3] = x.max[offs+3] = 25.0;       // s

  // initial states
  x.initial[offs+0] = 0.0;    // phi
  x.initial[offs+1] = 0.0;    // omega
  x.initial[offs+2] = 0.0;    // v
  x.initial[offs+3] = 25.0;   // s

// default values for parameters
Fscale = 1000.0;
g = 9.81;
l = 10.0;
md = 1000.0;
ml = 4000.0;

// interface elements for unbound variables
_ifList.append(new If_Float("prg_Fscale", &Fscale));
_ifList.append(new If_Float("prg_g", &g));
_ifList.append(new If_Float("prg_l", &l));
_ifList.append(new If_Float("prg_md", &md));
_ifList.append(new If_Float("prg_ml", &ml));

// model inputs and parameters
double Fscale;
double g;
double l;
double md;
double mdl;
double ml;

// dynamic model variables
adouble den, omega, phi, s, sinphi, u_control, v;

# interface elements for unresolved variables
```

Appendix C

Copyright

Omuses and HQP are free software according to the conditions of the GNU LIBRARY GENERAL PUBLIC LICENSE (LGPL), Version 2 (see below).

Following contributions with partly different copyrights are included:

Meschach –
matrix library by D.E. Steward and Z. Leyk (version 1.2b)

ADOL-C –
automatic differentiation by A. Griewank et al (version 1.7)

Mehrotra –
Mehrotra's interior point algorithm implemented by E. Arnold

LQDOCP –
block-oriented matrix solver by E. Arnold

SpSC –
sparse Schur complement matrix solver by H. Linke

OdeTs –
ODE integration using ADOL-C by H. Linke

RKsuite –
ODE integration by R.W. Brankin et al (release 1.0)

qsort –
quick sort routine from NetBSD

gmalloc –
GNU malloc routines

Meschach

NetBSD

Omuses and HQP

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent

and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and

distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of

that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!