Chapter 10
First Approach to Algencan

Algencan is a Fortran subroutine for solving constrained optimization problems using the Augmented Lagrangian techniques described in this book. Understanding the Augmented Lagrangian principles will help you to make a good use of Algencan. (Analogous statements are valid for every numerical software.) Algencan involves many parameters with crucial influence in the algorithmic behavior. Most of them assume predetermined default values and you do not need to think about them in a first approach. Eventually, we will explain how to modify these parameters too.

Trained users should be able to obtain much better results using constrained optimization software than will untrained users. The software is your car, but you are the driver. A good use of Algencan usually overtakes a naive use to several orders of magnitude. Clever exploitation of the software capabilities may transform failures into remarkable successes.

Correct choices of parameters make a big difference. It is common that users change initial points after a presumed failure but it is less frequent to change algorithmic parameters when, for example, the local minimizer obtained is not satisfactory for practical purposes. We strongly encourage algorithmic parameter variations with at least the same intensity that we encourage changing initial approximations. Families of problems (perhaps a family to which your problem belongs) may be satisfactorily solved employing some set of algorithmic parameters that could not be adequate for other families.

Throughout this chapter, you will see that, to use Algencan, you need to code subroutines that compute functions and derivatives. In fact, you can avoid writing codes for computing derivatives (in which case Algencan will employ finite differences) but this is not recommended in terms of efficiency.[1]

10.1 ▪ Problem definition

In this chapter the notation is slightly different from that used in previous chapters. In the theoretical chapters we employed a notation that favors understanding mathematical definitions and proofs, whereas here the notation is more compatible with existing codes and subroutines. For example, instead of using $h(x) = 0$ and $g(x) \leq 0$ for equality and inequality constraints, respectively, we use $c_j(x)$ with $j \in E$ for functions that define equality constraints and $c_j(x)$ with $j \in I$ for inequality constraints. Equality constraints are assumed to be of the form $c_j(x) = 0$, while inequality constraints are assumed to be

[1]Algencan enjoys the benefits of first- and second-order derivatives computed by automatic differentiation tools through its AMPL and CUTEst interfaces.

of the form $c_j(x) \le 0$. An inequality constraint of the form $c_j^\ell \le c_j(x) \le c_j^u$ must be rewritten as (a) $c_j(x) - c_j^u \le 0$ and $c_j^\ell - c_j(x) \le 0$, or (b) $c_j(x) - s = 0$ and $c_j^\ell \le s \le c_j^u$, where s is a new auxiliary variable. The general form of the problems tackled by Algencan is then given by

$$\begin{array}{ll} \text{Minimize} & f(x) \\ \text{subject to} & c_j(x) = 0,\ j \in E, \\ & c_j(x) \le 0,\ j \in I, \\ & \ell \le x \le u, \end{array} \tag{10.1}$$

where the sets of indices E and I are such that $E \cap I = \emptyset$, $E \cup I = \{1, 2, \dots, m\}$, and f and $c_1, \dots, c_m$ are real valued functions in the n-dimensional space. The vectors ℓ and u (lower and upper bounds of the variables, respectively) are Algencan parameters, as well as the number of variables n, the number of constraints m, and the sets E and I. Functions f and c_j, $j = 1, \dots, m$, and, optionally, their first and second derivatives, must be coded by the user. Note that the definition (10.1) involves only the case in which the lower-level set Ω is defined by a box $\ell \le x \le u$.

For further reference, we restate the definition of the Lagrangian and Augmented Lagrangian functions given in (4.2) and (4.3), respectively, using the notation that will be adopted from this chapter on. The Lagrangian function can be restated as

$$\mathscr{L}(x, \lambda) = f(x) + \sum_{j \in E \cup I} \lambda_j c_j(x), \tag{10.2}$$

while the Augmented Lagrangian function can be restated as

$$L_\rho(x, \lambda) = f(x) + \sum_{j \in E \cup I} \mathscr{P}_\rho(c_j(x), \lambda_j), \tag{10.3}$$

where

$$\mathscr{P}_\rho(c_j(x), \lambda_j) = \begin{cases} c_j(x)\left(\lambda_j + \frac{1}{2}\rho c_j(x)\right) & \text{if } j \in E \text{ or } \lambda_j + \rho c_j(x) > 0, \\ -\frac{1}{2}\lambda_j^2/\rho & \text{otherwise.} \end{cases}$$

Note that, from now on, Lagrange multipliers will be denoted by $\lambda \in \mathbb{R}^m$, independently of being associated with equality or inequality constraints.

10.2 ▪ Parameters of the subroutine Algencan

We assume that the user is reasonably familiar with Fortran. The subroutine Algencan may be called from any main code or any other subroutine provided (coded) by the user. You can use Algencan to solve a single problem or a sequence of problems that may appear in your application. As a Fortran subroutine, Algencan has a rather large list of parameters. As in the case of any other optimization code, the behavior of Algencan depends on the choice of the parameters and the correct coding of the user-provided subroutines. You must provide all the input parameters, the names and meaning of which will be given below.

The prototype of Algencan is as follows:

```
subroutine algencan(fsub,gsub,hsub,csub,jacsub,hcsub,fcsub,gjacsub, &
     gjacpsub,hlsub,hlpsub,jcnnzmax,hnnzmax,epsfeas,epsopt,efstain, &
     eostain,efacc,eoacc,outputfnm,specfnm,nvparam,vparam,n,x,l,u,   &
     m,lambda,equatn,linear,coded,checkder,f,cnorm,snorm,nlpsupn,    &
     inform)
```

Therefore, the program that calls Algencan must declare the specific type of each parameter (double precision, integer, logical, or character) and must include a statement of the following form:

```
call algencan(myevalf,myevalg,myevalh,myevalc,myevaljac,myevalhc,   &
     myevalfc,myevalgjac,myevalgjacp,myevalhl,myevalhlp,jcnnzmax,   &
     hnnzmax,epsfeas,epsopt,efstain,eostain,efacc,eoacc,outputfnm,  &
     specfnm,nvparam,vparam,n,x,l,u,m,lambda,equatn,linear,coded,   &
     checkder,f,cnorm,snorm,nlpsupn,inform)
```

We usually talk about input and output parameters, independent of the existence of an ontological difference between them in the programming language at hand. (The difference exists in Fortran 90, but it does not exist in Fortran 77.) In the calling sequence, `myevalf`, `myevalg`, `myevalh`, `myevalc`, `myevaljac`, `myevalhc`, `myevalfc`, `myevalgjac`, `myevalgjacp`, `myevalhl`, `myevalhlp`, `jcnnzmax`, `hnnzmax`, `epsfeas`, `epsopt`, `efstain`, `eostain`, `efacc`, `eoacc`, `outputfnm`, `specfnm`, `nvparam`, `vparam`, `n`, `l`, `u`, `m`, `equatn`, `linear`, `coded`, and `checkder` are input parameters. Parameters `f`, `cnorm`, `snorm`, `nlpsupn`, and `inform` are output parameters. The remaining ones, `x` and `lambda`, are, at the same time, input and output parameters.

10.2.1 ▪ Brief description of the input parameters

The subroutines denominated in the calling program as `myevalf`, `myevalg`, `myevalh`, `myevalc`, `myevaljac`, `myevalhc`, `myevalfc`, `myevalgjac`, `myevalgjacp`, `myevalhl`, and `myevalhlp` might be coded by the user to represent the functions that define the problem. The parameters `jcnnzmax`, `hnnzmax`, `epsfeas`, `epsopt`, `efstain`, `eostain`, `efacc`, `eoacc`, `outputfnm`, `specfnm`, `nvparam`, `vparam`, `n`, `l`, `u`, `m`, `equatn`, `linear`, `coded`, and `checkder` are input parameters with different degrees of relevance.

The parameters related to the description of the problem are the number of variables `n`, the number of constraints `m`, the vector of lower bounds `l` and the vector of upper bounds `u`, a logical vector `equatn` that defines which constraints are equalities and which are inequalities, and a logical vector `linear` that says whether each constraint is linear. The logical vector `coded` indicates which functional subroutines were effectively coded by the user. Finally, `checkder` is a logical variable by means of which you may express your desire for checking the correctness of coded derivatives. `jcnnzmax` must store an upper bound on the number of nonnull elements in (more precisely, the number of triplets used to store the sparse representation of) the sparse Jacobian of the constraints. `hnnzmax` must be an upper bound for the sum of the number of (triplets required to represent the) nonnull elements in the lower triangles of the Hessian of the objective function, the Hessians of the constraints, and the matrix $\sum_{j=1}^{m} \nabla c_j(x) \nabla c_j(x)^T$. The meaning of parameter `hnnzmax` may vary depending on the subroutines coded by the user and the method used to solve the Augmented Lagrangian subproblems.

The parameters `epsfeas` and `epsopt` should be small positive numbers used by Algencan to stop the execution declaring feasibility and optimality, respectively. Parameters `efstain` and `eostain` are related to feasibility and optimality tolerances, respectively, and are used by Algencan to stop the execution declaring that an infeasible stationary point of the sum of the squared infeasibilities was found. Their values should depend on whether the user is interested in stopping the execution of Algencan at this type of point. Parameters `efacc` and `eoacc` are feasibility and optimality levels, below which a Newton-based acceleration process is launched.

Parameter `ouputfnm` is a string that may contain the name of an output file. Declaring `outputfnm = ''` in the calling program indicates that no output file is required. `nvparam`, `vparam`, and `specfnm` are parameters related to the setting of additional (or implicit) input parameters of Algencan.

10.2.2 ▪ Tolerances to declare convergence: `epsfeas` and `espsopt`

The input parameters `epsfeas` and `epsopt` are double precision values related to the Algencan main stopping criterion and correspond to the *feasibility tolerance* $\varepsilon_{\text{feas}}$ and to the *optimality tolerance* ε_{opt}, respectively. Roughly speaking, Algencan declares convergence when feasibility has been obtained with tolerance $\varepsilon_{\text{feas}}$ and optimality has been obtained with tolerance ε_{opt}. It is highly recommended that after returning from Algencan, you test your own criteria of feasibility and optimality, since you may not be happy with the ones adopted by Algencan. Below we give a more detailed description of the meaning of $\varepsilon_{\text{feas}}$ and ε_{opt}. You may skip this explanation when first reading this chapter.

Scaling and stopping

Algencan considers a scaled version of problem (10.1) given by

$$\begin{array}{ll} \text{Minimize} & w_f f(x) \\ \text{subject to} & w_{c_j} c_j(x) = 0,\ j \in E, \\ & w_{c_j} c_j(x) \leq 0,\ j \in I, \\ & \ell \leq x \leq u, \end{array} \tag{10.4}$$

where w_f and w_{c_j}, $j = 1, \dots, m$, are scaling factors such that $0 < w_f \leq 1$ and $0 < w_{c_j} \leq 1$, $j = 1, \dots, m$. By default, the scaling factors are computed as

$$\begin{aligned} w_f &= 1/\max(1, \|\nabla f(x^0)\|_\infty), \\ w_{c_j} &= 1/\max(1, \|\nabla c_j(x^0)\|_\infty), \quad j = 1, \dots, m, \end{aligned} \tag{10.5}$$

where x^0 is the initial estimation to the solution given by the user. Instructions on how to modify these scaling factors adopted by Algencan will be given in Section 12.5.

Let $(x^k, \bar{\lambda}^k)$ be an iterate of primal and dual variables and, following (4.7) and (4.8) applied to the scaled problem (10.4), define

$$\lambda_j^{k+1} = \begin{cases} \bar{\lambda}_j^k + \rho_k w_{c_j} c_j(x^k) & \text{if } j \in E \text{ or } \bar{\lambda}_j^k + \rho_k w_{c_j} c_j(x^k) \geq 0, \\ 0 & \text{otherwise} \end{cases}$$

for $j = 1, \dots, m$. The pair (x^k, λ^{k+1}) is considered an approximate solution to (10.1) when

$$\left\| P_{[\ell,u]}\left(x^k - \left[w_f \nabla f(x^k) + \sum_{j=1}^m \lambda_j^{k+1} w_{c_j} \nabla c_j(x^k) \right] \right) - x^k \right\|_\infty \leq \varepsilon_{\text{opt}}, \tag{10.6}$$

$$\max\left\{ \max_{j\in E}\{|w_{c_j} c_j(x^k)|\}, \max_{j \in I}\{|\min\{-w_{c_j} c_j(x^k), \lambda_j^{k+1}\}|\} \right\} \leq \varepsilon_{\text{feas}}, \tag{10.7}$$

and

$$\max\left\{ \max_{j\in E}\{|c_j(x^k)|\}, \max_{j \in I}\{c_j(x^k)_+\} \right\} \leq \varepsilon_{\text{feas}}, \tag{10.8}$$

where in (10.6), $P_{[\ell,u]}$ represents the Euclidean projection operator onto the box $\{x \in \mathbb{R}^n \mid \ell \le x \le u\}$. Conditions (10.6), (10.7) say that (x^k, λ^{k+1}) is an approximate stationary pair of the *scaled* problem (10.4), while (10.8) says that x^k also satisfies the required feasibility tolerance for the original *unscaled* problem (10.1).[2] Fulfillment of the stopping criterion (10.6)–(10.8) is reported by Algencan by returning `inform = 0`.

10.2.3 ▪ Tolerances to declare convergence to an infeasible point: `efstain` and `eostain`

In addition to the main stopping criterion described in the previous subsection, there is another stopping criterion related to the convergence to an infeasible point. This stopping criterion considers two additional double precision parameters, named `efstain` and `eostain`, that are associated with the tolerances $\varepsilon_{\text{fstain}}$ and $\varepsilon_{\text{ostain}}$, respectively. Algencan considers that an infeasible stationary point of the infeasibility measure

$$\Phi(x) = \frac{1}{2}\left(\sum_{j\in E} w_{c_j}^2 c_j(x)^2 + \sum_{j\in I} w_{c_j}^2 c_j(x)_+^2\right) \tag{10.9}$$

was found if the current point x^k is such that $\ell \le x^k \le u$,

$$\max\left\{\max_{j\in E}\{|w_{c_j} c_j(x^k)|\}, \max_{j\in I}\{[w_{c_j} c_j(x^k)]_+\}\right\} > \varepsilon_{\text{fstain}}, \tag{10.10}$$

and

$$\left\| P_{[\ell,u]}\left(x^k - \left[\sum_{j\in E} w_{c_j}^2 c_j(x^k)\nabla c_j(x^k) + \sum_{j\in I} w_{c_j}^2 c_j(x^k)_+ \nabla c_j(x^k)\right]\right) - x^k \right\|_\infty \le \varepsilon_{\text{ostain}}. \tag{10.11}$$

Whether this stopping criterion should be enabled or inhibited is not a simple question. On the one hand, Algencan may be able to "visit" an iterate x^k that satisfies (10.10), (10.11) but could abandon this point and finally converge to a point that satisfies the main stopping criterion (10.6)–(10.8) associated with success. On the other hand, the previous described situation may be very time-consuming and painful, and the user may prefer to rapidly stop the optimization process if (10.10), (10.11) is satisfied, perhaps modifying the initial guess or some algorithmic parameter, and starting the optimization process all over again.

Possible values for $\varepsilon_{\text{fstain}}$ and $\varepsilon_{\text{ostain}}$, in order to enable the stopping criterion (10.10), (10.11), would be $\sqrt{\varepsilon_{\text{feas}}}$ and $\varepsilon_{\text{opt}}^{1.5}$, respectively. If an iterate x^k satisfying (10.10), (10.11) is found, Algencan stops with the diagnostic parameter `inform = 1`.

To inhibit the stopping criterion (10.10), (10.11), it would be enough to set $\varepsilon_{\text{ostain}}$ to any negative value and to leave $\varepsilon_{\text{fstain}}$ undefined. In this scenario, conditions (10.10), (10.11) are never satisfied and convergence to an infeasible point may be perceived by the occurrence of a very large value of the penalty parameter or by the exhaustion of the maximum number of iterations of Algencan.

10.2.4 ▪ Thresholds to launch the acceleration process: `efacc` and `eoacc`

The input double precision parameters `efacc` and `eoacc` correspond to feasibility and optimality tolerances $\varepsilon_{\text{facc}}$ and $\varepsilon_{\text{oacc}}$, respectively, embedded in a criterion used to launch

[2]A similar stopping criterion (similar in that it considers a scaled problem but enforces the feasibility tolerance to be satisfied independently of the scaling) is also considered by the interior-points method Ipopt [254].

the so-called acceleration process. As suggested by its name, the acceleration process aims to accelerate the convergence of Algencan. It is automatically activated when some criterion determines that the current point is close enough to a solution. Namely, the main criterion to launch the acceleration process (there are other additional criteria too) is fulfilled when a pair $(x^k, \bar{\lambda}^k)$ satisfies (10.6)–(10.7) with $\varepsilon_{\rm feas}$ and $\varepsilon_{\rm opt}$ replaced by $\max\{\sqrt{\varepsilon_{\rm feas}}, \varepsilon_{\rm facc}\}$ and $\max\{\sqrt{\varepsilon_{\rm opt}}, \varepsilon_{\rm oacc}\}$, respectively. Starting at the iteration in which this criterion is satisfied, a KKT system of the original *unscaled* problem (10.1) is built and an attempt to solve it by Newton's method is made. On success, the whole optimization method stops. Otherwise, the attempt is ignored and the Augmented Lagrangian execution continues.

Large positive values of `efacc` and `eoacc` should be used to launch the acceleration process at the early stages of Algencan. Null values may be used to launch the acceleration process when Algencan reaches half the required precision, i.e., when (10.6)–(10.7) is satisfied with $\varepsilon_{\rm feas}$ and $\varepsilon_{\rm opt}$ replaced by $\sqrt{\varepsilon_{\rm feas}}$ and $\sqrt{\varepsilon_{\rm opt}}$, respectively. To inhibit the usage of the acceleration process, parameters `efacc` and `eoacc` should both be set to any negative value.

Note that the acceleration process requires first-order and second-order derivatives to be provided by the user plus requires the availability of a linear system solver. (Algencan may use the Fortran 95 subroutines MA57, MA86, or MA97 from the HSL Mathematical Software Library [261].) A description of the acceleration process will be given in Section 12.10 and can also be found in [54]. Below, we give a brief description in order to state the stopping criterion that is satisfied when the last iterate of Algencan is found in the acceleration process. You may skip this explanation when first reading this chapter.

Acceleration process and stopping

The acceleration process deals with the only-equalities reformulation of the *unscaled* original problem (10.1) given by

$$\begin{array}{lllll} \text{Minimize} & f(x) & & & \\ \text{subject to} & c_j(x) & = 0, & j \in E, & \\ & c_j(x) + 1/2\, s_j^2 & = 0, & j \in I, & \\ & \ell_i - x_i + 1/2\,(s_i^\ell)^2 & = 0, & i = 1,\dots,n, & \\ & x_i - u_i + 1/2\,(s_i^u)^2 & = 0, & i = 1,\dots,n, & \end{array} \tag{10.12}$$

whose KKT system is given by

$$\nabla f(x) + \sum_{j \in E \cup I} \lambda_j \nabla c_j(x) - \lambda^\ell + \lambda^u = 0, \tag{10.13}$$

$$c_j(x) = 0,\ j \in E, \tag{10.14}$$

$$c_j(x) + 1/2\, s_j^2 = 0,\ j \in I, \tag{10.15}$$

$$\ell_i - x_i + 1/2\,(s_i^\ell)^2 = 0,\ i = 1,\dots,n, \tag{10.16}$$

$$x_i - u_i + 1/2\,(s_i^u)^2 = 0,\ i = 1,\dots,n, \tag{10.17}$$

$$\lambda_j s_j = 0,\ j \in I, \tag{10.18}$$

$$\lambda_i^\ell s_i^\ell = 0,\ i = 1,\dots,n, \tag{10.19}$$

$$\lambda_i^u s_i^u = 0,\ i = 1,\dots,n. \tag{10.20}$$

The acceleration process consists of solving the nonlinear system (10.13)–(10.20) by Newton's method iteratively. Based on simple strategies to identify active bound constraints and nonactive inequality constraints, reduced Newtonian linear systems (with a possible correction of its inertia) are solved at each step.

On success, the acceleration process returns $(\tilde{x}^k, \tilde{\lambda}^k)$ such that $\ell \le \tilde{x}^k \le u$, $\tilde{\lambda}_j^k \ge 0$ for all $j \in I$, and

$$\left\| P_{[\ell,u]}\left(\tilde{x}^k - \left[\nabla f(\tilde{x}^k) + \sum_{j=1}^{m} \tilde{\lambda}_j^k \nabla c_j(\tilde{x}^k) \right] \right) - \tilde{x}^k \right\|_\infty \le \varepsilon_{\text{opt}}, \tag{10.21}$$

$$\max \left\{ \max_{j \in E} \left\{ \left| c_j(\tilde{x}^k) \right| \right\}, \max_{j \in I} \left\{ \left| \min \left\{ -c_j(\tilde{x}^k), \tilde{\lambda}_j^k \right\} \right| \right\} \right\} \le \varepsilon_{\text{feas}}. \tag{10.22}$$

This means that $(\tilde{x}^k, \tilde{\lambda}^k)$ is an approximate stationary pair of the *unscaled* original problem (10.1). In this case, Algencan stops.

10.2.5 ▪ Output filename

Algencan displays on the screen information related to the parameters being used, the problem being solved, the progress of the optimization process, and some final simple statistics. The level of the detail of the displayed information depends on a couple of parameters that will be described in Section 12.2. A copy of the information displayed on the screen may be sent to an output file if desired. The parameter `outputfnm` is a string that may contain the name of the output file (restricted to a length of 80 characters), as, for example, `myalgencan.out`. If the output file is not desired, `outputfnm` should be set to an empty string, i.e., `outputfnm = ''`.

10.2.6 ▪ Setting of the additional or implicit parameters

A rather large number of additional input parameters have default values set by Algencan. The default value of any of these additional parameters (also called "implicit"), which are not part of the Algencan calling sequence, may be modified with two alternatives: (a) the specification file (whose name is given by parameter `specfnm`) or (b) the array of parameters (driven by parameters `nvparam` and `vparam`).

The (80-character-long) string parameter `specfnm` corresponds to the name of the so-called specification file. In a first run of Algencan, parameter `specfnm` may be set to an empty string, i.e., `specfnm = ''`. The other option for setting Algencan implicit parameters is to use the array of (80-character-long) strings named `vparam`. The integer parameter `nvparam` corresponds, as suggested by its name, to the number of entries of `vparam`. In a first run of Algencan, it would be enough to set `nvparam = 0`.

In either case, the modification of a default value of an implicit parameter is done by passing a string to Algencan. The string must contain a *keyword*, sometimes (but not always) followed by an additional value that may be an integer number, a real number, or a string. For example, Algencan may save the final approximation to the solution (primal and dual variables) into a file, action that *is not* done by default. In order to save the final approximation to the solution into a file named `mysolution.txt`, the string `SOLUTION-FILENAME mysolution.txt` should be passed to Algencan. This can be done in two different ways:

1. In the calling subroutine, before calling Algencan, set

   ```
   nvparam = 1
   vparam(1) = 'SOLUTION-FILENAME mysolution.txt'
   ```

2. In the calling subroutine, before calling Algencan, set the name of the specification file to, e.g., myalgencan.dat, with the command

   ```
   specfnm = 'myalgencan.dat'
   ```

   Then, create a text file in the current folder named `myalgencan.dat` and containing a line with the sentence

   ```
   SOLUTION-FILENAME mysolution.txt
   ```

These methods are equivalent and their usage is described in Section 12.1.

As can be seen in the example above, modifying the default value of an implicit parameter requires the usage of a keyword that may or may not be followed by some value (integer, real, or string), depending on the parameter whose value is being set. For completeness, Table 10.1 lists all possible keywords. If a keyword in the table appears followed by a *D*, an *I*, or an *S*, it requires an extra real, integer, or string value, respectively. If the letter is lowercase, the additional value is optional.

Table 10.1. *Keywords that may be used to set Algencan's additional or implicit parameters.*

Keyword	Additional value
SKIP-ACCELERATION-PROCESS	
LINEAR-SYSTEMS-SOLVER-IN-ACCELERATION-PROCESS	*S*
TRUST-REGIONS-INNER-SOLVER	*s*
LINEAR-SYSTEMS-SOLVER-IN-TRUST-REGIONS	*S*
NEWTON-LINE-SEARCH-INNER-SOLVER	*s*
LINEAR-SYSTEMS-SOLVER-IN-NEWTON-LINE-SEARCH	*S*
TRUNCATED-NEWTON-LINE-SEARCH-INNER-SOLVER	*s*
MATRIX-VECTOR-PRODUCT-IN-TRUNCATED-NEWTON-LS	*S*
FIXED-VARIABLES-REMOVAL-AVOIDED	
ADD-SLACKS	
OBJECTIVE-AND-CONSTRAINTS-SCALING-AVOIDED	
IGNORE-OBJECTIVE-FUNCTION	
ITERATIONS-OUTPUT-DETAIL	*I*
NUMBER-OF-ARRAYS-COMPONENTS-IN-OUTPUT	*I*
SOLUTION-FILENAME	*S*
ACCELERATION-PROCESS-ITERATIONS-LIMIT	*I*
INNER-ITERATIONS-LIMIT	*I*
OUTER-ITERATIONS-LIMIT	*I*
PENALTY-PARAMETER-INITIAL-VALUE	*D*
LARGEST-PENALTY-PARAMETER-ALLOWED	*D*

10.2.7 ▪ Variables, bounds, and constraints

The input parameters `n`, `l`, `u`, `m`, `equatn`, and `linear` are part of the problem description: `n` and `m` are integer variables, `l` and `u` are `n`-dimensional double precision arrays, and `equatn` and `linear` are `m`-dimensional logical arrays. Their meanings follow:

`n`: number of variables n.

`l`: lower bound ℓ (ℓ_i may be equal to $-\infty$ if x_i has no lower bound).

`u`: upper bound u (u_i may be equal to $+\infty$ if x_i has no upper bound).

`m`: number of constraints m.

`equatn`: `equatn(j)` must be true if the jth constraint is an equality and false if it is an inequality.

`linear`: `linear(j)` must be true if $c_j(x)$ is linear and false otherwise.

In the description of the bounds on the variables, you should set `l(i) = -1.0d+20` if $\ell_i = -\infty$ and set `u(i) = 1.0d+20` if $u_i = +\infty$. Any value smaller than -10^{20} for a lower bound or greater than 10^{20} for an upper bound would also be acceptable. These values indicate that the corresponding bound does not exist and that it should be ignored. Any other value within the open interval $(-10^{20}, 10^{20})$ is considered by Algencan as a bound constraint to be satisfied.

10.2.8 ▪ Initial approximation and solution

The parameters `x` (an n-dimensional double precision array) and `lambda` (an m-dimensional double precision array) are input/output parameters. On input, they represent the initial estimation of the primal and dual (Lagrange multipliers) variables. On output, they are the final estimations computed by Algencan. If you do not have reliable initial estimates of the Lagrange multipliers, set `lambda = 0.0d0`.

10.2.9 ▪ Checking derivatives

`checkder` is a logical input parameter that should be used to indicate whether the user would like to check the coded subroutines that compute derivatives against simple finite differences approximations. Since computing derivatives by hand is prone to error, it is recommended to set `checkder = .true.` in the first attempt to solve a problem using Algencan. In this case, previously to the optimization process, each coded subroutine that computes derivatives will be called to compute the derivatives at a random perturbation of the initial guess `x` and its output will be compared against the same derivatives computed by finite differences. Both results (analytic values given by the subroutines coded by the user and finite differences approximations) will be displayed on the screen, relative and absolute errors will be shown, and it will be left to the user's discretion whether derivatives' subroutines coded by the user appear to be delivering the correct result. If derivatives appear to be wrong, the execution should be interrupted and the code corrected, compiled, and rerun. If, after a few iterations of this correction-testing phase, derivatives appear to be correct, `checkder` should be set to false.

Note that calls to the user-provided subroutines made during this checking phase *are* taken into account in the final counting of calls to each user-provided subroutine.

Skipping this derivatives-checking phase may seem to be a time-saving shortcut to zippy users, but in the authors' experience it happens to be a big headache in most cases.

10.2.10 ▪ Subroutines coded by the user

The input parameters `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, `hcsub`, `fcsub`, `gjacsub`, `gjacpsub`, `hlsub`, and `hlpsub` correspond to the names of the subroutines that can be coded by the user to describe the problem functions and, optionally, their derivatives. They must be declared as external in the calling program or subroutine. The input logical array `coded`, with at least 11 elements, must indicate whether each subroutine was coded. Although redundant, this array aids the robust and appropriate usage of Algencan. Table 10.2 shows the correspondence between the entries of the array `coded` and the subroutines that the user may potentially provide.

Table 10.2. *Correspondence between the entries of array* `coded` *and the potentially user-supplied subroutines.*

`coded`	Name	Subroutine description
`coded(1)`	`fsub`	Objective function
`coded(2)`	`gsub`	Gradient of the objective function
`coded(3)`	`hsub`	Sparse Hessian of the objective function
`coded(4)`	`csub`	Individually computed constraints
`coded(5)`	`jacsub`	Sparse gradient of an individual constraint
`coded(6)`	`hcsub`	Sparse Hessian of an individual constraint
`coded(7)`	`fcsub`	Objective function and all constraints
`coded(8)`	`gjacsub`	Gradient of the objective function and sparse Jacobian of the constraints
`coded(9)`	`gjacpsub`	Gradient of the objective function and product of the Jacobian (or its transpose) by a given vector
`coded(10)`	`hlsub`	Sparse Hessian of the Lagrangian
`coded(11)`	`hlpsub`	Product of the Hessian of the Lagrangian by a given vector

Based on the array `coded` (i.e., based on the subroutines coded by the user to define the problem), different algorithmic options within Algencan may be available. For now, let us say that the only mandatory subroutines are `fsub` and `csub`, to compute the objective function and the constraints, respectively, or alternatively, `fcsub`, to compute both together. If the problem has no constraints (other than the bound constraints), then coding `csub` is not mandatory and coding `fsub` is the natural choice. An adequate choice of the subroutines that should be coded to represent a problem at hand is the subject of Chapter 11.

We consider now the subroutines related to the problem evaluation listed in Table 10.2. The prototypes of the subroutines are as follows:

```
subroutine fsub(n,x,f,flag)
subroutine gsub(n,x,g,flag)
subroutine hsub(n,x,hrow,hcol,hval,hnnz,lim,lmem,flag)
subroutine csub(n,x,ind,cind,flag)
subroutine jacsub(n,x,ind,jcvar,jcval,jcnnz,lim,lmem,flag)
subroutine hcsub(n,x,ind,hcrow,hccol,hcval,hcnnz, &
           lim,lmem,flag)
subroutine fcsub(n,x,f,m,c,flag)
subroutine gjacsub(n,x,g,m,jcfun,jcvar,jcval,jcnnz, &
           lim,lmem,flag)
subroutine gjacpsub(n,x,g,m,p,q,work,gotj,flag)
subroutine hlsub(n,x,m,lambda,sf,sc,hlrow,hlcol,hlval,hlnnz, &
           lim,lmem,flag)
subroutine hlpsub(n,x,m,lambda,sf,sc,p,hp,gothl,flag)
```

Subroutines fsub, gsub, and hsub should compute the objective function $f(x)$, its gradient $\nabla f(x)$, and its Hessian $\nabla^2 f(x)$, respectively. Subroutines csub, jacsub, and hcsub should compute, given a constraint index ind, $c_{\text{ind}}(x)$, $\nabla c_{\text{ind}}(x)$, and $\nabla^2 c_{\text{ind}}(x)$, respectively. Subroutine fcsub should compute the objective function $f(x)$ and all constraints $c(x)$, while subroutine gjacsub should compute the gradient of the objective function $\nabla f(x)$ and the Jacobian $J(x)$ of the constraints, defined as

$$J(x) = \begin{pmatrix} \nabla c_1(x)^T \\ \vdots \\ \nabla c_m(x)^T \end{pmatrix}. \tag{10.23}$$

Subroutine gjacpsub should compute the gradient of the objective function $\nabla f(x)$ and the product of the Jacobian of the constraints $J(x)$, or its transpose, by a given vector, depending on the value of parameter work. Finally, subroutine hlsub should compute the *scaled* Hessian of the Lagrangian given by

$$\nabla^2 \mathscr{L}(x,\lambda) = w_f \nabla^2 f(x) + \sum_{j \in E \cup I} \lambda_j w_{c_j} \nabla^2 c_j(x), \tag{10.24}$$

and subroutine hlpsub should compute the product of the *scaled* Hessian of the Lagrangian by a given vector.

The integer parameter n and the double precision n-dimensional array parameter x are input parameters for the subroutines, while the integer parameter flag is an output parameter in all cases. In the subroutines coded by the user, the output parameter flag must be used to indicate whether an exception occurred during the computation. For example, if the objective function calculation (or any other) requires a division to be done and the denominator is null, or if a square root should be taken and the radicand is negative, parameter flag must be set to any nonnull value. In this way, on return Algencan will know that the required quantity was not properly computed and a warning message will be shown to the user. Depending on the situation, the optimization process may be interrupted. If everything went well in the computations, parameter flag must be set to zero.

The Jacobian and Hessians must be stored in the well-known coordinate scheme (see, for example, [94]). In the coordinate scheme, a sparse matrix A is specified as its set of entries in the form of an unordered set of triplets (a_{ij}, i, j). The set of triplets is held in one double precision array aval and two integer arrays arow and acol. The number of entries of these three arrays should be annz. When a Hessian is required, only the lower triangle is required and any computed value above the diagonal will be ignored. For the Jacobian and Hessians, if more than one triplet is present for the same element of the matrix, the sum of the values of the duplicated triplets is considered as the element value. No order is required.

Subroutines that compute sparse matrices (Jacobian and Hessians) have an input integer parameter named lim and an output logical parameter named lmem. lim corresponds to the dimension of the arrays that play the role of arow, acol, and aval and should be used to avoid accessing elements out of the arrays' range. If the arrays' dimension is not enough to save the matrix being computed, parameter lmem (meaning "lack of memory") must be set to true. Otherwise, it must be set to false.

Summing up, we have the following:

fsub computes f $= f(x)$.

gsub computes the dense n-dimensional vector g $= \nabla f(x)$.

hsub computes the lower triangle of the sparse Hessian $\nabla^2 f(x)$. An element h_{ij} with $i \geq j$ must be stored as `hrow(k)` $= i$, `hcol(k)` $= j$, and `hval(k)` $= h_{ij}$ for some `k` between 1 and `hnnz`.

csub computes `cind` $= c_{\text{ind}}(x)$.

jacsub computes the sparse gradient $\nabla c_{\text{ind}}(x)$. An element $[\nabla c_{\text{ind}}(x)]_i$ must be stored as `jcvar(k)` $= i$ and `jcval(k)` $= [\nabla c_{\text{ind}}(x)]_i$ for some `k` between 1 and `jcnnz`.

hcsub computes the lower triangle of the sparse Hessian $\nabla^2 c_{\text{ind}}(x)$. The storage scheme is identical to the one used by `hsub` but using `hcrow`, `hccol`, `hcval`, and `hcnnz`.

fcsub computes `f` $= f(x)$ and the m-dimensional array `c` such that `c(j)` $= c_j(x)$ for $j = 1, \ldots, m$.

gjacsub computes the gradient of the objective function `g` $= \nabla f(x)$ and the Jacobian of the constraints (10.23). An element $[\nabla c_j(x)]_i$ must be saved as `jcfun(k)` $= j$, `jcvar(k)` $= i$, and `jcval(k)` $= [\nabla c_j(x)]_i$ for some `k` between 1 and `jcnnz`.

gjacpsub computes the gradient of the objective function `g` $= \nabla f(x)$. It also computes the product $q = J(x)^T p$ when `work` is equal to T or t and computes the product $p = J(x)q$ otherwise. Note that q refers to `q` and p refers to `p`; i.e., `p` and `q` play the role of input or output parameters depending on the value of `work`. Moreover, note that, by the role they play, it is clear that `p` is an m-dimensional double precision array, while `q` is an n-dimensional double precision array. `gotj` is a logical input/output parameter that is set to false by the calling subroutine every time subroutine `gjacpsub` is called by the first time with a point `x`. So, assume that your subroutine `gjacpsub` computes the Jacobian matrix (10.23) at `x`, saves it, sets `gotj = .true.`, and computes the required matrix-vector product. If, in a forthcoming call to `gjacpsub`, `gotj` is still true, then you can use the stored Jacobian matrix instead of computing it again. Moreover, in this case, if the gradient of the objective function was also stored, it can be copied into the output parameter `g` without recomputing it.

hlsub computes the lower triangle of the sparse scaled Hessian of the Lagrangian (10.24) with $x =$ `x`, $\lambda =$ `lambda`, $w_f =$ `sf`, and $w_{c_j} =$ `sc(j)` for $j = 1, \ldots, m$. The storage scheme is identical to the one used by `hsub` and `hcsub` but using `hlrow`, `hlcol`, `hlval`, and `hlnnz`.

hlpsub computes `hp` $= \nabla^2 \mathscr{L}(x,\lambda)p$, where $\nabla^2 \mathscr{L}(x,\lambda)$ is the scaled Hessian of the Lagrangian given by (10.24) and $p =$ `p`. `gothl` is a logical input/output parameter that is set to false by the calling subroutine every time subroutine `hlpsub` is called with a new set of parameters `(x,lambda,sf,sc)`. So, assume that your subroutine `hlpsub` computes the Hessian matrix (10.24) and saves it, set `gothl = .true.`, and compute the required matrix-vector product. If, in a forthcoming call to `hlpsub`, `gothl` is still true, then you can use the stored Hessian matrix instead of computing it again.

Not-coded (empty-body) subroutines

If a subroutine of the Algencan calling sequence (also listed in Table 10.2) is not coded by the user, the corresponding parameter in the calling sequence may point to a dummy subroutine, since it will never be called by Algencan. However, for example, even in the case of setting `coded(7) = .false.`, it would be a conservative choice to code subroutine `fcsub` with an empty body as follows:

```
subroutine fcsub(n,x,f,m,c,flag)
  implicit none
  integer, intent(in) :: m,n
  integer, intent(out) :: flag
  real(kind=8), intent(in) :: x(n)
  real(kind=8), intent(out) :: c(m),f
end subroutine fcsub
```

This is because it is not rare for a user to choose some subroutines to code and to set array `coded` incorrectly. If this is the case, an uncoded subroutine may be called by Algencan. To rapidly detect this case and correct the coding problem, it is recommended to include the statement `flag = -1` in all empty-body unused subroutines. Namely,

```
subroutine fcsub(n,x,f,m,c,flag)
  implicit none
  integer, intent(in) :: m,n
  integer, intent(out) :: flag
  real(kind=8), intent(in) :: x(n)
  real(kind=8), intent(out) :: c(m),f
  flag = - 1
end subroutine fcsub
```

10.2.11 ▪ Maximum sizes `jcnnzmax` and `hnnzmax`

Integer input parameters `jcnnzmax` and `hnnzmax` are used by Algencan to allocate the arrays that save the sparse Jacobian and the (lower triangle of the) "Hessians," respectively. Therefore, the user must set these parameters in the main code with upper bounds on the number of nonnull elements in the Jacobian and the Hessian matrices, respectively.

In most cases, a nonnull element a_{ij} of a matrix A (Jacobian or Hessian) is represented with a single triplet (a_{ij}, i, j), where i is the row index, j is the column index, and a_{ij} is the value of the element. In these cases, the number of triplets that are necessary to describe the whole matrix is equal to the number of nonnull elements. However, in some cases, it is easier to represent an element a_{ij} of a matrix by two triplets (b_1, i, j) and (b_2, i, j) in such a way that $a_{ij} = b_1 + b_2$. Algencan allows the user to proceed in that way and even more than two triplets can be used to represent only one element. However, it must be noted that, in those cases, `jcnnzmax` and `hnnzmax` must be upper bounds on the number of *triplets* and not on the number of *nonnull elements* of the corresponding matrices. Strictly speaking `jcnnzmax` and `hnnzmax` should be *upper bounds on the number of triplets that one uses to describe the nonnull elements of the corresponding matrices*. However, in order to avoid the pedantism of this definition, we will refer to these parameters, when this does not lead to confusion, as upper bounds on the number of nonzero elements of the corresponding matrices.

Independently of having coded `jacsub`, `gjacsub`, or none of them (in which case the Jacobian of the constraints is approximated by finite differences), `jcnnzmax` should be enough to hold the whole sparse Jacobian of the constraints. The number of nonnull elements in the Jacobian matrix may be difficult to compute when the Jacobian is not being coded. In this case, $n \times m$ is a suitable and conservative upper bound. If this amount of memory is unaffordable, a smaller quantity may be found by trial and error. `jcnnzmax` may be set to zero if the user codes `gjacpsub`.

If the user has coded subroutines `hsub` and `hcsub`, then `hnnzmax` should be enough to save, at the same time, the (lower triangle of the) Hessian matrix of the objective function and the (lower triangles of the) m Hessians of the constraints. On the other hand, if the user coded subroutine `hlsub`, then `hnnzmax` should be enough to save (the lower triangle of) the Hessian of the Lagrangian matrix.

Independently of the coded subroutines (`hsub` and `hcsub`, or `hlsub`), and in addition to the mentioned Hessians, if the method used to solve the Augmented Lagrangian subproblems is the Euclidean trust-region method (see Section 12.8), `hnnzmax` must be such that there exists enough space to save also the lower triangle of the (symmetric) matrix $\sum_{j=1}^{m} \nabla c_j(x) \nabla c_j(x)^T$. If Newton's or the truncated Newton's method is selected to solve the Augmented Lagrangian subproblems, no additional space needs to be considered when setting the value of parameter `hnnzmax`.

An upper bound on the number of (triplets required to represent the) nonnull elements of the lower triangle of matrix $\sum_{j=1}^{m} \nabla c_j(x) \nabla c_j(x)^T$ can be easily computed as $\frac{1}{2}\sum_{j=1}^{m} \texttt{jcnnzmax}_j(\texttt{jcnnzmax}_j + 1)$, where $\texttt{jcnnzmax}_j$ is the number of (triplets used to represent the) nonnull elements of the jth row of the Jacobian (gradient of the jth constraint).

If second-order derivatives were not coded by the user, `hnnzmax` may be set to zero. If the user did code `hlpsub`, `hnnzmax` may be set to zero too.

10.2.12 ▪ Output parameters `nlpsupn`, `snorm`, and `cnorm`

On output, if the final iterate is the result of an Augmented Lagrangian iteration, then Algencan returns $\texttt{x} = x^k$, $\texttt{lambda} = \lambda^{k+1}$, $\texttt{f} = f(x^k)$, `nlpsupn` equal to the left-hand side of (10.6), `snorm` equal to the left-hand side of (10.7), and `cnorm` equal to the left-hand side of (10.8), i.e.,

$$\begin{aligned}
\texttt{nlpsupn} &= \left\| P_{[\ell,u]}\left(x^k - \left[w_f \nabla f(x^k) + \sum_{j=1}^{m} \lambda_j^{k+1} w_{c_j} \nabla c_j(x^k) \right] \right) - x^k \right\|_\infty, \\
\texttt{snorm} &= \max\left\{ \max_{j\in E}\{|w_{c_j} c_j(x^k)|\}, \max_{j\in I}\{|\min\{-w_{c_j} c_j(x^k), \lambda_j^{k+1}\}|\} \right\}, \\
\texttt{cnorm} &= \max\left\{ \max_{j\in E}\{|c_j(x^k)|\}, \max_{j\in I}\{c_j(x^k)_+\} \right\}.
\end{aligned} \tag{10.25}$$

If the final iterate is the result of a successful acceleration process, Algencan returns $\texttt{x} = \tilde{x}^k$, $\texttt{lambda} = \tilde{\lambda}^k$, $\texttt{f} = f(\tilde{x}^k)$, `nlpsupn` equal to the left-hand side of (10.21), `snorm` equal to the left-hand side of (10.22), and `cnorm` equal to the left-hand side of (10.8) evaluated at $\tilde{x}^k$ and $\tilde{\lambda}^k$, i.e.,

$$\begin{aligned}
\texttt{nlpsupn} &= \left\| P_{[\ell,u]}\left(\tilde{x}^k - \left[\nabla f(\tilde{x}^k) + \sum_{j=1}^{m} \tilde{\lambda}_j^{k} \nabla c_j(\tilde{x}^k) \right] \right) - \tilde{x}^k \right\|_\infty, \\
\texttt{snorm} &= \max\left\{ \max_{j\in E}\{|c_j(\tilde{x}^k)|\}, \max_{j\in I}\{|\min\{-c_j(\tilde{x}^k), \tilde{\lambda}_j^{k}\}|\} \right\}, \\
\texttt{cnorm} &= \max\left\{ \max_{j\in E}\{|c_j(\tilde{x}^k)|\}, \max_{j\in I}\{c_j(\tilde{x}^k)_+\} \right\}.
\end{aligned} \tag{10.26}$$

The arrays `x` and `lambda` must be declared double precision with at least n and m positions, respectively, in the code that calls the subroutine Algencan. The scalars `f`, `nlpsupn`, `cnorm`, and `snorm` must be double precision scalars in the calling code.

10.2.13 ▪ Diagnostic parameter `inform`

The value of the output integer parameter `inform` is equal to 0 when Algencan stops satisfying the stopping criterion (10.6)–(10.8) or the stricter stopping criterion (10.21), (10.22), both related to success.

The returned value `inform` $= 1$ suggests that x^k is infeasible and stationary for the infeasibility measure (10.9) subject to $\ell \le x \le u$. Therefore, Algencan returns `inform` equal to 1 when a point $x^k \in [\ell, u]$ that satisfies (10.10), (10.11) is found.

The constant $\rho_{\max}$ is an additional parameter of Algencan whose default value is 10^{20}. If, at iteration k, x^k was computed considering $\rho_k > \rho_{\max}$, Algencan stops returning `inform = 2`. Algencan returns `inform = 3` when x^k was computed and $k \ge k_{\max}$, where $k_{\max}$ is an implicit parameter of Algencan whose default value is 100. Both stopping criteria may also suggest that an infeasible point x^k that is stationary for the infeasibility measure (10.9) subject to $\ell \le x \le u$ was found.

The value of $k_{\max}$ may be modified with the keyword OUTER-ITERATIONS-LIMIT (see Section 12.3), while the value of $\rho_{\max}$ may be modified with the keyword LARGEST-PENALTY-PARAMETER-ALLOWED (see Section 12.4).

10.3 ▪ A simple example

Consider the problem of finding the rectangle (centered at the origin) with smallest area within which p circles with radii $r_i = i$ can be packed without overlapping. The problem can be modeled as

$$\begin{array}{lrcccl}
\text{Minimize} & wh & & & & \\
\text{subject to} & (c_i^x - c_{i'}^x)^2 + (c_i^y - c_{i'}^y)^2 & \ge & (r_i + r_{i'})^2, & & i' > i, \\
& -w/2 + r_i & \le & c_i^x & \le \; w/2 - r_i & \text{for all } i, \\
& -h/2 + r_i & \le & c_i^y & \le \; h/2 - r_i & \text{for all } i, \\
& w, h \ge 0. & & & &
\end{array} \qquad (10.27)$$

The variables of the problem are the centers of the circles (c_i^x, c_i^y), $i = 1, \ldots, p$, the width w, and the height h of the rectangle. The objective function to be minimized is the area of the rectangle. The first set of constraints models the nonoverlapping between the circles, requesting the necessary minimum distance between their centers (distance is squared to preserve differentiability), and the remaining linear constraints say that the circles must be placed within the rectangle.

We will illustrate the usage of Algencan to solve this problem coding subroutines `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub`. Other reasonable choices would have been to code (i) `fcsub`, `gjacsub`, and `hlsub` or (ii) `fcsub`, `gjacpsub`, and `hlpsub`. Our choice of subroutines to be coded is the simplest one and hence the most appropriate for our first example. Objective function, constraints, gradients, and Hessians are all coded in separately. Case (i) would be appropriate if, for some reason, coding all the constraints at once or coding the whole Jacobian of the constraints at once brings some cost savings. The price to be paid is to code the sparse Hessian of the Lagrangian, which requires coding the sum of the sparse Hessians of the constraints plus the Hessian of the objective function. Naive approaches to this task may lead to time-consuming subroutines. Case (ii) would be recommended when the problem is such that computing individual gradients and Hessians, or computing the Jacobian of the constraints or the Hessian of the Lagrangian, is a very time-consuming task but efficient algorithms exist to compute the product of the Jacobian of the constraints or the Hessian of the Lagrangian by a given vector. An example of this situation will be given in Chapter 11.

We start by coding the definition of the problem and setting the Algencan parameters in a main program named `algencanma`, as shown in Listing 10.1. Let $p > 0$ be the number of circles to be packed. The problem has $n = 2p+2$ variables and $m = p(p-1)/2+4p$ constraints. Variables are given by $x = (c_1^x, c_1^y, c_2^x, c_2^y, \dots, c_p^x, c_p^y, w, h)^T \in \mathbb{R}^n$. All the constraints are inequalities. The first $p(p-1)/2$ constraints correspond to the nonoverlapping nonlinear constraints. The remaining constraints are linear. (See Listing 10.1.) The other subroutines are self-explanatory and are shown in Listings 10.2–10.7. Subroutines that correspond to `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub` were named `myevalf`, `myevalg`, `myevalh`, `myevalc`, `myevaljac`, and `myevalhc`, respectively, in the present example.

Subroutines `myevalf` and `myevalg` are very simple and require no further explanations. Subroutine `myevalh` is also very simple and its only highlight is that only the lower triangle of the Hessian matrix of the objective function is being computed. The lower triangle of the matrix has a single element $h_{n,n-1} = 1$. In subroutine `myevalc` (as well as in `myevaljac` and `myevalhc`), the first $p(p-1)/2$ constraints correspond to the nonoverlapping nonlinear constraints. The p constraints that follow correspond to $-w/2 + r_i - c_i^x \le 0$, $i = 1, \dots, p$. Then, p constraints correspond to $-w/2 + r_i + c_i^x \le 0$, $i = 1, \dots, p$. The constraints $-h/2 + r_i - c_i^y \le 0$, $i = 1, \dots, p$, follow and, finally, $-h/2 + r_i + c_i^y \le 0$, $i = 1, \dots, p$. Subroutines `myevaljac` and `myevalhc` compute the sparse gradient and the lower triangle of the sparse Hessian, respectively, of the `ind`th constraint. In these three subroutines related to the constraints, a subroutine named `pair` is being used. It is assumed that, given $1 \le k \le p(p-1)/2$, the subroutine implements a bijection that returns a pair (i, j) satisfying $1 \le i < j \le p$. (See Problem 10.1.) The external subroutine `drand` is the random number generator of Schrage [234].

Adding the remaining empty-body subroutines `myevalfc`, `myevalgjac`, `myevalgjacp`, `myevalhl`, and `myevalhlp`, we solved an instance of problem (10.27) with $p = 3$. Figure 10.1 shows Algencan's solution, and Figure 10.2 shows a graphical representation of the solution found.

10.3.1 ▪ Output analysis

We now analyze the output given by Algencan (Figure 10.1). The first lines correspond to a banner that indicates the version of Algencan being used, which is Algencan 3.0.0 in this example. Since Algencan is a live code, this version number is essential when reporting the performance of Algencan. Then, a sentence appears indicating that no additional-parameters default values are being modified through the array of parameters `vparam`, and another sentence indicates that the specification file is not being used either. The following phrase shows the HSL subroutines to deal with linear systems that were compiled together with Algencan and, therefore, are available for use. In this example, subroutines MA57, MA86, and MA97 for solving linear systems are present.

Then comes the so-called preamble, which indicates the values of some parameters being used by Algencan that strongly affect its performance. `firstde` stands for "first derivatives" and its possible values are "true" and "false." In the same way, `seconde` stands for "second derivatives," `truehpr` stands for "true Hessian-vector product" (where by Hessian we mean Hessian of the Augmented Lagrangian function), and their possible values are "true" and "false." In our example, since we code first and second derivatives, all those parameters are true. At this point, we skip all other parameters reported in the preamble, with the exception of `epsfeas`, `epsopt`, `efstain`, `eostain`, `efacc`,

Listing 10.1. *Main program.*

```
program algencanma
  implicit none
  ! LOCAL SCALARS
  logical :: checkder
  integer :: hnnzmax,i,inform,jcnnzmax,m,n,npairs,nvparam
  real(kind=8) :: cnorm,efacc,efstain,eoacc,eostain,epsfeas,epsopt,f,nlpsupn,seed,snorm
  ! LOCAL ARRAYS
  character(len=80) :: specfnm,outputfnm,vparam(10)
  logical :: coded(11)
  logical, pointer :: equatn(:),linear(:)
  real(kind=8), pointer :: l(:),lambda(:),u(:),x(:)
  ! COMMON SCALARS
  integer :: p
  ! COMMON BLOCKS
  common /pdata/ p
  ! FUCNTIONS
  real(kind=8) :: drand
  ! EXTERNAL SUBROUTINES
  external :: myevalf,myevalg,myevalh,myevalc,myevaljac,myevalhc,myevalfc, &
       myevalgjac,myevalgjacp,myevalhl,myevalhlp
  ! Problem data
  p = 3
  npairs = p * ( p - 1 ) / 2
  ! Number of variables
  n = 2 * p + 2
  ! Set lower bounds, upper bounds, and initial guess
  allocate(x(n),l(n),u(n))
  l(1:2*p) = - 1.0d+20
  u(1:2*p) =   1.0d+20
  l(2*p+1:n) = 0.0d0
  u(2*p+1:n) = 1.0d+20
  seed = 654321.0d0
  do i = 1,2*p
     x(i) = - 5.0d0 + 10.0d0 * drand(seed)
  end do
  x(2*p+1:n) = 10.0d0
  ! Constraints
  m = npairs + 4 * p
  allocate(equatn(m),linear(m),lambda(m))
  equatn(1:m) = .false.
  linear(1:npairs) = .false.
  linear(npairs+1:m) = .true.
  lambda(1:m) = 0.0d0
  ! Coded subroutines
  coded(1:6)  = .true.  ! fsub,gsub,hsub,csub,jacsub,hcsub
  coded(7:11) = .false. ! fcsub,gjacsub,gjacpsub,hlsub,hlpsub
  ! Upper bounds on the number of sparse-matrices non-null elements
  jcnnzmax = 4 * npairs + 2 * ( 4 * p )
  hnnzmax  = 1 + 6 * npairs + 10 * npairs + 3 * ( 4 * p )
  ! Checking derivatives?
  checkder = .false.
  ! Parameters setting
  epsfeas   =   1.0d-08
  epsopt    =   1.0d-08
  efstain   =   1.0d+20
  eostain   = - 1.0d+20
  efacc     = - 1.0d+20
  eoacc     = - 1.0d+20
  outputfnm = ''
  specfnm   = ''
  nvparam   = 0
  ! Optimize
  call algencan(myevalf,myevalg,myevalh,myevalc,myevaljac,myevalhc, &
       myevalfc,myevalgjac,myevalgjacp,myevalhl,myevalhlp,jcnnzmax, &
       hnnzmax,epsfeas,epsopt,efstain,eostain,efacc,eoacc,outputfnm,&
       specfnm,nvparam,vparam,n,x,l,u,m,lambda,equatn,linear,coded, &
       checkder,f,cnorm,snorm,nlpsupn,inform)
  deallocate(x,l,u,lambda,equatn,linear)
  stop
end program algencanma
```

eoacc, specfnm (specification filename), and outputfnm (output filename), that display the values that we set in the main program.

After the preamble, Algencan shows the number of variables, equality constraints, inequality constraints, and bound constraints. The displayed figures can be easily checked

Listing 10.2. *Subroutine evalf.*

```fortran
subroutine myevalf(n,x,f,flag)
  implicit none
  ! SCALAR ARGUMENTS
  integer, intent(in) :: n
  integer, intent(out) :: flag
  real(kind=8), intent(out) :: f
  ! ARRAY ARGUMENTS
  real(kind=8), intent(in) :: x(n)
 ! Compute objective function
  flag = 0
  f = x(n-1) * x(n)
end subroutine myevalf
```

Listing 10.3. *Subroutine evalg.*

```fortran
subroutine myevalg(n,x,g,flag)
  implicit none
  ! SCALAR ARGUMENTS
  integer, intent(in) :: n
  integer, intent(out) :: flag
  ! ARRAY ARGUMENTS
  real(kind=8), intent(in) :: x(n)
  real(kind=8), intent(out) :: g(n)
  ! Compute gradient of the objective function
  flag = 0
  g(1:n-2) = 0.0d0
  g(n-1)   = x(n)
  g(n)     = x(n-1)
end subroutine myevalg
```

Listing 10.4. *Subroutine evalh.*

```fortran
subroutine myevalh(n,x,hrow,hcol,hval,hnnz,lim,lmem,flag)
  implicit none
  ! SCALAR ARGUMENTS
  logical, intent(out) :: lmem
  integer, intent(in) :: lim,n
  integer, intent(out) :: flag,hnnz
  ! ARRAY ARGUMENTS
  integer, intent(out) :: hcol(lim),hrow(lim)
  real(kind=8), intent(in)  :: x(n)
  real(kind=8), intent(out) :: hval(lim)
  ! Compute (lower triangle of the) Hessian of the objective function
  flag = 0
  lmem = .false.
  hnnz = 1
  if ( hnnz .gt. lim ) then
     lmem = .true.
     return
  end if
  hrow(1) = n
  hcol(1) = n - 1
  hval(1) = 1.0d0
end subroutine myevalh
```

against the model of problem (10.27) for the particular case $p = 3$. Then, Algencan says that “There are no fixed variables to be removed.” In fact, Algencan removes from the problem variables x_i such that $\ell_i = u_i$. This kind of variable should not be part of any problem, but sometimes its existence simplifies the modeling process. To maintain fixed variables in the problem, without a negative impact in the problem resolution, Algencan allows the user to include such artificial variables that are then removed and do not take part in the optimization process. Besides the number of removed fixed variables, the output shows the scaling factors for the objective function and the constraints, automatically computed by Algencan. It is easy to see in the main program `algencanma`

Listing 10.5. *Subroutine evalc.*

```fortran
subroutine myevalc(n,x,ind,c,flag)
  implicit none
  ! SCALAR ARGUMENTS
  integer, intent(in) :: ind,n
  integer, intent(out) :: flag
  real(kind=8), intent(out) :: c
  ! ARRAY ARGUMENTS
  real(kind=8), intent(in) :: x(n)
  ! COMMON SCALARS
  integer :: p
  ! LOCAL SCALARS
  integer :: i,j
  ! COMMON BLOCKS
  common /pdata/ p
  ! Compute ind-th constraint
  flag = 0
  if ( 1 .le. ind .and. ind .le. p * ( p - 1 ) / 2 ) then
     call pair(p,ind,i,j)
     c = ( dble(i) + dble(j) ) ** 2 - &
           ( x(2*i-1) - x(2*j-1) ) ** 2 - ( x(2*i) - x(2*j) ) ** 2
  else if ( ind .le. p * ( p - 1 ) / 2 + p ) then
     i = ind - p * ( p - 1 ) / 2
     c = - 0.5d0 * x(n-1) + dble(i) - x(2*i-1)
  else if ( ind .le. p * ( p - 1 ) / 2 + 2 * p ) then
     i = ind - p * ( p - 1 ) / 2 - p
     c = - 0.5d0 * x(n-1) + dble(i) + x(2*i-1)
  else if ( ind .le. p * ( p - 1 ) / 2 + 3 * p ) then
     i = ind - p * ( p - 1 ) / 2 - 2 * p
     c = - 0.5d0 * x(n) + dble(i) - x(2*i)
  else
     i = ind - p * ( p - 1 ) / 2 - 3 * p
     c = - 0.5d0 * x(n) + dble(i) + x(2*i)
  end if
end subroutine myevalc
```

(Listing 10.1) that at the initial guess x^0, we have $w = h = 10$. This means that $\nabla f(x^0) = (0,\dots,0,10,10)^T$, $\|\nabla f(x^0)\|_\infty = 10$, and $w_f = 1/\max(1, \|\nabla f(x^0)\|_\infty) = 0.1$. The displayed smallest constraints' scale factor corresponds to $\min_{j=1,\dots,m}\{w_{c_j}\}$. (Its actual value cannot be computed here without showing the values of the first $n-2$ components of the initial guess x^0.)

The output described so far corresponds to a preprocessing phase. Then, the optimization subroutine is effectively called. It starts by displaying the actual number of variables (discarding removed variables) and constraints (equalities plus inequalities) of the problem to be optimized. Considering the default level of detail in Algencan's output (which corresponds to 10 and can be modified with the keyword ITERATIONS-OUTPUT-DETAIL followed by an integer value between 0 and 99, as will be explained in Section 12.2), Algencan prints a single line per iteration, starting with "iteration 0," that corresponds to the initial guess x^0. For each iteration, the line displays (from left to right) (a) the iteration number k, (b) the penalty parameter ρ_k, (c) the (unscaled) objective function value $f(x^k)$, (d) the (unscaled) infeasibility measure `cnorm` as defined in (10.25), (e) the scaled objective function $w_f f(x^k)$, (f) the scaled infeasibility measure given by

$$\max\{\max_{j\in E}\{|w_{c_j} c_j(x^k)|\}, \max_{j\in I}\{[w_{c_j} c_j(x^k)]_+\}\},$$

(g) the scaled infeasibility-complementarity measure `snorm` as defined in (10.25), (h) the sup-norm of the scaled projected gradient of the Lagrangian `nlpsupn` as defined in (10.25), (i) the sup-norm of the projected gradient of the infeasibility measure (10.9) given by the left-hand side of (10.11), and (j) the accumulated total number of inner iterations needed to solve the subproblems. The letter glued to the number of inner iterations corresponds to the termination criterion satisfied by the inner solver. C means convergence, M means

Listing 10.6. *Subroutine evaljac.*

```
subroutine myevaljac(n,x,ind,jcvar,jcval,jcnnz,lim,lmem,flag)
  implicit none
  ! SCALAR ARGUMENTS
  logical, intent(out) :: lmem
  integer, intent(in) :: ind,lim,n
  integer, intent(out) :: flag,jcnnz
  ! ARRAY ARGUMENTS
  integer, intent(out) :: jcvar(lim)
  real(kind=8), intent(in) :: x(n)
  real(kind=8), intent(out) :: jcval(lim)
  ! COMMON SCALARS
  integer :: p
  ! LOCAL SCALARS
  integer :: i,j
  ! COMMON BLOCKS
  common /pdata/ p
  ! Compute gradient of the ind-th constraint
  flag = 0
  lmem = .false.
  if ( 1 .le. ind .and. ind .le. p * ( p - 1 ) / 2 ) then
     call pair(p,ind,i,j)
     jcnnz = 4
     if ( jcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     jcvar(1) = 2 * i - 1
     jcval(1) = - 2.0d0 * ( x(2*i-1) - x(2*j-1) )
     jcvar(2) = 2 * j - 1
     jcval(2) =   2.0d0 * ( x(2*i-1) - x(2*j-1) )
     jcvar(3) = 2 * i
     jcval(3) = - 2.0d0 * ( x(2*i) - x(2*j) )
     jcvar(4) = 2 * j
     jcval(4) =   2.0d0 * ( x(2*i) - x(2*j) )
  else if ( ind .le. p * ( p - 1 ) / 2 + p ) then
     i = ind - p * ( p - 1 ) / 2
     jcnnz = 2
     if ( jcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     jcvar(1) = n - 1
     jcval(1) = - 0.5d0
     jcvar(2) = 2 * i - 1
     jcval(2) = - 1.0d0
  else if ( ind .le. p * ( p - 1 ) / 2 + 2 * p ) then
     i = ind - p * ( p - 1 ) / 2 - p
     jcnnz = 2
     if ( jcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     jcvar(1) = n - 1
     jcval(1) = - 0.5d0
     jcvar(2) = 2 * i - 1
     jcval(2) =   1.0d0
  else if ( ind .le. p * ( p - 1 ) / 2 + 3 * p ) then
     i = ind - p * ( p - 1 ) / 2 - 2 * p
     jcnnz = 2
     if ( jcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     jcvar(1) = n
     jcval(1) = - 0.5d0
     jcvar(2) = 2 * i
     jcval(2) = - 1.0d0
  else
     i = ind - p * ( p - 1 ) / 2 - 3 * p
     jcnnz = 2
     if ( jcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     jcvar(1) = n
     jcval(1) = - 0.5d0
     jcvar(2) = 2 * i
     jcval(2) =   1.0d0
  end if
end subroutine myevaljac
```

Listing 10.7. *Subroutine evalhc.*

```
subroutine myevalhc(n,x,ind,hcrow,hccol,hcval,hcnnz,lim,lmem,flag)
  implicit none
  ! SCALAR ARGUMENTS
  logical, intent(out) :: lmem
  integer, intent(in) :: ind,lim,n
  integer, intent(out) :: flag,hcnnz
  ! ARRAY ARGUMENTS
  integer, intent(out) :: hccol(lim),hcrow(lim)
  real(kind=8), intent(in) :: x(n)
  real(kind=8), intent(out) :: hcval(lim)
  ! COMMON SCALARS
  integer :: p
  ! LOCAL SCALARS
  integer :: i,j
  ! COMMON BLOCKS
  common /pdata/ p
  ! Compute lower-triangle of the ind-th constraint's Hessian
  flag = 0
  lmem = .false.
  if ( 1 .le. ind .and. ind .le. p * ( p - 1 ) / 2 ) then
     call pair(p,ind,i,j)
     hcnnz = 6
     if ( hcnnz .gt. lim ) then
        lmem = .true.
        return
     end if
     hcrow(1) = 2 * i - 1
     hccol(1) = 2 * i - 1
     hcval(1) = - 2.0d0
     hcrow(2) = 2 * i
     hccol(2) = 2 * i
     hcval(2) = - 2.0d0
     hcrow(3) = 2 * j - 1
     hccol(3) = 2 * j - 1
     hcval(3) = - 2.0d0
     hcrow(4) = 2 * j
     hccol(4) = 2 * j
     hcval(4) = - 2.0d0
     hcrow(5) = 2 * j - 1
     hccol(5) = 2 * i - 1
     hcval(5) =   2.0d0
     hcrow(6) = 2 * j
     hccol(6) = 2 * i
     hcval(6) =   2.0d0
  else
     hcnnz = 0
  end if
end subroutine myevalhc
```

maximum number of iterations reached, P means lack of progress, and U means that the subproblem seems to be unbounded from below. The last two columns correspond to the number of times the acceleration process was launched and the accumulated number of Newtonian iterations that were used in those trials, respectively. Their exact meaning will be clarified later.

The output shows that, after five outer iterations, the stopping criterion (10.6)–(10.8) is satisfied and Algencan stops. In fact, the sentence "Flag of ALGENCAN: Solution was found." indicates that the stopping criterion (10.6)–(10.8) was satisfied. The total CPU time in seconds is displayed and the total number of calls to each subroutine coded by the user is shown.

10.4 ▪ Installing and running Algencan

This section contains simple instructions for installing Algencan on your computer and running the code for the first time. The instructions below may be outdated when you read this book. Updated instructions may be found in the README file that comes with the Algencan distribution that can be downloaded from the TANGO Project Web

```
 ==============================================================================
 This is ALGENCAN 3.0.0.
 ALGENCAN, an Augmented Lagrangian method for nonlinear programming, is part of
 the TANGO Project: Trustable Algorithms for Nonlinear General Optimization.
 See http://www.siam.org/books/fa10 for details.
 ==============================================================================

 There are no strings to be processed in the array of parameters.

 The specification file is not being used.

 Available HSL subroutines = MA57 MA86 MA97

 ALGENCAN PARAMETERS:

 firstde                =                    T
 seconde                =                    T
 truehpr                =                    T
 hptype in TN           =               TRUEHP
 lsslvr in TR           =            MA57/MC64
 lsslvr in NW           =            MA57/MC64
 lsslvr in ACCPROC      =            MA57/MC64
 innslvr                =                   TR
 accproc                =                    T
 rmfixv                 =                    T
 slacks                 =                    F
 scale                  =                    T
 epsfeas                =           1.0000D-08
 epsopt                 =           1.0000D-08
 efstain                =           1.0000D+20
 eostain                =          -1.0000D+20
 efacc                  =          -1.0000D+20
 eoacc                  =          -1.0000D+20
 iprint                 =                   10
 ncomp                  =                    6

 Specification filename =                   ''
 Output filename        =                   ''
 Solution filename      =                   ''

 Number of variables               :       8
 Number of equality constraints    :       0
 Number of inequality constraints  :      15
 Number of bound constraints       :       2

 There are no fixed variables to be removed.

 Objective function scale factor   : 1.0D-01
 Smallest constraints scale factor : 7.5D-02

 Entry to ALGENCAN.
 Number of variables  :       8
 Number of constraints:      15

out penalt  objective infeas  scaled    scaled infeas norm   |Grad| inner Newton
ite         function  ibilty  obj-funct infeas +compl graLag infeas totit forKKT
  0         1.000D+02 9.D+00  1.000D+01 1.D+00 1.D+00 1.D+00 1.D+00    0   0    0
  1 8.D+01  8.356D+01 4.D-02  8.356D+00 4.D-02 4.D-02 3.D+00 4.D-02   10C  0    0
  2 8.D+01  5.923D+01 1.D-02  5.923D+00 1.D-02 1.D-02 3.D-07 9.D-03   28C  0    0
  3 8.D+01  5.939D+01 1.D-04  5.939D+00 3.D-05 3.D-05 1.D-10 2.D-05   33C  0    0
  4 8.D+01  5.939D+01 1.D-07  5.939D+00 1.D-07 1.D-07 1.D-11 9.D-08   35C  0    0
  5 8.D+01  5.939D+01 2.D-09  5.939D+00 4.D-10 5.D-10 2.D-14 3.D-10   37C  0    0

 Flag of ALGENCAN: Solution was found.

 User-provided subroutines calls counters:

 Subroutine fsub     (coded=T):      117
 Subroutine gsub     (coded=T):       63
 Subroutine hsub     (coded=T):       37
 Subroutine csub     (coded=T):     1916 (     127 calls per constraint in avg)
 Subroutine jacsub   (coded=T):      332 (      22 calls per constraint in avg)
 Subroutine hcsub    (coded=T):      184 (      12 calls per constraint in avg)
 Subroutine fcsub    (coded=F):        0
 Subroutine gjacsub  (coded=F):        0
 Subroutine gjacpsub (coded=F):        0
 Subroutine hlsub    (coded=F):        0
 Subroutine hlpsub   (coded=F):        0


 Total CPU time in seconds:     0.00
```

Figure 10.1. *Algencan's output when solving an instance of problem (10.27) with* $p = 3$.

site. Moreover, several Web addresses pointing to third-party software and compilers are provided; these URLs should still be valid at the time you read this book.

In order to run Algencan, calling it from a Fortran code, follow the instructions below. It is assumed that you use a standard Linux environment, that basic commands such as `tar` and `make` are installed, and that `gfortran` (the Fortran compiler from GNU) is also installed. Instructions are divided into two main steps: (i) building the Algencan library and (ii) compiling your own code that calls Algencan. The Algencan library needs to be built only once.

The instructions below are a simple and arbitrary suggestion, since Algencan can also be used in Windows and Mac OS platforms as well and with a variety of Fortran compilers. Algencan can also be used from a C/C++ calling code or in connection with the modeling languages AMPL [119] and CUTEst [131].

10.4.1 ▪ Installing Algencan: Building the library

Step 1: Download the Algencan "tarball" file from the TANGO Project Web site

`http://www.siam.org/books/fa10`

and save it into the folder where you would like to install Algencan. For example, assume that the name of this folder is `/home/myusername/`.

Step 2: Go to the folder `/home/myusername/` and uncompress the tarball file by typing

```
tar -zxvf algencan-3.0.0.tgz
```

We assume the name of the downloaded file is `algencan-3.0.0.tgz`, which is the name of the file associated with the current version of Algencan. Note that as a consequence of this step, a folder called `algencan-3.0.0` has been created within `/home/myusername/`.

Step 3: Optionally (highly recommended), place the Fortran 95 version of subroutine MA57 from HSL into the folder

`/home/myusername/algencan-3.0.0/sources/hsl/`

that was automatically created in Step 2. Subroutine MA57 must be contained in a file named `hsl_ma57d.f90`. Additionally, files named `hsl_zd11d.f90`, `ma57ad.f`, `mc21ad.f`, `mc34ad.f`, `mc47ad.f`, `mc59ad.f`, `mc64ad.f`, `mc71ad.f`, and `fakemetis.f` from HSL (which correspond to the MA57 dependencies from HSL) and files named `dgemm.f`, `dgemv.f`, `dtpmv.f`, `dtpsv.f`, `idamax.f`, `lsame.f`, and `xerbla.f` from BLAS (which correspond to the MA57 dependencies from BLAS) must be placed within the same folder too. How to obtain those files and other options, related to the usage of subroutines MA86 and MA97 from HSL and the usage of the BLAS and LAPACK libraries, is detailed below.

Step 4: Go to folder `/home/myusername/algencan-3.0.0/` and type

```
make
```

If everything went well, the Algencan's library file `libalgencan.a` has been created within the folder `/home/myusername/algencan-3.0.0/lib/`. If the optional Step 3 has been followed, an HSL-related library file `libhsl.a` has been created (within the same folder) too.

10.4.2 ▪ Compiling your own code that calls Algencan

Step 5: Set an environment variable with the complete path to the Algencan folder. For example, if Algencan is installed at `/home/myusername/`, type

```
export ALGENCAN=/home/myusername/algencan-3.0.0
```

Step 6: Go to the folder where your main program and problem subroutines are. Assume that the file `myprob.f90` contains the source code of the main program and subroutines that are needed to run Algencan. Then, you can obtain the executable file `myprob` by typing

```
gfortran -O3 myprob.f90 -L$ALGENCAN/lib -lalgencan -lhsl -o myprob
```

if you followed Step 3 or by typing

```
gfortran -O3 myprob.f90 -L$ALGENCAN/lib -lalgencan -o myprob
```

if you skipped Step 3.

Step 7: Type

```
./myprob
```

to run and see the output in the screen.

10.4.3 ▪ Installation and compilation remarks

Algencan is an open source software. Therefore, the tarball file downloaded in Step 1 of the instructions above includes all the source files of Algencan, as well as the source files of *all* the examples described in the present and forthcoming chapters. Compiled versions for different platforms are *not* distributed.

The statement in Step 2 uncompresses the tarball file and creates the folders structure of Algencan. This means that in the place where the tarball file is uncompressed, the Algencan main folder, named `algencan-3.0.0` for the current version, is created. Within this main folder there are four files and three subfolders. The files are (a) the license file `license.txt`, (b) the `README` file with a few instructions to compile Algencan as well as the different Algencan's interfaces with AMPL, C/C++, and CUTEst, (c) a file named `WHATSNEW` that describes the main features of each release of Algencan, and (d) the main `Makefile` file that is used in the compilation process. As the README file explains, a few variables with paths to third-party codes may need to be edited within the file `Makefile` in order to compile the Algencan interfaces. The three subfolders are (a) `sources`, where the Algencan source files are, as well as the interfaces' source files, and several examples of usage of Algencan, (b) `bin`, and (c) `lib`. These two last subfolders are empty and receive the interfaces' executable files and the Algencan lib file, respectively, after the respective compilation processes.

Remarks on the possibilities related to the usage of the HSL Mathematical Software Library (in Step 3 of the compilation process) will be given below.

In Step 4, the Algencan library file is created and saved within folder `lib`. In Step 5, an environment variable with the full path to Algencan should be set. This variable is used in Step 6 to indicate to the compiler where the Algencan library is located (using the flag `-L`). Several examples in Fortran 77 and 90 are provided within folders `sources/examples/f77/` and `sources/examples/f90/`, respectively. Steps 6–8 may be repeated several times to test the different provided examples and/or to solve your own problems.

10.4.4 ▪ Usage of HSL subroutines in Algencan

Step 3, which is optional but highly recommended, is the step where the HSL linear algebra subroutines are made available to Algencan. Algencan is ready to solve linear systems using subroutines MA57, MA86, and/or MA97 from HSL. The more the better, since different subroutines may be used in different places of Algencan, depending on the dimension of the linear system that may need to be solved at each time. Moreover, Algencan allows the user to determine which linear system solver should be used each time.

Use of the mentioned HSL subroutines is not mandatory but is highly recommended and has a strong influence on Algencan's performance. This does not mean that the HSL linear algebra subroutines are much more powerful than other linear algebra subroutines included in Algencan. It means that Algencan does not include any linear algebra subroutine and that if the HSL subroutines are not available, only a few basic subalgorithms will be available within Algencan. These subalgorithms, which do not depend on linear algebra subroutines and are matrix-free, may be adequate only for huge-scale problems.

When running Algencan, the output's preamble shows in a dedicated line which linear algebra subroutines are available for solving linear systems. Checking the content of this line is crucial for verifying whether the HSL subroutines are being used by Algencan. If the user places the HSL subroutines in the wrong place, with the wrong filenames, or with missing dependencies, Algencan might not consider them.

Each HSL [261] subroutine (among MA57, MA86, and/or MA97) has its own dependencies from HSL, BLAS [66], and LAPACK [6]. If the BLAS or LAPACK library already exists on your computer, the respective dependencies may be omitted, while the corresponding flags (`-lblas` and/or `-llapack`) must be added to the compilation/linking command in Step 6. Depending on the installation of the libraries, the complete path to them may need to be indicated with the flag `-L` (as is done with the path to the Algencan library). HSL subroutines may be downloaded from `http://www.hsl.rl.ac.uk/`, and BLAS and LAPACK subroutines may be downloaded from `http://www.netlib.org/blas/`, and `http://www.netlib.org/lapack/`, respectively. When this book was written, HSL subroutines were available at no cost for academic research and teaching, while BLAS and LAPACK packages could be freely downloaded. In any case, it is the users' responsibility to check the third-party software licencing terms and conditions.

Each dependency must be placed in a single file. The relation between the subroutine and the filename is immediate and, for completeness, the dependencies of each linear system solver HSL subroutine that may be used by Algencan follow.

Subroutine MA57 should be saved with a file named `hsl_ma57d.f90`. Its dependencies from HSL are `hsl_zd11d.f90`, `ma57ad.f`, `mc21ad.f`, `mc34ad.f`, `mc47ad.f`, `mc59ad.f`, `mc64ad.f`, and `mc71ad.f`. Its dependencies from BLAS are `dgemm.f`, `dgemv.f`, `dtpmv.f`, `dtpsv.f`, `idamax.f`, `lsame.f`, and `xerbla.f`. Subroutine MA57 may be used in connection with Metis [158]. If Metis is not used, file `fakemetis.f` (with the empty subroutine metis_nodend) must be included.

Subroutine MA86 should be saved with a file named `hsl_ma86d.f90`. Its dependencies from HSL are `hsl_mc34d.f90`, `hsl_mc68i.f90`, `hsl_mc69d.f90`, `hsl_mc78i.f90`, `hsl_zb01i.f90`, `mc21ad.f`, `mc64ad.f`, and `mc77ad.f`. Its dependencies from BLAS are `daxpy.f`, `dcopy.f`, `dgemm.f`, `dgemv.f`, `dswap.f`, `dtrsm.f`, `dtrsv.f`, `lsame.f`, and `xerbla.f`. Subroutine MA86 has no dependencies from LAPACK. Subroutine MA86 may be used in connection with Metis. If Metis is not used, file `fakemetis.f` (with the empty subroutine metis_nodend) must be included.

Subroutine MA97 should be saved with a file named `hsl_ma97d.f90`. Its dependencies from HSL are `hsl_mc34d.f90`, `hsl_mc64d.f90`, `hsl_mc68i.f90`, `hsl_mc69d.f90`, `hsl_mc78i.f90`, `hsl_mc80d.f90`, `hsl_zb01i.f90`, `hsl_zd11d.f90`, `mc21ad.f`, `mc30ad.f`, `mc64ad.f`, and `mc77ad.f`. Its dependencies from BLAS are `daxpy.f`, `ddot.f`, `dgemm.f`, `dgemv.f`, `dnrm2.f`, `dscal.f`, `dswap.f`, `dsyrk.f`, `dtrmm.f`, `dtrmv.f`, `dtrsm.f`, `dtrsv.f`, `lsame.f`, and `xerbla.f`. Its dependencies from LAPACK are `disnan.f`, `dlaisnan.f`, `dpotf2.f`, `dpotrf.f`, `ieeeck.f`, `ilaenv.f`, and `iparmq.f`. As well as subroutine MA57 and MA86, Subroutine MA97 may be used in connection with Metis. If Metis is not used, file `fakemetis.f` (with the empty subroutine metis_nodend) must be included.

Note that subroutines MA86 and MA97 use OpenMP to be able to run in parallel. Therefore, in order to take advantage of the parallelism provided by them, OpenMP must be installed on your computer. Note that, today, OpenMP and GFortran are both provided by GCC (the GNU Compiler Collection). This means that if you have GCC installed on your computer, then you simply need to add the flag `-fopenmp` within the file `Makefile` in the Algencan main folder `/home/myusername/algencan-3.0.0/` in order to use OpenMP. This flag should be added if at least one subroutine between MA86 and MA97 is present (since MA57 does not use parallelism). Refer to the OpenMP or HSL documentation to see how to set the desired number of threads. Documentation related to the GCC implementation of OpenMP can be found here:
`http://gcc.gnu.org/onlinedocs/libgomp/`

10.5 ▪ Common questions

1. I do not want to use the automatic scaling because I feel that I have a better one for my problem. How should I proceed? Do I need to code subroutines with scale as well?

   It may be the case that the user considers his or her problem well scaled, or that the user has the ability to redefine the original problem in such a way that it is well scaled. If this is the case, the automatic scaling provided by default by Algencan should be inhibited in order to have $w_f \equiv 1$ and $w_{c_j} \equiv 1$ for all $j \in E \cup I$ in (10.4). This makes the scaled problem (10.4), which is the one solved by Algencan, to coincide with the original problem (10.1). The use of the default scaling factors (10.5) computed by Algencan can be inhibited with the keyword OBJECTIVE-AND-CONSTRAINTS-SCALING-AVOIDED. For details, refer to Section 12.5.

2. Does Algencan use proprietary subroutines? If so, which are the legal procedures that I must follow?

   Algencan may optionally use some linear algebra subroutines from the HSL Mathematical Software Library, namely, MA57, MA86, or MA97, for solving linear systems. Of course, dependencies (from HSL, BLAS, and LAPACK) of those subroutines would also be required. It is the user's responsibility to check the license terms of third-party software. (When this book was written, HSL subroutines were available free of charge and on demand for academic research and teaching.)

3. Can I call Algencan from a program written in a language other than Fortran?

   Yes. Algencan is coded in Fortran but has interfaces to be called from C/C++ programs or to be used in connection with the modeling languages AMPL and CUTEst. The README file that comes with the Algencan distribution includes examples and instruction on how to interface Algencan with the mentioned languages.

4. Which are the recommended Fortran compilers and compilation options?

   Algencan source code is divided into several source files and folders. The Algencan distribution includes a Makefile that should work well in the current common platforms running Unix/Linux, Mac OS, or Windows with MinGW or Cygwin. The current version of Algencan (3.0.0) works well with the current version of GFortran (which is part of GCC version 4.6.3) (see http://gcc.gnu.org/fortran/). Other Fortran compilers were not tested but are expected to work well.

10.6 ▪ Review and summary

We described a Fortran subroutine for minimizing a smooth function with smooth equality and inequality constraints and bounds on the variables. The subroutine is an implementation of Algorithm 4.1 with an approximate KKT condition for the solution of subproblems. The subproblems are solved using the active set strategies with projected gradients described in Chapter 9. A first, very simple example was presented, a few algorithmic parameters were discussed, and the output was described. Presentation of the most adequate way to code a problem and a discussion of the available algorithmic choices were delayed to forthcoming chapters.

10.7 ▪ Further reading

The Fortran 90 implementation of the simple example presented in Section 10.3 can be found in the file `chap10-ex1.f90` within folder `sources/examples/f90/` of the Algencan distribution. In the same folder, file `chap10-simplest-example.f90` presents an additional simpler example, as well as file `toyprob.f90`. The behavior of Algencan in a trivial infeasible problem (described in Section 10.2.3) can be observed in the problem given in file `infeas.f90`, located in the same folder.

10.8 ▪ Problems

10.1 Listing 10.8 shows a simple version of subroutine `pair`, used in the example of Section 10.3. This solution has time complexity $O(p)$. The same goal can be easily achieved with time complexity $O(1)$ by saving the pairs in a $2 \times p(p-1)/2$ matrix in an initialization phase (like within the main program `algencanma` of Listing 10.1) and then accessing this matrix in constant time. However, there exists an $O(1)$ version of subroutine `pair` that requires no initialization and no auxiliary arrays. Code it.

Listing 10.8. *Subroutine pair.*

```fortran
subroutine pair(p,ind,i,j)
  implicit none
  ! SCALAR ARGUMENTS
  integer, intent(in) :: p,ind
  integer, intent(out) :: i,j
  ! LOCAL SCALARS
  integer :: k
  i = 1
  k = ind
  do while ( k .gt. p - i )
     k = k - ( p - i )
     i = i + 1
  end do
  j = i + k
end subroutine pair
```

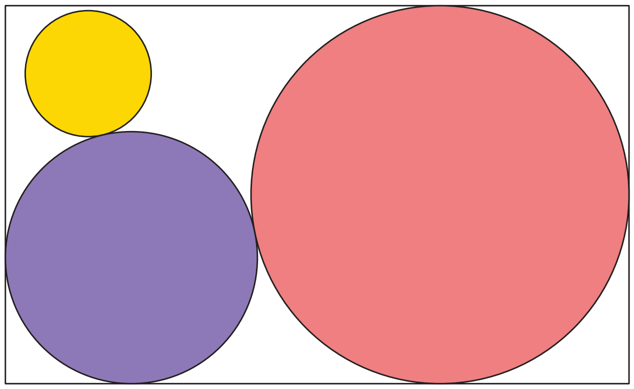

Figure 10.2. *Graphical representation of the solution obtained by Algencan.*

10.2 With the single executable statement `flag = - 1`, code subroutines `myevalfc`, `myevalgjac`, `myevalgjacp`, `myevalhl`, and `myevalhlp`. Save all of them in a file named myfirstexample.f, together with the main program `algencanma` and subroutines `myevalf`, `myevalg`, `myevalh`, `myevalc`, `myevaljac`, and `myevalhc`, provided in Listings 10.1–10.7, plus subroutine `pair` from Problem 10.1. Solve problem (10.27) for different values of p. Analyze the output identifying the main elements of the model Algorithm 4.1.

10.3 Add a mistake to any of the subroutines that computes derivatives, set `checkder` equal to true in the main program `algencanma`, and analyze the way Algencan compares coded derivatives with finite differences approximations.

10.4 (a) Code subroutines `fcsub`, `gjacsub`, and `hlsub` for problem (10.27). (b) Code subroutines `fcsub`, `gjacpsub`, and `hlpsub` for problem (10.27). (c) Solve problem (10.27) using Algencan with the provided subroutines and with the subroutines from parts (a) and (b). Compare the results. In particular, check the number of calls to each user-provided subroutine.

10.5 (a) Solve problem (10.27), discarding the subroutines that compute second derivatives. (b) Do the same, discarding subroutines that compute first derivatives. (c) Compare the results.

10.6 In order to analyze the obtained solutions, code your own subroutine to generate a graphical representation of the solution, as depicted in Figure 10.2.

10.7 Consider the infeasible problem of minimizing $x_1 + x_2$ subject to $x_1 + x_2 \leq -2$ and $x_1 + x_2 \geq 2$. To check the behavior of Algencan when dealing with an infeasible problem, solve it twice, inhibiting and enabling the stopping criterion described in Section 10.2.3. Compare the results.

Chapter 13
Practical Examples

In this chapter, we illustrate the use of Algencan in four practical applications. In all cases, Algencan 3.0.0 with the (arbitrarily chosen) HSL MA57 subroutine was considered. Algencan was compiled with GNU Fortran (GFortran) included in GCC version 4.6.3. All the experiments were executed on a 2.4-GHz Intel Core 2 Quad Q6600 with 4.0 GB of RAM memory running the GNU/Linux operating system.

13.1 ▪ Packing molecules

Molecular dynamics is a powerful technique for comprehension at the molecular level of a great variety of chemical processes. With the enhancement of computational resources, very complex systems can be studied. The simulations need starting points that must have adequate energy requirements. However, if the starting configuration has close atoms, the temperature scaling is disrupted by excessive potentials that accelerate molecules over the accepted velocities for almost any reasonable integration time step. In fact, the starting coordinates must be reliable in that they must not exhibit overlapping or close atoms, so that temperature scaling can be performed with reasonable time steps for a relatively fast energy equilibration of the system.

In [186, 192], the problem of finding the initial positions of the molecules is represented as a "packing problem." The goal is to place known objects in a finite domain in such a way that the distance between any pair of points of objects is larger than a threshold tolerance. In our case, objects are molecules and points are atoms. Following this idea, an optimization problem is defined. The mathematical (optimization) problem consists of the minimization of a function of (generally) many variables, subject to constraints that define the region in which the molecules should be placed.

Let us call $nmol$ the total number of molecules that we want to place in a region $\mathscr{R}$ of the three-dimensional space. For each $i = 1, \dots, nmol$, let $natom(i)$ be the number of atoms of the ith molecule. Each molecule is represented by the orthogonal coordinates of its atoms. To facilitate the visualization, assume that the origin is the barycenter of all the molecules. For all $i = 1, \dots, nmol$, $j = 1, \dots, natom(i)$, let

$$A(i,j) = (a_1^{ij}, a_2^{ij}, a_3^{ij})$$

be the coordinates of the jth atom in the ith molecule.

Suppose that one rotates the ith molecule sequentially around the axes x_1, x_2, and x_3, where $\gamma^i = (\gamma_1^i, \gamma_2^i, \gamma_3^i)$ are the angles that define such rotations. Moreover, suppose that after these rotations, the whole molecule is displaced so that its barycenter, instead of the origin, becomes $t^i = (t_1^i, t_2^i, t_3^i)$. These movements transform the atom of coordinates $A(i,j)$ in a displaced atom of coordinates

$$P(i,j) = (p_1^{ij}, p_2^{ij}, p_3^{ij}).$$

Observe that $P(i,j)$, $j = 1,\ldots,natom(i)$, is a function of (t^i, γ^i), the relation being

$$P(i,j) = t^i + R(\gamma^i)A(i,j),\ j = 1,\ldots,natom(i),$$

where

$$R(\gamma^i) = \begin{pmatrix} c_1^i c_2^i c_3^i - s_1^i s_3^i & s_1^i c_2^i c_3^i + c_1^i s_3^i & -s_2^i c_3^i \\ -c_1^i c_2^i s_3^i - s_1^i c_3^i & -s_1^i c_2^i s_3^i + c_1^i c_3^i & -s_2^i s_3^i \\ c_1^i s_2^i & s_1^i s_2^i & c_2^i \end{pmatrix}, \tag{13.1}$$

in which $s_k^i \equiv \sin \gamma_k^i$ and $c_k^i \equiv \cos \gamma_k^i$ for $k = 1,2,3$.

In [186, 192], the objective is to find angles γ_i and displacements t_i, $i = 1,\ldots,nmol$, in such a way that whenever $i \neq i'$,

$$\|P(i,j) - P(i',j')\|_2^2 \geq d^2 \tag{13.2}$$

for all $j = 1,\ldots,natom(i)$, $j' = 1,\ldots,natom(i')$, where $d > 0$ is the required minimum distance, and

$$P(i,j) \in \mathscr{R} \tag{13.3}$$

for all $i = 1,\ldots,nmol$, $j = 1,\ldots,natom(i)$. In other words, the rotated and displaced molecules must remain in the specified region and the distance between any pair of atoms must not be less than d. Note that region $\mathscr{R}$ does not need to be convex and that it could be replaced by a region $\mathscr{R}^{ij}$ for each atom of each molecule (as required in most real cases).

The objective (13.2) leads us to define the following merit function f:

$$\begin{aligned} & f(t_1,\ldots,t_{nmol},\gamma_1,\ldots,\gamma_{nmol}) \\ &= \sum_{i=1}^{nmol} \sum_{j=1}^{natom(i)} \sum_{i'=i+1}^{nmol} \sum_{j'=1}^{natom(i')} \max\left\{0, d^2 - \|P(i,j) - P(i',j')\|_2^2\right\}^2. \end{aligned} \tag{13.4}$$

Note that $f(t_1,\ldots,t_{nmol},\gamma_1,\ldots,\gamma_{nmol})$ is nonnegative for all angles and displacements. Moreover, f vanishes if and only if the objective (13.2) is fulfilled. This means that if we find displacements and angles where $f = 0$, the atoms of the resulting molecules are sufficiently separated. This leads us to define the following minimization problem:

$$\text{Minimize } f(t_1,\ldots,t_{nmol},\gamma_1,\ldots,\gamma_{nmol}) \tag{13.5}$$

subject to (13.3) for all $i = 1,\ldots,nmol$, $j = 1,\ldots,natom(i)$.

The objective function f is continuous and differentiable, although their second derivatives are discontinuous. The number of variables is $6 \times nmol$ (three angles and a displacement per molecule). The analytical expression of f is cumbersome, since it involves consecutive rotations and its first derivatives are not very easy to code. However, optimization experience leads us to pay the cost of writing a code for computing derivatives with the expectation that algorithms that take advantage of first-order information are profitable, especially when the number of variables is large. Having a code that computes f and its gradient, we are prepared to solve (13.5) using constrained optimization techniques.

13.1.1 ▪ Analyzing a simplified version

In order to focus in the application of Algencan to the described molecules packing problem, we consider a simplified (less realistic) instance in which molecules are all identical and they are composed of a single atom. Note that the single-atom feature eliminates the rotation angles as variables and the variables of the problem become the translations only. If, in addition, we set $nmol = N$, $d = 2r$, and $\mathscr{R} = \{x \in \mathbb{R}^3 \mid \|x\|_2 \le R\}$, the problem can be seen as the problem of placing N identical spheres with radius r within a sphere with radius R centered at the origin. This is one of the many variants of the well-known packing problems described in, for example, [65].

Summing up, the problem considered in this section is given by

$$\text{Minimize } f(x) \text{ subject to } c(x) \le 0, \tag{13.6}$$

where $x = (t_1, \dots, t_N) \in \mathbb{R}^{3N}$,

$$f(t_1, \dots, t_N) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} \max\{0, (2r)^2 - \|t_i - t_j\|_2^2\}^2, \tag{13.7}$$

and

$$c_i(x) = \|t_i\|_2^2 - (R - r)^2, \; i = 1, \dots, N. \tag{13.8}$$

A key point that strongly affects the difficulty of an instance of a packing problem is its density (occupied fraction), which in the case of problem (13.6)–(13.8) is given by $N(r/R)^3$. Considering that each atom is a sphere with a radius of 1Å, the volume occupied by the atoms in liquid water is roughly 30% of the total volume. This density was used in the illustrative examples of this section.

The treatment of the packing problem that we present below is valid as an illustration of the techniques applied to solve the much more complex molecules packing problem described in the previous subsection. Numerical examples intend to resemble the techniques implemented in the software Packmol [186, 192]. If the focus were classical packing problems by themselves, many other techniques, such as those based on lattices [85], might be considered.

The problem has $n = 3N$ variables and $m = N$ inequality constraints. Its has no equality constraints and no bounds on the variables. We coded subroutines `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub`. The last three subroutines compute the constraints and their first- and second-order derivatives, respectively, and they are very easy to code. Subroutine `fsub` computes the objective function. It is also easy to code and its naive implementation requires $O(N^2)$ operations to evaluate f at a given point. (This feature will be addressed below.) Subroutine `gsub` is also easy to code, while coding subroutine `hsub` is rather boring and prone to error. They both share with `fsub` the $O(N^2)$ time complexity to evaluate the gradient and the Hessian of the objective function, respectively, at a given point. The six subroutines are part of the Algencan distribution (file `chap13-packmol-dense.f90` within folder `sources/examples/f90/`). As a starting guess, we consider $x^0 = (t_1^0, \dots, t_N^0)$ with uniformly distributed random $t_i^0 \in [-R, R]^3$.

Before considering a particular instance, we need to analyze (a) the number of nonnull elements in the Jacobian and (b) the number of memory positions required to compute the lower triangle of the Hessian of the Augmented Lagrangian function of problem (13.6)–(13.8). These are the values that must be given to Algencan's parameters `jcnnzmax` and `hnnzmax`, respectively, when the user opts for choice S1 (see Chapter 11) and codes subroutines `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub`. The number of positions in

(b) is the sum of (b1) the number of nonnull elements in the lower triangle of the Hessian of the objective function, (b2) the sum of the number of nonnull elements in the lower triangle of the Hessian matrix of each constraint, and (b3) the number of nonnull elements in the lower triangle of the matrix given by the transpose of the Jacobian times the Jacobian. An upper bound on this last number is given by $\frac{1}{2}\sum_{j=1}^{m} \texttt{jcnnzmax}_j(\texttt{jcnnzmax}_j + 1)$, where $\texttt{jcnnzmax}_j$ is the number of nonnull elements of the jth row of the Jacobian (i.e., the number of nonnull elements in the gradient of the jth constraint). Including the quantity described in (b3) in the value of `hnnzmax` is a conservative approach, since it is required only if the method used to solve the Augmented Lagrangian subproblems is the Euclidean trust-region method (which is not the usual choice in large-scale problems; see Sections 10.2.11 and 12.8). Anyway, including this quantity is costless in this case because the transpose of the Jacobian times the Jacobian is a very sparse matrix for the problem at hand.

Looking at the nonlinear interaction of the variables in the constraints (13.8), it is easy to see that the gradient of each constraint has no more than three nonnull elements and that the (diagonal and constant) Hessian matrix of each constraint has exactly three nonnull elements. This means that (i) the Jacobian matrix has no more than $3N$ nonnull entries and we must set the Algencan's parameter $\texttt{jcnnzmax} = 3N$, (ii) the sum of the number of nonnull elements in the lower triangle of the Hessian matrix of each constraint is $\texttt{hnnzmax}_{b2} = 3N$, and (iii) the number of nonnull elements in the lower triangle of the matrix given by the transpose of the Jacobian times the Jacobian is $\texttt{hnnzmax}_{b3} = 6N$. The objective function (13.7) has a potential nonlinear interaction between every pair of variables, meaning that its Hessian matrix is potentially dense and its lower triangle may have up to $\texttt{hnnzmax}_{b1} = 3N(3N+1)/2$ nonnull elements. Therefore, we must set the Algencan parameter `hnnzmax` as $\texttt{hnnzmax}_{b1}$ plus $\texttt{hnnzmax}_{b2}$ plus $\texttt{hnnzmax}_{b3}$, i.e., $\texttt{hnnzmax} = 3N(3N+1)/2 + 3N + 6N$. This $O(N^2)$ memory requirement, which together with the $O(N^2)$ time complexity for evaluating the objective function may prevent the application of Algencan to instances with large N, will be tackled soon.

13.1.2 ▪ Tuning problem's subroutines and Algencan parameters

As a starting example, we consider the application of Algencan to an instance of problem (13.6)–(13.8) with $r = 1$, $N = 1{,}000$, and $R = 15$ (density ≈ 0.3). We set $\texttt{epsfeas} = \texttt{epsopt} = 10^{-8}$, $\texttt{efstain} = \sqrt{\texttt{epsfeas}}$, $\texttt{eostain} = \texttt{epsopt}^{1.5}$, $\texttt{efacc} = \sqrt{\texttt{epsfeas}}$, and $\texttt{eoacc} = \sqrt{\texttt{epsopt}}$. We also set `outputfnm = ''`, `specfnm = ''`, and `nvparam = 0`. A solution was found as a result of the second acceleration process, using 3 outer iterations (65 inner iterations, 398 calls to `fsub`, 118 calls to `gsub`, 74 calls to `hsub`, 429 calls to `csub` per constraint in average, 25 calls to `jacsub` per constraint in average, and 8 calls to `hcsub` per constraint in average) and 3.53 seconds of CPU time.

Since we are seeking applications with much larger values of N, we decided to tackle the $O(N^2)$ time and memory requirements mentioned above. The idea is very simple: at a solution, each sphere "touches" no more than other 12 identical spheres (the kissing number in $\mathbb{R}^3$ [85]). This means that, near a solution, most of the spheres pairs are such that spheres are far from each other and do not contribute to the sum in (13.7). The strategy to compute (13.7) efficiently is based on a partitioning of the three-dimensional space into cubes of side $2r$ and consists of (a) assigning each t_i to a cube and (b) computing the terms in the summation in (13.7) associated with every pair (i, j) such that t_i and t_j belong to the same cube or to neighbor cubes. Remaining pairs do not contribute to the sum in (13.7) and can be ignored. See [65] for details. This idea is based on efficient algorithms developed to reduce the asymptotic computational complexity of the N-body problem [147].

We implemented this idea (which involves modifications in subroutines `fsub`, `gsub`, and `hsub`) and solved the same instance again. The modified subroutines are part of the Algencan distribution (file `chap13-packmol-sparse.f90` within folder `sources/examples/f90/`). From now on, we will call "dense" and "sparse" implementation of problem (13.6)–(13.8) the implementation that computes all terms in (13.7) and the implementation that computes only a few terms, respectively. Since the implementations perform floating point operations in different orders, slightly different results may be expected. In fact, Algencan found a solution as a result of the first acceleration process, using 2 outer iterations (61 inner iterations, 377 calls to `fsub`, 106 calls to `gsub`, 67 calls to `hsub`, 407 calls to `csub` per constraint in average, 26 calls to `jacsub` per constraint in average, and 10 calls to `hcsub` per constraint in average). The highlight is that the number of computed terms in the summation in (13.7) went from $N(N-1)/2 = 499{,}500$ to 6,775 in average (computed over all points at which the objective function was evaluated during the problem resolution). The same reduction applies to the evaluation of the gradient and the Hessian of the objective function. Assuming that computing the objective function and its derivatives is the dominant task, a similar reduction (99%) would also be expected in the elapsed CPU time. This was *not* the case. The elapsed CPU time was 1.45 seconds.

In fact, in the dense implementation of problem (13.6)–(13.8), the computation of the objective function and its derivatives represents 70% of the computational effort (2.46 seconds of CPU time, over a total of 3.53 seconds). The actual reduction, estimated in 99% considering the reduction in the number of computed terms in the summation in (13.7), was in fact 92%, due to the overheads associated with the sparse implementation of (13.7). This explains the overall reduction of 59%, going from the 3.53 seconds of CPU time of the dense implementation to the 1.45 seconds of the sparse implementation.

If we now profile the sparse implementation of problem (13.6)–(13.8), the most expensive task happens to be the factorization of matrices, consuming 74% of the computational effort. Hence, the natural question is, are the matrices of problem (13.6)–(13.8) (Hessian of the Augmented Lagrangian and Jacobian of the KKT system) such that they favor the application of iterative linear systems solvers instead of direct methods (i.e., matrices factorizations)?

Since first- and second-order derivatives were coded, the MA57 subroutine from HSL is available for solving linear systems, and the instance at hand has more than 500 variables, the active set strategy used to solve the Augmented Lagrangian subproblems employs a line-search Newton's method within the faces (see Sections 8.5.1 and 12.8). This is one of the places where linear systems are being solved. The other place is within the acceleration process (see Section 12.10) to solve the KKT system by Newton's method. The alternative that avoids the usage of a direct linear systems solver in the first case is to use a line-search truncated Newton approach, which solves Newtonian systems by conjugate gradients (see Sections 8.5.2 and 12.8). This selection of the inner-to-the-faces method can be achieved with the keyword TRUNCATED-NEWTON-LINE-SEARCH-INNER-SOLVER. Regarding the second place where linear systems are being solved, since the current implementation of Algencan does not consider the possibility of applying the acceleration process in connection with an iterative linear systems solver, the only choice is to inhibit the application of the acceleration process with the keyword SKIP-ACCELERATION-PROCESS.

We modified these two parameters and applied Algencan once again to the same instance. Algencan found a solution using 6 outer iterations (47 inner iterations, 95 calls to fsub, 67 calls to gsub, 47 calls to hsub, 95 calls to csub per constraint in average, 8 calls to jacsub per constraint in average, and 6 calls to hcsub per constraint in average) and 0.16 seconds of CPU time. In this run of Algencan, approximately 22 times faster than our first trial, the most expensive tasks were the computation of the Hessian of the Augmented Lagrangian, its products by a given vector (main task of the CG method), the linear algebra of the CG method itself, and the evaluation of the objective function and its derivatives.

After having modified the evaluation of the objective function and its derivatives, and having tuned a few parameters of Algencan, it appears that we are now ready to address larger instances of problem (13.6)–(13.8).

13.1.3 ▪ Solving large instances

In the present section, we are illustrating the application of Algencan to a simple packing problem that mimics a real problem in the field of molecular dynamics. This is why we mentioned in the previous subsection that we were interested in instances of problem (13.6)–(13.8) with a "density" of approximately 0.3. For the same reason, we close the section by showing the performance of Algencan in instances with $10^5 \leq N \leq 10^6$, which is of the same order of magnitude of the largest real applications reported in the literature (see [192]).

Before running Algencan (on large-scale instances) with the parameters chosen in the previous subsection, there is a single parameter to adjust: the number hnnzmax of memory positions required to store the lower triangle of the Hessian of the Augmented Lagrangian function. In fact, when a method different from the Euclidean trust-region method is used as part of the Augmented Lagrangian subproblems' solver, hnnzmax must be an upper bound on the number of triplets needed to store the Hessian matrix of the Lagrangian function (instead of the Hessian matrix of the *Augmented* Lagrangian function). (See Section 10.2.11.) Since this is the case of the present numerical experiments, in which we chose a truncated Newton line-search strategy, the value of hnnzmax may be modified. Moreover, the motivation to compute a new value for hnnzmax is that the value computed in the previous subsections (hnnzmax $= 3N(3N+1)/2+3N+6N$) is not suitable for large values of N. Since the transpose of the Jacobian times the Jacobian does not need to be stored any more, $6N$ memory positions can be disregarded and the new value for hnnzmax might be $3N(3N+1)/2+3N$, which is also *not suitable* for large values of N. However, in practice, a much smaller amount of memory is required. Therefore, based on the sparsity of the Hessian of the objective function near a solution discussed above, in the next experiment we heuristically set hnnzmax $= 100N$.

Table 13.1 shows the results and Figure 13.1 illustrates the solutions found. In the table, outit is the number of outer iterations, innit is the total number of inner iterations, fcnt, gcnt, and hcnt are, respectively, the number of calls to subroutines fsub, gsub, and hsub. Finally, ccnt, jcnt, and hccnt are the average number of calls to subroutines csub, jacsub, and hcsub per constraint, and Time is the CPU time in seconds.

Table 13.1. *Computational effort measures of the application of Algencan to large-scale instances of the packing problem (13.6)–(13.8).*

N	R	outit	innit	fcnt	gcnt	hcnt	ccnt	jcnt	hccnt	Time
10^5	70	6	138	532	158	138	532	11	8	92.56
5×10^5	120	6	262	1,000	282	262	1,000	20	17	1,340.11
10^6	150	6	375	1,561	395	375	1,561	24	21	4,506.39

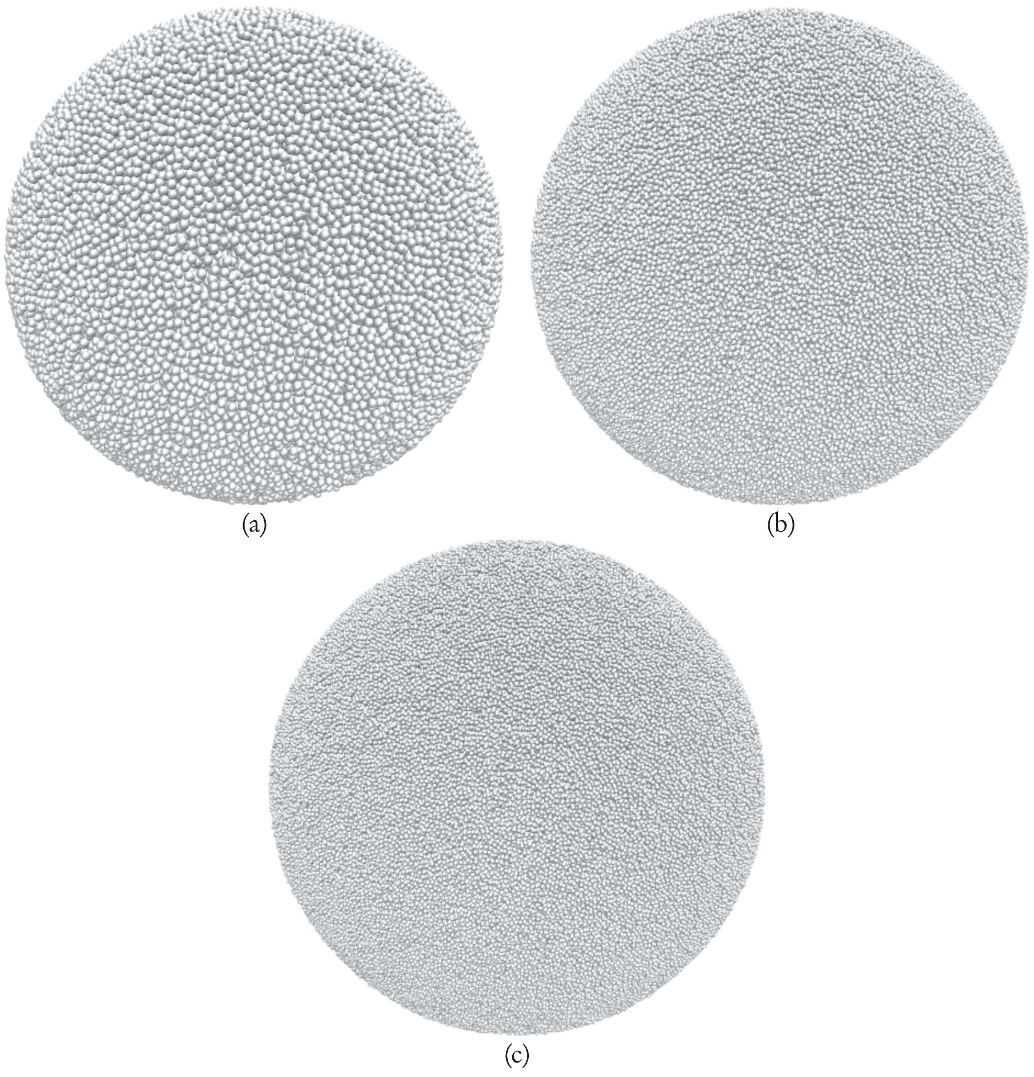

Figure 13.1. *Graphical representation of the solution found to three instances of the unitary-radius (i.e., $r = 1$) packing problem (13.6)–(13.8) with (a) $N = 100{,}000$ and $R = 70$, (b) $N = 500{,}000$ and $R = 120$, and (c) $N = 1{,}000{,}000$ and $R = 150$.*

13.2 ▪ Drawing proportional maps

A sketch of a map of the Americas is presented in Figure 13.2. The sketch was drawn by joining with segments 132 arbitrarily selected points in the borders of 17 regions of a regular map. Most of the regions are associated with countries (from south to north): Argentina, Chile, Uruguay, Brazil, Paraguay, Bolivia, Peru, Ecuador, Colombia, Venezuela, the Guianas (Guyana, Suriname, and French Guiana), Central America (Panama, Costa Rica, Nicaragua, Honduras, El Salvador, Guatemala, and Belize), Mexico, Cuba, Canada, and Alaska. It is a fact that no map can maintain proportionality between areas and distances simultaneously. Here, we wish to redraw the map of the Americas keeping the proportionality among the area of the regions. Moreover, the redrawn map should be as similar as possible to the regular map [184]. As a different project, we wish to draw maps in which the size of each region is proportional to the population of the region or to its gross domestic product (GDP).

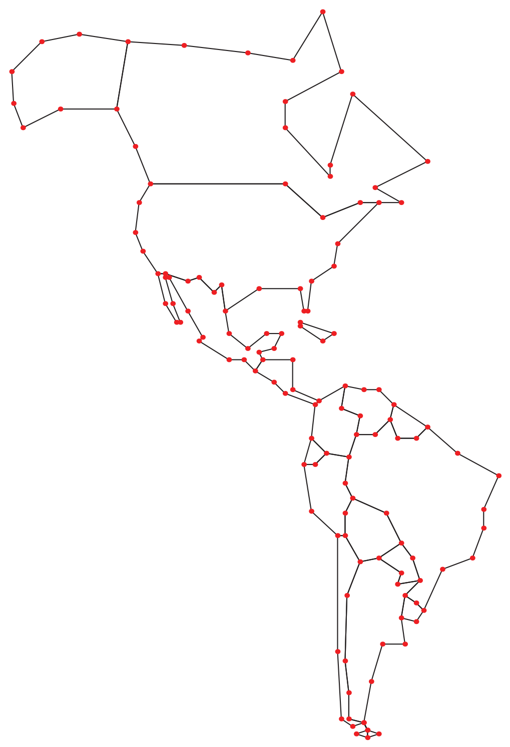

Figure 13.2. *Sketch of a regular map of the Americas. Points were arbitrarily selected in the borders of some regions and borders were approximated by joining the selected points with segments.*

Let $n_r = 17$ be the number of considered regions and let $n_p = 132$ be the number of arbitrarily selected points $\bar{p}_1, \dots, \bar{p}_{n_p}$ (illustrated in Figure 13.2). For each region j, let o_j be the number of points in its border and let $\gamma_{1j}, \dots, \gamma_{o_j j}$ be the indices of its vertices (regions are in fact polygons) numbered counterclockwise. The area $\bar{\alpha}_j$ of the polygon that represents region j can be computed (see, for example, [67]) as

$$\bar{\alpha}_j = \sum_{i=1}^{o_j} (\bar{p}^x_{\gamma_{ij}} \bar{p}^y_{\gamma_{i\oplus1,j}} - \bar{p}^x_{\gamma_{i\oplus1,j}} \bar{p}^y_{\gamma_{ij}}),$$

where $\bar{p}_i = (\bar{p}^x_i, \bar{p}^y_i)^T$ for $i = 1, \dots, n_p$, and for each region j, $\gamma_{i\oplus1,j}$ represents the index of its $(i+1)$th vertex if $i < o_j$ and $\gamma_{o_j\oplus1,j} \equiv \gamma_{1,j}$. This means that the area of each region j in the considered regular map is approximately $\bar{\alpha}_j$ and the total area of the considered regular map of the Americas is given by $\bar{\alpha} = \sum_{j=1}^{n_r} \bar{\alpha}_j$. Let $\bar{\beta}_j$ be the target area of region j for $j = 1, \dots, n_r$, and let $\bar{\beta} = \sum_{j=1}^{n_r} \bar{\beta}_j$. These target areas may represent the real territorial areas of the regions, or their population, or the GDP.

13.2.1 ▪ Problem definition: A first approach

We are ready to define our optimization problem, whose variables will be the "new" locations $p_1, \dots, p_{n_p} \in \mathbb{R}^2$ of the given points $\bar{p}_1, \dots, \bar{p}_{n_p}$. Constraints regarding the proportionality among the regions will be given by

$$\frac{1}{2}\sum_{i=1}^{o_j}(p^x_{\gamma_{ij}} p^y_{\gamma_{i\oplus 1,j}} - p^x_{\gamma_{i\oplus 1,j}} p^y_{\gamma_{ij}}) = \beta_j(\bar{\alpha}/\bar{\beta}),\ j=1,\dots,n_r, \tag{13.9}$$

where the scaling has the purpose of obtaining a new map of the same size of the considered regular map. The objective function, which represents the desire to obtain a map similar to the considered regular map, may be given by

$$\frac{1}{2}\sum_{j=1}^{n_p}\|p_j - \bar{p}_j\|^2. \tag{13.10}$$

Hence, the problem consists of minimizing (13.10) subject to (13.9). The problem has $n = 2n_p$ variables, $m = n_r$ (equality) constraints, and no bound constraints.

First- and second-order derivatives are easy to code by hand and since there is no relation (common expressions) between the constraints, we opted for coding subroutines `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub`. The six subroutines are part of the Algencan distribution (file `chap13-america-areas.f90` within folder `sources/examples/f90/`). As a starting guess, we consider the natural choice $p_i = \bar{p}_i$ for $i = 1,\dots,n_p$.

A relevant comment concerns the computation of the gradient and Hessian of each constraint in (13.9). Constraint j involves points (variables) $p_{\gamma_1,j},\dots,p_{\gamma_{o_j},j} \in \mathbb{R}^2$. Each point appears twice (interacting with the previous and the next vertex of the polygon that represents the jth region), and since the constraint is a sum, it is natural to think of each partial derivative as the sum of the partial derivatives of the two terms where each variable appears. Since subroutines `jacsub` and `hcsub` must compute sparse gradients and Hessians, respectively, this reasoning leads us to conclude that the user should implement (within those subroutines) the sum of these particular sparse arrays and matrices. Another option is to use the possibility of representing an element of a sparse array or matrix as the sum of several triplets in the sparse structure built up by the user-provided subroutine. By this, we mean that if, for example, the sparse Hessian of constraint j computed by subroutine `hcsub` contains two different entries k_1 and k_2 saying

```
hcrow(k1) = r, hccol(k1) = c, hcval(k1) = v,
```

and

```
hcrow(k2) = r, hccol(k2) = c, hcval(k2) = w,
```

Algencan understands this as $[\nabla^2 c_j]_{rc} = v + w$. This interpretation eliminates the requirement of computing the sum of sparse arrays and/or matrices within the user-provided subroutine, simplifying the user coding task. The price to be paid is that Algencan might need to save and manipulate sparse structures with a larger number of elements. This may be undesirable in critical cases, but in the present problem, in which the number of variables and constraints is small, it appears to be a very convenient choice.

It is not hard to see that the number of memory positions needed to save the Jacobian of the constraints is `jcnnzmax` $= \sum_{j=1}^{n_r} 4o_j$. The number `hnnzmax` of memory positions needed to save the Hessian of the Augmented Lagrangian is given by `hnnzmax1` $=$ $2n_p$ for the (diagonal) Hessian of the objective function plus `hnnzmax2` $= \sum_{j=1}^{n_r} 2o_j$ to save, simultaneously, the lower triangles of the Hessians of all constraints plus `hnnzmax3` $=$

$\sum_{j=1}^{n_r}(4o_j)(4o_j+1)/2$ to save the lower triangle of $c'(x)^T c'(x) = \sum_{j=1}^{n_r} \nabla c_j(x) \nabla c_j(x)^T$, i.e., hnnzmax $= 2\, n_p + \sum_{j=1}^{n_r} 2o_j + \sum_{j=1}^{n_r}(4o_j)(4o_j+1)/2$.

The amount of memory hnnzmax described in the paragraph above is the one required by Algencan to compute the Hessian matrix of the Augmented Lagrangian (see, for example, (12.3)). This matrix needs to be computed and factorized if the trust-region approach is used for solving the Augmented Lagrangian subproblems (which is the default choice for small problems with available second-order derivatives and an available linear systems solver). On the other hand, if the Newton line-search approach is used to solve the Augmented Lagrangian subproblems, the matrix to be computed and factorized is the one in (12.5). In this case, the quantity hnnzmax3 is not needed and can be dismissed in the computation of hnnzmax. The same remark applies if a truncated Newton approach is used to solve the Augmented Lagrangian subproblems (see Section 12.8 for details). In any case, since the value of hnnzmax must be an upper bound on the required number of memory positions, the one computed above may be used in combination with any algorithmic possibility for solving the Augmented Lagrangian subproblems, except in large-scale cases in which sharp upper bounds on the memory requirements may be necessary.

Setting the target values $\bar{\beta}_j$ as the real territorial areas of the 17 considered regions, we are ready to solve the problem with Algencan. (Constants that define the problem can be found in the source file chap13-america-areas.f90 within folder sources/examples/f90/, that accompanies the Algencan distribution. The constants are in fact defined in the module modamerica.f90, within the same folder.) We set epsfeas $=$ epsopt $= 10^{-8}$, efstain $= \sqrt{\texttt{epsfeas}}$, eostain $=$ epsopt$^{1.5}$, efacc $= \sqrt{\texttt{epsfeas}}$, and eoacc $= \sqrt{\texttt{epsopt}}$. We also set outputfnm = '', specfnm = '', and nvparam = 0. A solution was found as a result of the first acceleration process, using 8 outer iterations (19 inner iterations, 43 calls to fsub, 48 calls to gsub, 21 calls to hsub, 51 calls to csub per constraint in average, 48 calls to jacsub per constraint in average, and 21 calls to hcsub per constraint in average) and 0.05 seconds of CPU time. Figures 13.3(a) and 13.3(b) show, respectively, the considered regular map of the Americas and the map redrawn with sizes proportional to the real area of each region.

Note that the map in Figure 13.3(b) is indeed a map (no crossing segments). This bonus comes from the fact that the regular map is similar to the redrawn map, and using it as a starting point, together with nearly satisfied constraints, the rest of the map was easily drawn. We say it is a bonus because there is no constraint in the model requesting the solution to be associated with the picture of a map. If the target values are replaced by the population or the GDP of each region, the regular map is far from a solution and the solution of the optimization problem described above is not a map anymore. For those cases a new model is needed.

13.2.2 ▪ Dealing with population and GDP proportional maps

The inconvenience of the model presented in the previous subsection is that the minimization of the objective function (13.10) is not enough to obtain a solution associated with a map (i.e., a set of points that after being joined with segments has no crossing segments). Since adding a small number of constraints to represent exactly this requirement may be hard and adding simple sufficient constraints (like each point p_i to be in a vicinity of $\bar{p}_i$) can make the problem infeasible, we will try with a different objective function. The idea is to minimize the distance of each redrawn region to a scaled, slightly rotated and translated rendering of the region in the regular map.

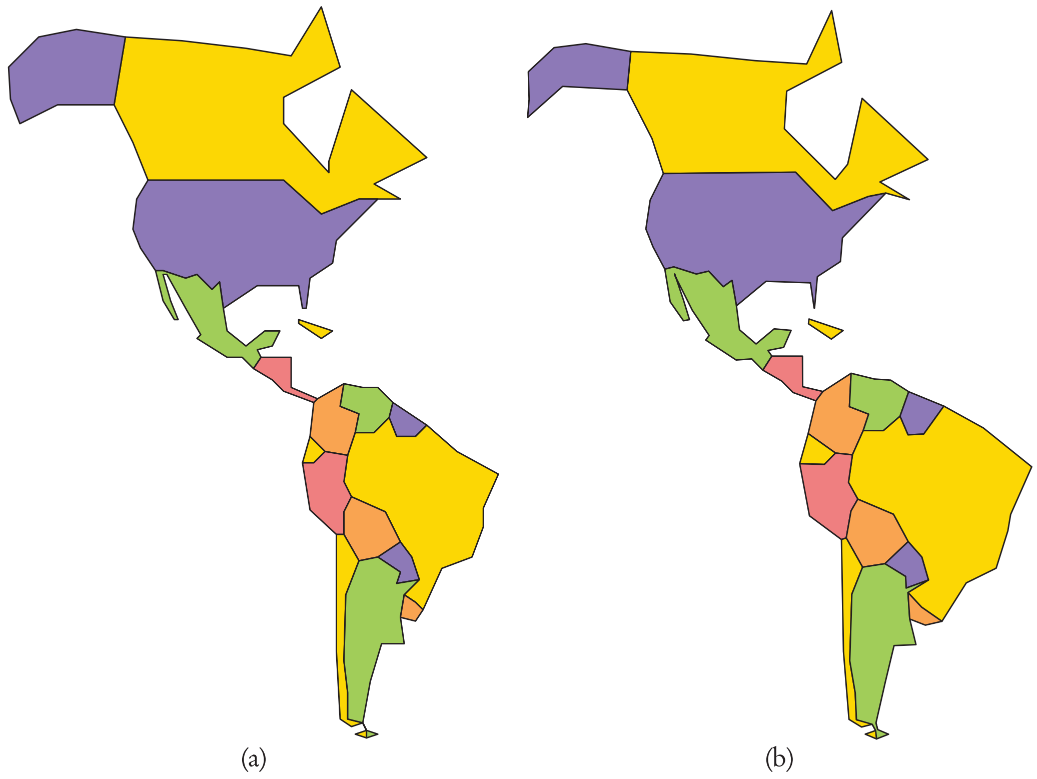

Figure 13.3. *(a) Regular map of the Americas. (b) Map redrawn with sizes proportional to the real area of each region.*

The new model can be written as

$$
\begin{array}{lll}
\text{Minimize} & \frac{1}{2}\sum_{j}^{n}\sum_{i=1}^{o_j} \|p_{\gamma_{ij}} - (t_j + \bar{c}_j + R_j D_j(\bar{p}_{\gamma_{ij}} - \bar{c}_j))\|^2 & \\
\text{subject to} & \frac{1}{2}\sum_{i=1}^{o_j}(p^x_{\gamma_{ij}} p^y_{\gamma_{i\oplus 1,j}} - p^x_{\gamma_{i\oplus 1,j}} p^y_{\gamma_{ij}}) = \beta_j(\bar{\alpha}/\bar{\beta}), & j = 1,\dots,n, \\
& -0.1 \ \le\ \theta_j \le 0.1, & j = 1,\dots,n, \\
& 0 \ \le\ d^j_k \le 2, & j = 1,\dots,n, k = 1,2, \\
& -0.1|\bar{c}^j_k| \ \le\ t^j_k \le 0.1|\bar{c}^j_k|, & j = 1,\dots,n, k = 1,2,
\end{array}
\qquad (13.11)
$$

where $\bar{c}_j = (1/o_j)\sum_{i=1}^{o_j} \bar{p}_{\gamma_{ij}}$ is the constant "center of mass" of each region j,

$$
R_j = \begin{pmatrix} \cos(\theta_j) & -\sin(\theta_j) \\ \sin(\theta_j) & \cos(\theta_j) \end{pmatrix},
$$

$D_j = \text{diag}(d^j_1, d^j_2)$, and $t_j \in \mathbb{R}^2$ are a variable (counterclockwise) rotation matrix, a variable deformation, and a variable translation, respectively, for each region j. The objective function in (13.11) says that a redrawn region resembles its usual picture if it is similar to a scaled, slightly rotated and/or translated version of its usual picture. The bound constraints in (13.11) are arbitrary and express our idea of "slightly" rotated and translated.

Problem (13.11) has $n = 2n_p + 5n_r$ variables and $m = n_r$ constraints. Second-order derivatives of the objective function are a little bit harder to code (with respect to the second derivatives of the objective function of the previous model), but it is our experience that their availability, in general, improves the behavior of Algencan. Note that problems presented up to now in this chapter were solved with the acceleration process, which can be used only if second derivatives are available. The value of `jcnnzmax` is the same as in the previous subsection (since the constraints, other than the bound constraints, are the same), i.e., `jcnnzmax` $= \sum_{j=1}^{n_r} 4o_j$. The upper bound `hnnzmax` on the number of memory positions required to compute the Hessian matrix of the Augmented Lagrangian is given by `hnnzmax` = `hnnzmax1` + `hnnzmax2` + `hnnzmax3`. Values of `hnnzmax2` and `hnnzmax3` also depend only on the constraints and remain unchanged, i.e., `hnnzmax2` $= \sum_{j=1}^{n_r} 2o_j$ and `hnnzmax3` $= \sum_{j=1}^{n_r} (4o_j)(4o_j + 1)/2$. The value of `hnnzmax1` is given by `hnnzmax1` $= \sum_{j=1}^{n_r} 24o_j$.

Setting the target values $\bar{\beta}_j$ as the GDP of each region, we are ready to solve the problem with Algencan. The corresponding file that accompanies the Algencan distribution is `chap13-america-pop-gdp.f90` (within folder `sources/examples/f90/`). Setting the Algencan parameters with the values of the previous subsection, Algencan finds a solution as a result of the acceleration process after 17 outer iterations. Figure 13.4(a) shows the solution. In fact, Algencan reaches the required precisions to launch the acceleration process after the tenth outer iteration. However, although the required feasibility and optimality tolerances are almost obtained in every acceleration process, they are strictly satisfied only after the seventh trial. As a whole, Algencan uses 3.90 seconds of CPU time, 17 outer iterations, 839 inner iterations, 1,700 calls to `fsub` subroutine, 1,151 calls to `gsub` subroutine, 911 calls to `hsub` subroutine, 1,890 calls to subroutine `csub` per constraint in average, 1,151 calls to subroutine `jacsub` per constraint in average, and 911 calls to subroutine `hcsub` per constraint in average.

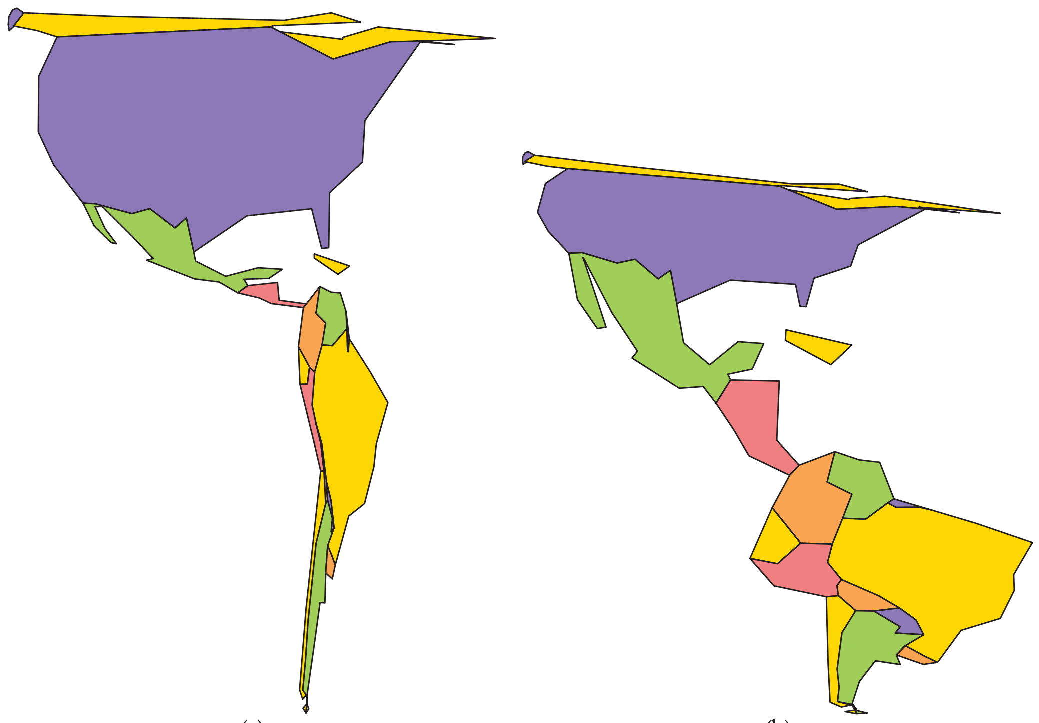

Figure 13.4. *(a) Map redrawn with sizes proportional to the GDP of each region and (b) map redrawn with sizes proportional to the population of each region.*

The behavior described in the paragraph above suggests that increasing the maximum number of iterations of the acceleration process (whose value is 10 by default) may improve the performance of Algencan. Setting this value to 100 with the help of the keyword ACCELERATION-PROCESS-ITERATIONS-LIMIT, we run Algencan again. This time, the same solution was found in the first acceleration process, using 55 iterations (of the Newton's method applied to the KKT system). As a whole, Algencan used 2.97 seconds of CPU time, 10 outer iterations, 693 inner iterations, 1,345 calls to `fsub` subroutine, 928 calls to `gsub` subroutine, 742 calls to `hsub` subroutine, 1,502 calls to subroutine `csub` per constraint in average, 928 calls to subroutine `jacsub` per constraint in average, and 742 calls to subroutine `hcsub` per constraint in average.

Only as an illustrative example of the expected difference in the performance of Algencan when second derivatives are not provided, we solved the same problems without considering the coded Hessians. A solution was found after 23 outer iterations. The solver of the Augmented Lagrangian subproblems achieved its maximum number of iterations when solving the subproblems of the second and third outer iterations. It also stopped by lack of progress when solving the subproblems of outer iterations 11 to 17. As a whole, Algencan used 29,984 calls to `fsub`, 6,918 calls to `gsub`, 30,074 calls to `csub` in average, 6,918 calls to `jacsub` in average, and 17.11 seconds of CPU time (four times the time required when using second-order derivatives). Of course, sometimes, second derivatives may not be available and running Algencan without them is the only possible choice. In such a situation, modifying the relatively strict values of parameters `epsfeas` and `epsopt` used in the present example may be a reasonable course of action.

To end this section, we show in Figure 13.4(b) the result when the target values $\bar{\beta}_j$ are the population of each region. Algencan used 13 outer iterations (429 inner iterations, 1,001 calls to `fsub` subroutine, 621 calls to `gsub` subroutine, 472 calls to `hsub` subroutine, 1,110 calls to subroutine `csub` per constraint in average, 621 calls to subroutine `jacsub` per constraint in average, and 472 calls to subroutine `hcsub` per constraint in average) and 2.38 seconds of CPU time.

13.3 ▪ Optimal control

Optimal control deals with the problem of finding a control law for a given system such that some functional cost is minimized. The problem includes state and control variables. The state variables (which describe the physical system) are solutions of a differential system of equations and the control variables should be chosen in order to optimize some criterion defined by the cost. The differential equations are known as dynamic constraints. The system may also include "path constraints" and boundary conditions. Many relevant engineering problems may be described by the control framework. Typical examples involve minimizing traveling times or fuel consumption of vehicles, from ordinary cars to rockets and satellites.

In this section, we consider control problems of the form

$$\begin{array}{ll} \text{Minimize} & \displaystyle\int_{t_0}^{t_f} f_0(s(t),u(t))\,dt \\ \text{subject to} & \dot{s}(t) = F(s(t),u(t)), \\ & s(t_0) = s_0, \end{array} \tag{13.12}$$

where the state variable is $s(t) \in \mathbb{R}^{n_s}$, $\dot{s} = ds/dt$, the control variable is $u(t) \in \mathbb{R}^{n_u}$, t varies between t_0 and t_f, $f_0 : \mathbb{R}^{n_s} \times \mathbb{R}^{n_u} \to \mathbb{R}$, and $F : \mathbb{R}^{n_s} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_s}$. The initial state is given by $s_0 \in \mathbb{R}^{n_s}$.

13.3.1 ▪ Discretization of the optimal control problem

Frequently, shooting techniques [21, 242] are employed for solving control problems. Unfortunately, shooting procedures are prone to severe ill-conditioning and usually require very good initial approximations [160]. A popular approach to overcoming these inconveniences consists of discretizing the domain $[t_0, t_f]$ with discretization of the derivatives of $s(t)$ and the conversion of the problem into a finite-dimensional optimization problem with as many dynamic constraints as discretization points [137].

We subdivide the time domain $[t_0, t_f]$ into N intervals with equidistant points $t_i = t_{i-1} + \Delta t$ or, equivalently, $t_i = t_0 + i\ \Delta t$, $i = 0, \ldots, N$, where $\Delta t = (t_f - t_0)/N$ and, hence, $t_N = t_f$. Considering the Euler discretization scheme $s_{i+1} = s_i + \Delta t F(s_i, u_i)$ and approximating the integral in the objective function of (13.12) by its Riemann sum, we arrive at the discretized optimal control problem

$$\begin{array}{ll} \text{Minimize} & \Delta t \sum_{i=0}^{N-1} f_0(s_i, u_i) \\ \text{subject to} & s_{i+1} = s_i + \Delta t F(s_i, u_i), i = 0, \ldots, N-1, \end{array}$$

where s_0 is given, the variables s_i approximate the states $s(t_i)$ for $i = 1, \ldots, N$, and the variables u_i approximate the controls $u(t_i)$ for $i = 0, \ldots, N-1$. The number of variables is $n = (n_s + n_u)N$ and the number of (equality) constraints is $m = n_s N$. Higher-order discretization schemes, such as the ones in the Runge–Kutta family of methods, can also be used.

13.3.2 ▪ Van der Pol system with unbounded control

As a simple example, we consider the van der Pol system with unbounded control (see, for example, [159]) given by

$$\begin{aligned} \dot{x}(t) &= y(t), \\ \dot{y}(t) &= -x(t) - (x(t)^2 - 1)y(t) + u(t), \end{aligned}$$

where $t \in [0, 1]$, $s(t) \equiv (x(t), y(t))^T \in \mathbb{R}^2$ describes the state of the system and $u(t) \in \mathbb{R}$ is the control. The aim is to minimize

$$\frac{1}{2} \int_0^1 (x(t)^2 + y(t)^2 + u(t)^2)\, dt.$$

The initial condition is given by $s_0 = (-2, 4)^T$.

The discretized optimal control problem (as described in the previous subsection) is given by

$$\begin{array}{lll} \text{Minimize} & \frac{1}{2} \Delta t \sum_{i=0}^{N-1} (x_i^2 + y_i^2 + u_i^2) & \\ \text{subject to} & x_{i+1} = x_i + \Delta t\ y_i, & i = 0, \ldots, N-1, \\ & y_{i+1} = y_i + \Delta t\ (-x_i - (x_i^2 - 1)y_i + u_i), & i = 0, \ldots, N-1, \end{array} \tag{13.13}$$

where $x_0 = -2$ and $y_0 = 4$ are constants. This problem has $n = 3N$ variables ($x_i, y_i, i = 1, \ldots, N$ and $u_i, i = 0, \ldots, N-1$) and $m = 2N$ constraints.

An equivalent formulation of (13.13) that is usually considered in practice is given by

$$\begin{array}{lll} \text{Minimize} & z_N & \\ \text{subject to} & x_{i+1} = x_i + \Delta t\ y_i, & i = 0, \ldots, N-1, \\ & y_{i+1} = y_i + \Delta t\ (-x_i - (x_i^2 - 1) y_i + u_i), & i = 0, \ldots, N-1, \\ & z_{i+1} = z_i + \Delta t\ (x_i^2 + y_i^2 + u_i^2)/2, & i = 0, \ldots, N-1, \end{array} \tag{13.14}$$

where $x_0 = -2$, $y_0 = 4$, and $z_0 = 0$ are constants. Problem (13.14) has $n = 4N$ variables ($x_i, y_i, z_i, i = 1, \ldots, N$, and $u_i, i = 0, \ldots, N-1$) and $m = 3N$ constraints.

Modern optimization methods for solving problems of this type may be found in [159] and [27]. Moreover, computationally more challenging versions of this problem can be considered by imposing constraints on the state and control variables.

13.3.3 ▪ A first run

To illustrate the usage of Algencan, we coded subroutines `fsub`, `gsub`, `hsub`, `csub`, `jacsub`, and `hcsub` for problems (13.13) and (13.14). Coded subroutines are part of the Algencan distribution and can be found as files `chap13-control.f90` and `chap13-control2.f90` (within folder `sources/examples/f90/`) for problems (13.13) and (13.14), respectively. In both cases first and second derivatives are very simple. Computing the values of parameters `jcnnzmax` and `hnnzmax` is also simple and possible values are `jcnnzmax` $= 7N$ and `hnnzmax` $= 21N$ for problem (13.13) and `jcnnzmax` $= 12N$ and `hnnzmax` $= 37N$ for problem (13.14). Setting the remaining parameters of Algencan as `epsfeas` = `epsopt` $= 10^{-8}$, `efstain` $= \sqrt{\texttt{epsfeas}}$, `eostain` = `epsopt`$^{1.5}$, `efacc` $= \sqrt{\texttt{epsfeas}}$, and `eoacc` $= \sqrt{\texttt{epsopt}}$, `outputfnm = ''`, `specfnm = ''`, and `nvparam` $= 0$, we are ready to run our first examples.

In a first set of numerical experiments, we considered $N \in \{10, 100, 1{,}000, 2{,}000, 3{,}000\}$. Table 13.2 shows some figures. In all cases Algencan found a solution with the required precision, and solutions were found as the result of the acceleration process. The last column in the table corresponds to the objective function value at the solution. The values of `cnorm`, `snorm`, and `nlpsupn` as defined in (10.26) are not shown because of lack of space, but they satisfy the stopping criterion of the acceleration process, i.e., they fulfill (10.21), (10.22). As expected, solutions to both models coincide. Regarding the remaining columns in the table, outit is the number of outer iterations, innit is the total number of inner iterations, fcnt, gcnt, and hcnt are, respectively, the number of calls to subroutines `fsub`, `gsub`, and `hsub`. Finally, ccnt, jcnt, and hccnt are the average number of calls to subroutines `csub`, `jacsub`, and `hcsub` per constraint, and Time is the CPU time in seconds.

The figures in Table 13.2 call attention to all instances of model (13.13) and to the last instance of model (13.14) in which the ratio between the number of functional evaluations (fcnt) and the number of inner iterations (innit) is relatively high. This is a symptom of poor descent directions when trying to solve the Augmented Lagrangian subproblems. Along those directions, many functional evaluations are spent in painful backtracking processes with a very slow decrease of the objective function. Many times the backtracking processes and, in consequence, the solver of the Augmented Lagrangian subproblems stop because of lack of progress in the objective function value or a very small step in the line search. This kind of behavior may be related to ill-conditioned (sub)problems. (However, if you observe a similar behavior of Algencan in your problem, the first step is to check the derivatives of the objective function and the constraints!) A detailed output to check the behavior of the subproblems' solver may be obtained with the keyword ITERATIONS-OUTPUT-DETAIL followed by, for example, the printing code number 19.

Table 13.2. *Computational effort measures of the application of Algencan to small and medium-scale instances of the control problems (13.13) and (13.14).*

Optimal control model (13.13)										
N	outit	innit	fcnt	gcnt	hcnt	ccnt	jcnt	hccnt	Time	$f(x^*)$
10	15	99	603	176	104	642	176	102	0.03	4.6139D+00
100	14	110	559	188	116	600	188	114	0.19	5.4477D+00
1,000	45	979	10,889	1,142	982	10,956	1142	979	18.19	5.5349D+00
2,000	24	142	1,120	326	239	1,147	326	234	18.29	5.5397D+00
3,000	20	105	861	213	145	883	213	140	32.24	5.5413D+00

Optimal control model (13.14)										
N	outit	innit	fcnt	gcnt	hcnt	ccnt	jcnt	hccnt	Time	$f(x^*)$
10	9	60	98	100	65	112	100	63	0.01	4.6139D+00
100	9	80	136	125	86	154	125	84	0.15	5.4477D+00
1,000	7	63	99	94	68	108	94	66	4.90	5.5349D+00
2,000	8	56	98	90	61	108	90	59	18.86	5.5397D+00
3,000	9	342	2,236	381	347	2,249	381	326	476.89	5.5413D+00

The most significant digit equal to 1 means that a single line will be printed for each Augmented Lagrangian (outer) iteration. The less significant digit equal to 9 means that you will get all possible details of the iterations of the subproblems' solver (inner iterations) (see Section 12.2). In order to check the coded derivatives against finite difference approximations, set the logical parameter `checkder` equal to true.

In the particular case of the instances mentioned in the paragraph above, the solver of the Augmented Lagrangian subproblems stops because of lack of progress in most of the subproblems. The meaning of lack of progress (for the subproblems' solver) embedded in Algencan is related to the progress in the objective function value, the norm of its projected gradient, and the size of the step (difference between consecutive iterates). If those quantities appear to be stuck for a certain number of consecutive iterations, the lack of progress is characterized and the subproblems' solver stops. However, the constants associated with determining that some of those values are "stuck" and the amount of consecutive iterations with "no progress" that must occur in order to characterize the lack of progress is arbitrary. Modifying those constants to determine the lack of progress in advance greatly improves the performance of Algencan in those instances. Unfortunately, in some other situations, a premature stop by lack of progress in the subproblems may impair the overall performance of Algencan.

13.3.4 ▪ Early acceleration

The figures in Table 13.2 do not point to a clear advantage of one of the models over the other. (Note that comparing the two models is completely outside the scope of these experiments.) However, the performance of Algencan for solving the control problem is clearly model dependent. When solving instances of model (13.14), solutions are always found in the first acceleration process. (The same does not happen with instances of model (13.13).) Moreover, in those cases, most of the time is spent in the Augmented Lagrangian iterations that precede the acceleration process. The question arises of whether to launch the acceleration process at the very beginning. Note that (refer to Section 10.2.4) setting the threshold parameters $\texttt{efacc} = \sqrt{\texttt{epsfeas}}$ and $\texttt{eoacc} = \sqrt{\texttt{epsopt}}$ (as we did in these numerical experiments) has no practical effect, since it reduces to the already default choice of Algencan, which is to start launching the acceleration process after having achieved half the required feasibility and optimality tolerances. In the following experiment, we show the behavior of Algencan under a different choice: launch the acceleration process from the beginning, i.e., starting at the given initial point and even previously to the first Augmented Lagrangian iteration. As described in Section 10.2.4, this run of Algencan can be obtained by setting parameters `efacc` and `eoacc` with large values like `efacc` = `eoacc` = 10^{20}. Table 13.3 shows the results.

Table 13.3. *Computational effort measures of the application of Algencan, with early launching of the acceleration process, to large-scale instances of the control problem model (13.14).*

N	nwtkktit	fcnt	gcnt	hcnt	ccnt	jcnt	hccnt	Time	$f(x^*)$
10	8	10	11	8	10	11	7	0.00	4.6139D+00
100	10	12	13	10	12	13	9	0.02	5.4477D+00
1,000	10	12	13	10	12	13	9	0.12	5.5349D+00
2,000	10	12	13	10	12	13	9	0.23	5.5397D+00
3,000	9	11	12	9	11	12	8	0.31	5.5413D+00
10,000	9	11	12	9	11	12	8	1.03	5.5436D+00
100,000	8	10	11	8	10	11	7	9.00	5.5445D+00
1,000,000	7	9	10	7	9	10	6	88.26	5.5447D+00

Neither outer nor inner iterations of Algencan are done to solve the set of problems considered in Table 13.3. Therefore, in the table we show nwtkktit, the number of iterations of the Newton method applied to the KKT system of the problem. The largest problem in Table 13.2 is the one with $N = 3{,}000$. (N is the number of (sub)intervals in the discretization of the optimal control problem, the number of variables is $n = 4N$, and the number of constraints is $m = 3N$.) Figures in Tables 13.2 and 13.3 show that this problem is solved between three and four orders of magnitude faster with this setting of Algencan parameters than with the previous setting. Moreover, Table 13.3 shows additional instances with $N \in \{10^4, 10^5, 10^6\}$ from which it is very clear that the required CPU time grows linearly with respect to N. Once again, Algencan satisfied the required precision in all the instances and the obtained values of the objective function coincide with those reported in Table 13.2. Figure 13.5 illustrates the solution to the optimal control problem (found when solving the instance with $N = 1{,}000$ of model (13.14)).

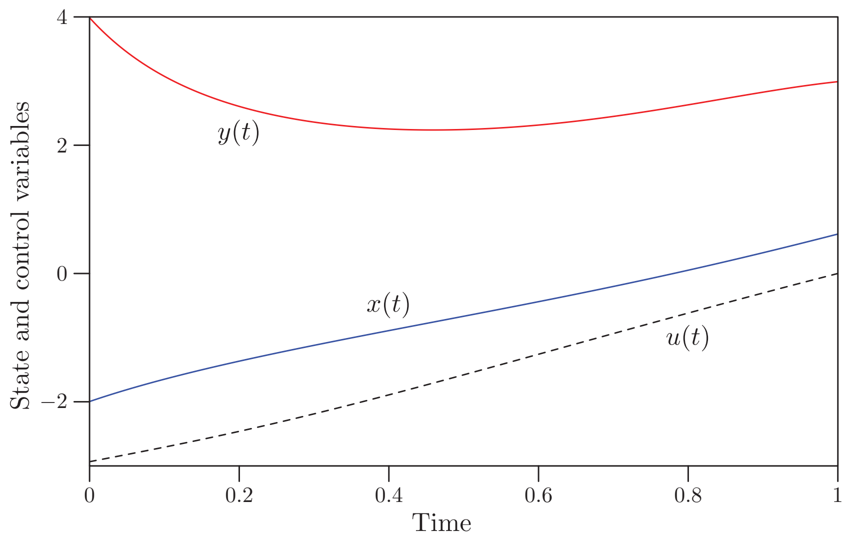

Figure 13.5. *Solution to the optimal control problem.*

13.4 ▪ Grid generation

Grid generation (also called mesh generation) is the art of generating a polygonal or polyhedral mesh that approximates a geometric domain. Typical uses are the representation of a 3D surface on a computer screen or the generation of meshes for finite element computations in computational fluid dynamics.

A curvilinear grid or structured grid is a grid with the same combinatorial structure as a regular grid, in which the cells are quadrilaterals or cuboids rather than rectangles or rectangular parallelepipeds.

Optimal, or well-behaved, meshes are crucial for the solution of partial differential equations in two-dimensional complex regions. A close correlation between the mesh quality and the errors obtained when solving elliptic equations using different meshes has been observed in [215].

Variational methods to generate good conforming meshes on curved regions have shown some difficulties for complicated regions, producing folded, crimped, or rough meshes [215]. Castillo [76] showed that optimizing a combination of several criteria produces better-behaved meshes than optimizing any single criterion alone. In [215], two criteria were employed for the generation of conforming logically rectangular meshes in two-dimensional (2D) curved regions: Minimization of the sum of the squares of all the cell sides and minimization of the sum of squares of all the cell areas.

For a 2D domain, for which a suitable bijection with a rectangle exists, a discrete set of points in its boundary, in the form $P(i,j)$ with $j \in \{1, n_{\text{ord}}\}$ and $i = 1,\ldots,n_{\text{abs}}$ and $i \in \{1, n_{\text{abs}}\}$ and $j = 1,\ldots,n_{\text{ord}}$, is given. The 2D domain is, in fact, defined by this finite set of points in its boundary. The interior points $(P(i,j), i = 2,\ldots,n_{\text{abs}}-1, j = 2, n_{\text{ord}}-1)$ are the unknowns of the problem. See Figure 13.6. The region in Figure 13.6 corresponds to the example in [76, p. 466]. For completeness and reproducibility of the numerical experiments that will be presented below, it is necessary to know that the boundary of the region is given by a unitary-radius half-circle and a half-ellipse with semiaxes $a = 6$ and $b = 3$.

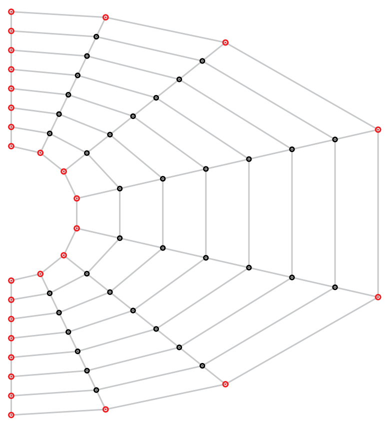

Figure 13.6. *Points in the boundary are given and define the 2D domain. The interior points are the unknowns. The depicted configuration of the interior points is the natural choice of the initial guess for an optimization process.*

The deformed rectangles (cells) have vertices $P(i,j)$, $P(i+1,j)$, $P(i+1,j+1)$, and $P(i,j+1)$ for $i = 1,\ldots,n_{\text{abs}}-1$ and $j = 1,\ldots,n_{\text{ord}}-1$. As is well known, if the Cartesian coordinates of $P(i,j)$ are $p_{ij} = (p_{ij}^x, p_{ij}^y)^T \in \mathbb{R}^2$, the area a_{ij} enclosed by a cell whose vertices in counterclockwise order are $P(i,j)$, $P(i+1,j)$, $P(i+1,j+1)$, and $P(i,j+1)$

is given by

$$\begin{aligned} a_{ij} \;=\; & (p^x_{ij}\, p^y_{i+1,j} - p^x_{i+1,j}\, p^y_{ij}) && + \; (p^x_{i+1,j}\, p^y_{i+1,j+1} - p^x_{i+1,j+1}\, p^y_{i+1,j}) \\ & + (p^x_{i+1,j+1}\, p^y_{i,j+1} - p^x_{i,j+1}\, p^y_{i+1,j+1}) && + \; (p^x_{i,j+1}\, p^y_{i,j} - p^x_{i,j}\, p^y_{i,j+1}). \end{aligned}$$

Let $x = (p_{11}^T, \dots, p_{1,n_{ord}}^T, p_{21}^T, \dots, p_{2,n_{\text{ord}}}^T, \dots p_{n_{\text{abs}},1}^T, \dots, p_{n_{\text{abs}},n_{\text{ord}}}^T)^T \in \mathbb{R}^n$ with $n = 2n_{\text{abs}}n_{\text{ord}}$ and define

$$f_S(x) = \frac{1}{c_S}\left[\frac{1}{2}\sum_{i=1}^{n_{\text{abs}}-1}\sum_{j=1}^{n_{\text{ord}}} \|p_{ij} - p_{i+1,j}\|^2 + \frac{1}{2}\sum_{j=1}^{n_{\text{ord}}-1}\sum_{i=1}^{n_{\text{abs}}} \|p_{ij} - p_{i,j+1}\|^2\right] \tag{13.15}$$

and

$$f_A(x) = \frac{1}{c_A}\left[\frac{1}{2}\sum_{i=1}^{n_{\text{abs}}-1}\sum_{j=1}^{n_{\text{ord}}-1} a_{ij}^2\right], \tag{13.16}$$

where $c_S = (n_{\text{abs}} - 1)n_{\text{ord}} + (n_{\text{ord}} - 1)n_{\text{abs}}$ and $c_A = (n_{\text{abs}} - 1)(n_{\text{ord}} - 1)$.

The *unconstrained* optimization problem defined in [215] is given by

$$\text{Minimize } \gamma f_S(x) + (1-\gamma) f_A(x), \tag{13.17}$$

where $\gamma \in [0, 1]$ is a given constant. It corresponds to a convex combination of the average of the squared sides of the cells and the average of the squared areas of the cells. Note that the points in the boundary are fixed and (although included in x) are not variables of the optimization problem. Hence, cell sides that correspond to a pair of points in the boundary are fixed too and they were included in the objective function for simplicity only. Figures 13.7(a)–13.7(c) show the solutions to problem (13.17) with $\gamma = 0$, $\gamma = 1$, and $\gamma = 0.1$, respectively, for the elliptical region depicted in Figure 13.6 with a 25×25 grid (i.e., $n_{\text{abs}} = n_{\text{ord}} = 25$). This problem has $23 \times 23 \times 2 = 1{,}058$ variables corresponding to the abscissae and ordinates of the grid inner points.

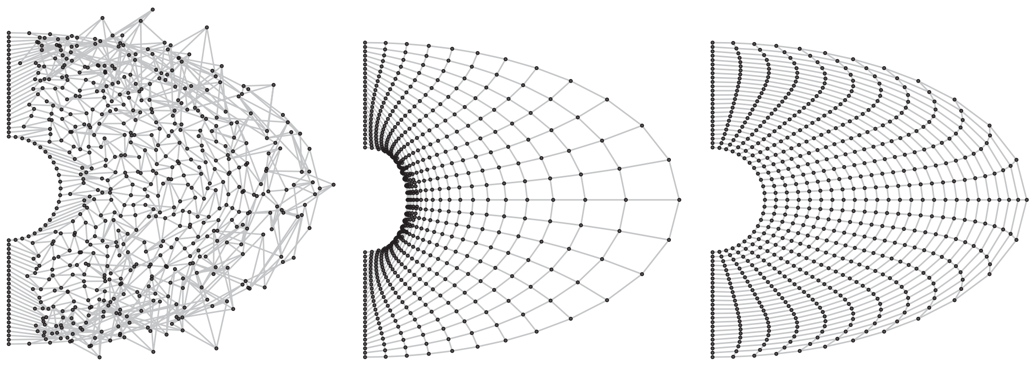

Figure 13.7. *(a) Solution to minimizing the average of the squared areas f_A. (b) Solution to minimizing the average of the squared sides f_S. (c) Solution to minimizing a convex combination with weight $\gamma = 0.1$ for the average of the squared sizes.*

The generated mesh depicted in Figure 13.7(c) illustrates, as observed in [76], that optimizing a combination of the two considered criteria (with the arbitrarily chosen parameter $\gamma = 0.1$) produces a better behaved mesh than those produced by the optimization of any of the two criteria individually. Observe that meshes in Figures 13.7(a) and 13.7(b) have mesh points outside the 2D region, while the mesh depicted in Figure 13.7(c) does not.

Problem (13.17) is unconstrained and a few words related to its resolution with Algencan are in order. When tackled by Algencan, any unconstrained or bound-constrained problem is solved by Algencan's subproblems' solver Gencan, outside the Augmented Lagrangians framework. Among the available methods for dealing with the subproblems (see Section 12.8), by reasons that will be clear soon, having coded first- and second-order derivatives, we opted for the Newton line-search strategy (keyword NEWTON-LINE-SEARCH-INNER-SOLVER). Other than the choice of the method used to solve the (sub)problem, the two relevant parameters of Algencan in the case of unconstrained or bound-constrained problems are the optimality tolerance ε_{opt} and the maximum number of (inner) iterations. The optimality tolerance corresponds, in the case of unconstrained and bound-constrained problems, to the required tolerance for the sup-norm of the gradient or the projected gradient of the objective function, respectively. In this experiment, it was arbitrarily set to 10^{-8}, i.e., `epsopt = 1.0d-08`. The maximum allowed number of inner iterations is an implicit or additional parameter whose value can be modified with the keyword INNER-ITERATIONS-LIMIT. Its default value, which was not modified in this experiment, is a huge number when the problem is unconstrained or bound constrained.

Regarding the output on the screen, by default Algencan shows a single line of information per Augmented Lagrangian (outer) iteration (see Section 12.2). This means that in the case of unconstrained and bound-constrained problems, in which there are no outer iterations, nothing is shown on the screen during the optimization process. In order to show a single line of information at each iteration of the subproblem's solver (i.e., at each inner iteration), the iterations' output detail should be set to the value 11 using the keyword ITERATIONS-OUTPUT-DETAIL followed by the integer value `11`. The relevant information here is, in fact, the less significant digit. Therefore, any value "ending" with `1` would have the same effect. Moreover, as already explained in Section 12.2, the larger the value of the digit, the larger the amount of information in the output. (The less significant digit corresponds to the inner iterations and the tens digit corresponds to the outer iterations.)

With the settings described in the paragraph above, Algencan found the solution to minimizing $f_A(\cdot)$, depicted in Figure 13.7(a), in 46 inner iterations and using 198 objective function evaluations, 68 gradient evaluations, 46 Hessian evaluations, and 1.01 seconds of CPU time. Naming the solution found as x_A^*, we have that $f_S(x_A^*) \approx 1.87 \times 10^{-1}$ and $f_A(x_A^*) \approx 4.25 \times 10^{-3}$. The solution to minimizing $f_S(\cdot)$, depicted in Figure 13.7(b), was found by Algencan using 2 inner iterations, 3 objective function evaluations, 4 gradient evaluations, 2 Hessian evaluations, and 0.02 seconds of CPU time. Naming the solution found as x_S^*, we have that $f_S(x_S^*) \approx 2.97 \times 10^{-2}$ and $f_A(x_S^*) \approx 1.90 \times 10^{-2}$. The solution to the minimization of the convex combination of both objectives (depicted in Figure 13.7(c)) was found using 4 iterations of Gencan, 5 objective function evaluations, 6 gradient evaluations, 4 Hessian evaluations, and 0.04 seconds of CPU time. Naming the solution as x_γ^*, we have that $f_S(x_\gamma^*) \approx 4.68 \times 10^{-2}$ and $f_A(x_\gamma^*) \approx 4.91 \times 10^{-3}$. The most relevant information of the numerical results mentioned in this paragraph is that minimizing $f_A(\cdot)$ alone appears to be much more time-consuming than minimizing $f_S(\cdot)$ or a combination of both. It seems as if $f_S(\cdot)$ would play a regularization role in the optimization process.

13.4.1 ▪ Constrained formulations

Consider the problem

$$\begin{array}{ll} \text{Minimize} & \xi_1 f_A(x) + \xi_2 f_S(x) \\ \text{subject to} & \xi_3 f_A(x) + \xi_4 f_S(x) \le \xi_5, \\ & \ell \le x \le u, \end{array} \tag{13.18}$$

where $\xi_1, \dots, \xi_5 \ge 0$ are given constants. Bound constraints serve only the purpose of fixing the boundary points. This means that for all t, if x_t is a component (abscissa or ordinate) of an interior point, we have $\ell_t = -\infty$ and $u_t = +\infty$, and if x_t is a component of a boundary point, then we have $\ell_t = u_t$. These kinds of fixed variables are not variables at all and they are presented only to simplify the definition of the problem. As mentioned in Section 12.6, by default Algencan eliminates those variables from the problem to be solved, in a preprocessing stage.

If $\xi_1 + \xi_2 = 1$, for adequate values of ξ_3, ξ_4, and ξ_5 (such as $\xi_3 = \xi_4 = \xi_5 = 0$), problem (13.18) coincides with problem (13.17), which deals with a weighted sum of the objective functions of the biobjective problem of minimizing $f_A(x)$ and $f_S(x)$. If we consider the ϵ-constraint method for multiobjective optimization [183, 141], the problems to be handled are of the form

$$\text{Minimize } f_A(x) \text{ subject to } f_S(x) \le \delta_S \tag{13.19}$$

and

$$\text{Minimize } f_S(x) \text{ subject to } f_A(x) \le \delta_A, \tag{13.20}$$

where δ_S and δ_A are given constants. Both problems are also particular cases of problem (13.18) for suitable (trivial) choices of constants $\xi_1, \dots, \xi_5$.

The ϵ-constraint method consists of (i) finding a solution x_S^* to minimizing $f_S(x)$ (which corresponds to solving problem (13.18) with $\xi_2 = 1$ and $\xi_1 = \xi_3 = \xi_4 = \xi_5 = 0$), and (ii) solving problem (13.19) with $\delta_S = (1 + \Delta_S) f_S(x_S^*)$ varying $\Delta_S \ge 0$ (which corresponds to solving problem (13.18) with $\xi_1 = \xi_4 = 1$, $\xi_2 = \xi_3 = 0$, and $\xi_5 = \delta_S$). The procedure considering problem (13.20) is analogous.

13.4.2 ▪ Coding the problem

Generating a grid by any of the methods discussed in the previous sections consists of solving one or more instances of problems (13.17), (13.19), and (13.20). All of these are particular cases of problem (13.18). Therefore, it appears that problem (13.18) might be a valuable tool for the generation of meshes. In the rest of this section, we show how to code and solve problem (13.18) using Algencan.

The objective function and the constraint of problem (13.18) differ only in the value of the constants and share *all* nontrivial expressions. Therefore, it is very natural to code them together, opting by coding subroutines `fcsub`, `gjacsub`, and `hlsub` to compute (a) the objective function and the constraint, (b) the gradient of the objective function and the gradient (Jacobian) of the constraint, and (c) the Hessian of the Lagrangian, respectively. See Section 10.2.10.

Problem (13.18) has $n = n_{\text{abs}} n_{\text{ord}}$ variables (although $n_{\text{abs}} n_{\text{ord}} - (n_{\text{abs}} - 1)(n_{\text{ord}} - 1)$ are fixed and will be eliminated by Algencan) and a single inequality constraint. The objective function $f(x)$ is given by

$$f(x) = \xi_1 f_A(x) + \xi_2 f_S(x)$$

and the inequality constraint $c_1(x) \le 0$ is given by

$$c_1(x) = \xi_3 f_A(x) + \xi_4 f_S(x) - \xi_5.$$

The objective function and the (inequality) constraint are easy to code and do not deserve any special comment. The only relevant decision is that, to avoid thinking in special cases, all $n_{\text{abs}} n_{\text{ord}}$ points in the grid are considered as variables. Then, points in the boundary are fixed (and thus eliminated from the optimization process by Algencan) by appropriately setting their bounds. Variables corresponding to inner points have no bound constraints. The gradients of the objective function and of the constraint are dense n-dimensional arrays. The density of the single-row Jacobian of the constraint will be addressed when dealing with the value of Algencan's parameter `hnnzmax`.

Subroutine `hlsub` must compute the lower triangle of the sparse matrix

$$s_f \nabla^2 f(x) + s_{c_1} \lambda_1 \nabla^2 c_1(x), \tag{13.21}$$

where $\nabla^2 f(x)$ is the Hessian matrix of the objective function $f(x)$ and $\nabla^2 c_1(x)$ is the Hessian matrix of the constraint $c_1(x)$. Parameters s_f, s_{c_1}, and λ_1 are input parameters of subroutine `hlsub` and they correspond to the scaling factor of the objective function, the scaling factor of the constraint, and the Lagrange multiplier of the constraint, respectively (see Section 10.2.10). Since

$$\nabla^2 f(x) = \xi_1 \nabla^2 f_A(x) + \xi_2 \nabla^2 f_S(x)$$

and

$$\nabla^2 c_1(x) = \xi_3 \nabla^2 f_A(x) + \xi_4 \nabla^2 f_S(x),$$

we have that (13.21) coincides with

$$\left(s_f \xi_1 + s_{c_1} \lambda_1 \xi_3\right) \nabla^2 f_A(x) + \left(s_f \xi_2 + s_{c_1} \lambda_1 \xi_4\right) \nabla^2 f_S(x). \tag{13.22}$$

Thus, the simpler way to compute the desired Hessian matrix of the scaled Lagrangian is first to compute $\nabla^2 f_A(x)$ and $\nabla^2 f_S(x)$ and then to multiply them by the constants in (13.22). The possibility of returning more than a single triplet for each matrix element (see Section 10.2.10) releases the user from the cumbersome task of efficiently computing the sum of both matrices.

Needless to say, although first- and second-order derivatives of the problem at hand are simple, coding derivatives (in particular coding sparse Hessians) is prone to error. Therefore, we admit that we made extensive use of the checking derivatives feature of Algencan, setting `checkder = .true.` until obtaining, after several rounds, correct codes for the derivatives.

13.4.3 ▪ Setting Algencan's parameters

By default, without a second thought, we set `epsfeas` = `epsopt` = 10^{-8}, `efstain` = $\sqrt{\texttt{epsfeas}}$, `eostain` = `epsfeas`$^{1.5}$, `efacc` = $\sqrt{\texttt{epsfeas}}$, `eoacc` = $\sqrt{\texttt{epsopt}}$, `outputfnm = ''`, and `specfnm = ''`. We also set, for a reason that will be elucidated below and is related to the value of parameter `hnnzmax`,

```
nvparam = 1
vparam(1) = 'NEWTON-LINE-SEARCH-INNER-SOLVER'
```

The value of Algencan's input parameter `jcnnzmax`, which must be an upper bound on the number of triplets used to represent the Jacobian of the constraints coded by the user, is set to n, since the gradient of the constraint is dense, i.e., we set `jcnnzmax = n`.

As mentioned in Section 10.2.11, the value of parameter hnnzmax depends on which subroutines are coded to represent the problem and on which method is used to solve the (Augmented Lagrangian) subproblems. On the one hand, hnnzmax must be an upper bound on the number of triplets needed to store the lower triangle of the Hessian of the scaled Lagrangian computed within subroutine hlsub, since this is the way we chose to code the second derivatives of the problem. On the other hand, in addition, extra space may be required to store some information related to the Jacobian $c'(x)$, which, together with the Hessian of the Lagrangian, is needed to obtain the Hessian of the *Augmented* Lagrangian (see (12.3)). If the Euclidean trust-region approach is used to solve the subproblems, then there must be space to store the lower triangle of the (symmetric) matrix $c'(x)^T c'(x)$, which in the present case is a dense $n \times n$ matrix. If the Newton line-search or any other strategy is used to solve the subproblems, no extra space needs to be considered when setting the parameter hnnzmax.

The choice of the approach used by Algencan to solve the subproblems is made, by default, by a simple and arbitrary rule based on the number of variables n (see Section 12.8). The Euclidean trust-region approach is used for small problems (the paragraph above explains this choice, at least partially) and the Newton (or truncated Newton) line-search approach is used for large problems. This is why, since we are thinking in large-scale instances of the grid generation problem, for which an $O(n^2)$ memory requirement would not be affordable, we used the Newton line-search approach. Inhibiting the Algencan default choice and fixing the subproblems' solver as the Newton line-search strategy, we are able to set parameter hnnzmax as an upper bound on the number of triplets needed to store the lower triangle of the Hessian of the Lagrangian only. Therefore, we set

```
hnnzmaxS = 6 * ( 2 * nabs * nord - nabs - nord )
hnnzmaxA = 36 * ( nabs - 1 ) * ( nord - 1 )
hnnzmax  = hnnzmaxS + hnnzmaxA
```

where nabs and nord correspond to $n_{\rm abs}$ and $n_{\rm ord}$, respectively, and hnnzmaxS and hnnzmaxA correspond to the number of triplets used to store (the lower triangle of) the Hessian matrices of f_S and f_A, respectively. The given values for hnnzmaxS and hnnzmaxA may not be trivial to see unless you code the Hessians by yourself or at least check the particular implementation of the Hessian of the Lagrangian considered in subroutine hlsub. Coded subroutines are part of the Algencan distribution and can be found as file chap13-grid-ellip.f90 (within folder sources/examples/f90/).

13.4.4 ▪ Solving a sequence of ϵ-constrained problems

As an example of the many possibilities of considering problem (13.18) to generate grids, based on the results reported in [76], we considered the problem of minimizing $f_S(x)$ subject to optimality or "near optimality" of $f_A(x)$, i.e., problem (13.20) (or problem (13.18) with $\xi_1 = \xi_4 = 0$, $\xi_2 = \xi_3 = 1$, and $\xi_5 = \delta_A$) with $n_{\rm abs} = n_{\rm ord} = 25$, $\delta_A = (1+\Delta_A) f_S(x_A^*)$, and $\Delta_A \in \{0.0, 0.1, 0.2, \dots, 0.9\}$, where x_A^* is the solution to minimize $f_A(x)$ (with no constraints) found at the beginning of the present section and depicted in Figure 13.7(a). In the numerical experiments, we considered $f_A(x_A^*) = 4.25251962159732197 \times 10^{-3}$.

Figure 13.8 shows the solution to minimize $f_S(x)$ subject to $f_A(x) \le f_A(x_A^*)$, i.e., with $\Delta_A = 0$. Of course, the constraint is active at the solution. The value of f_S at the solution is approximately 6.32×10^{-2} and, although it has some irregularities, the generated mesh appears to be similar to the one in Figure 13.7(c). Figure 13.9 shows the solutions to minimize $f_S(x)$ subject to $f_A(x) \le (1+\Delta_A) f_A(x_A^*)$ for increasing values of Δ_A. As expected, as Δ_A increases, the solutions increasingly resemble the solution to minimize $f_S(x)$ with no constraints, depicted in Figure 13.7(b). Note that, relaxing the optimality of f_A a little bit, the irregularities depicted on Figure 13.8 are avoided.

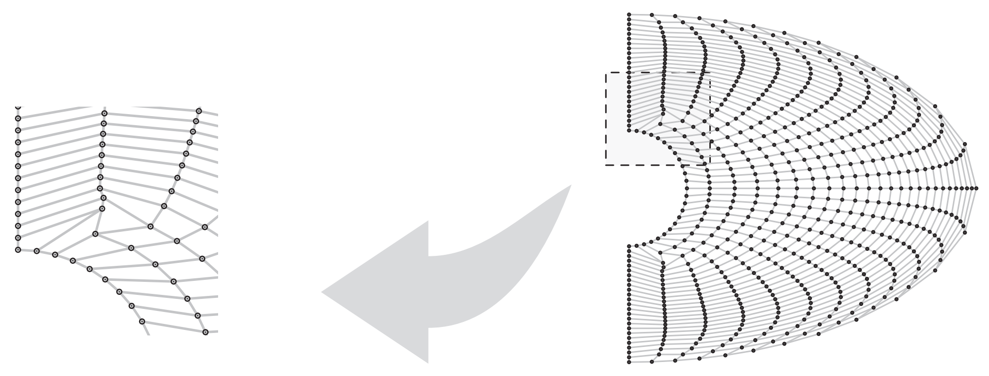

Figure 13.8. *Solution to the constrained problem of minimizing $f_S(x)$ subject to $f_A(x) \leq f_A(x_A^*)$, where x_A^* is the optimal solution of minimizing $f_A(x)$ without constraints. The zoom shows an "irregular" region of the mesh in detail.*

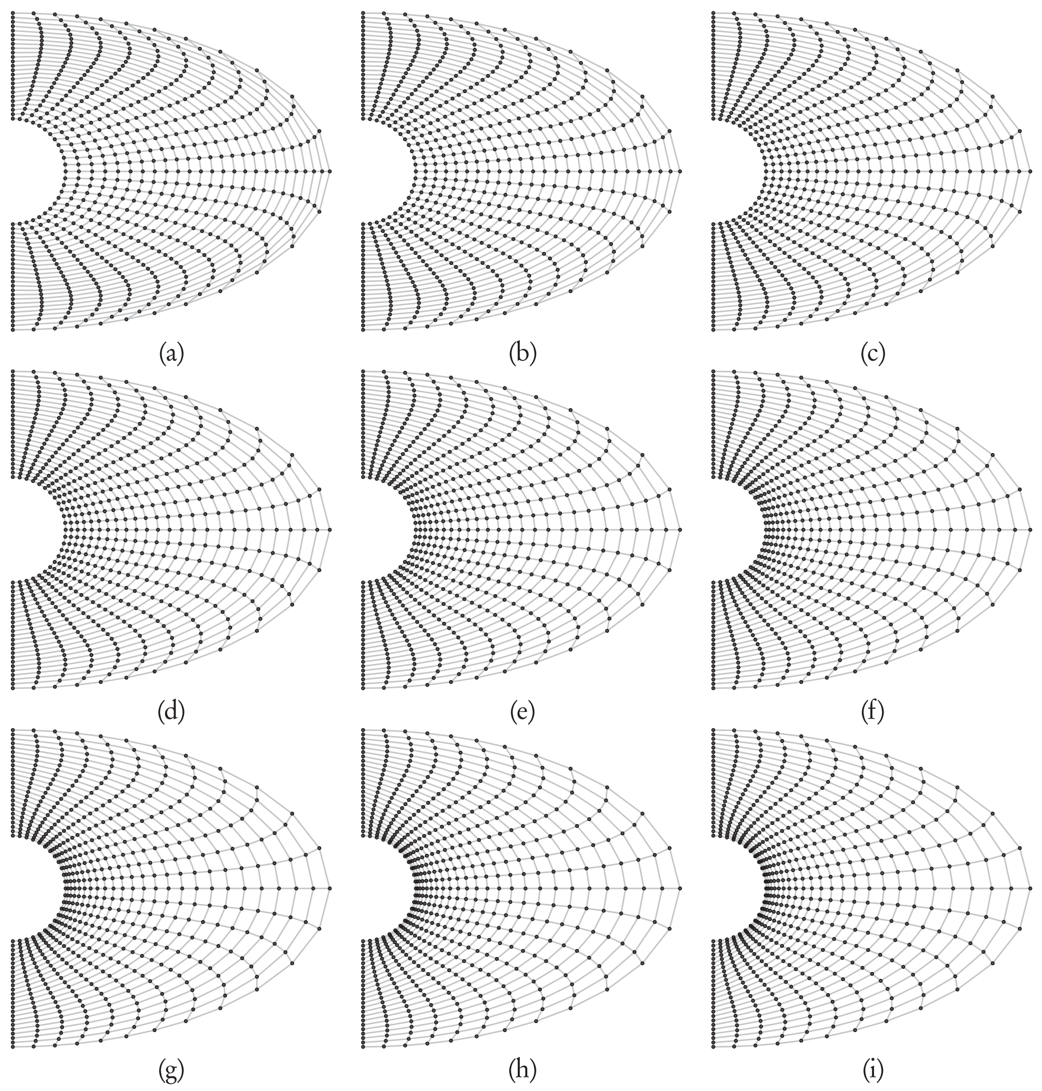

Figure 13.9. *Solutions to the constrained problem of minimizing $f_S(x)$ subject to $f_A(x) \leq (1+\Delta_A)f_A(x_A^*)$, where x_A^* is the optimal solution of minimizing $f_A(x)$ without constraints. Solutions depicted in (a)–(i) correspond to $\Delta_A = 0.1, 0.2, \ldots, 0.9$, respectively.*

Similar results can also be obtained for an 11×11 mesh in the hard horn-shaped 2D region depicted in Figure 13.10, which is known to be more difficult than the previous one, since it has no equal area solution [76]. For the sake of reproducibility, points in the border of the 2D domain are given by (a) a segment from $(1,1)^T$ to $(1,2)^T$ (left); (b) the arc from $(4,0)^T$ to $(4,4)^T$ of a circle centered at $(2.5,2)^T$ with radius 2.5 (right); (c) a parabola of the form $y = ax^2 + bx + c$ with $a = -5/21$, $b = -3.6a$, and $c = 1 + 2.6a$ (bottom); and (d) a parabola with $a = 10/21$, $b = -3.6a$, and $c = 1 + 2.6a$ (top). Figure 13.11 shows the solutions to minimize $f_S(x)$ subject to $f_A(x) \leq (1 + \Delta_A) f_A(x_A^*)$ for increasing values of Δ_A, where x_A^* is the optimal solution to minimizing $f_A(x)$ with no constraints. In the numerical experiments, we considered $f_A(x_A^*) = 1.00260900832775720 \times 10^{-2}$. Analyzing the quality of the generated meshes for this particular problem and determining the most adequate value of Δ_A are outside the scope of this work. Numerical experiments appear to show that problem (13.18) is a useful tool for developing meshes. Source codes related to the horn-shaped 2D region problem depicted in Figure 13.10 are part of the Algencan distribution and can be found as file `chap13-grid-horn.f90` (within folder `sources/examples/f90/`).

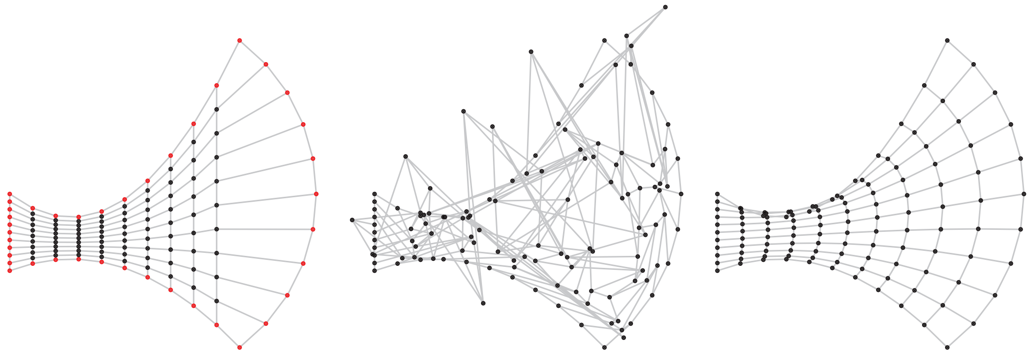

Figure 13.10. *(a) Given boundary points and "natural" initial guess. (b) Solution to minimizing the sum of the squared areas f_A. (c) Solution to minimizing the sum of the squared sides f_S.*

13.4.5 ▪ Solving a larger instance

Both problems considered in the previous section were taken from [76]. Merely increasing the number of points in the mesh (i.e., increasing n_{abs} and/or n_{ord}) is not enough to generate large instances of the optimization problem. This is because, in either of the two problems, if the area of the 2D domain remains fixed while the number of points in the mesh increases, the considered initial guess satisfies the stopping criteria with the prescribed tolerance $\varepsilon_{\mathrm{opt}} = 10^{-8}$. This serves as an alert to the fact that the value 10^{-8} for the optimality tolerance $\varepsilon_{\mathrm{opt}}$ is an arbitrary choice for a problem-dependent parameter.

In this section, we consider a 100×100 mesh in a scaled version of the horn-shaped region in which the boundary is given by (a) a segment from $(10,10)^T$ to $(10,20)^T$ (left); (b) the arc from $(40,0)^T$ to $(40,40)^T$ of a circle centered at $(25,20)^T$ with radius 25 (right); (c) a parabola of the form $y = ax^2 + bx + c$ with $a = -5/210$, $b = -36a$, and $c = -160a$ (bottom); and (d) a parabola with $a = 1/21$, $b = -36a$, and $c = 40 - 160a$ (top). The source files are available in the Algencan distribution as file `chap13-grid-horn-large.f90` within folder `sources/examples/f90/`.

Based on the experiments of the previous sections, we aim at first to solve the unconstrained problem of minimizing $f_A(x)$ with no constraints to obtain x_A^* and then to minimize $f_S(x)$ subject to $f_A(x) \leq (1 + 0.1) f_A(x_A^*)$.

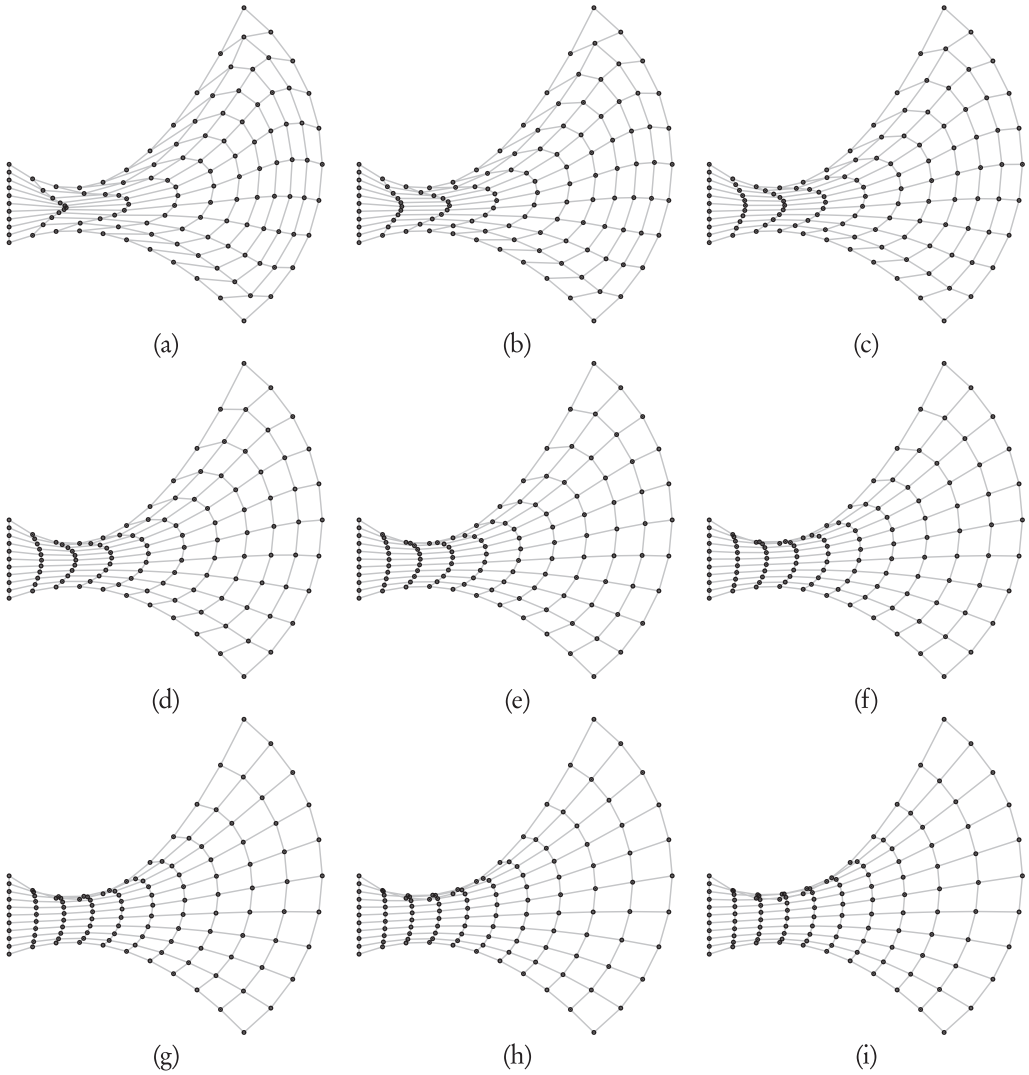

Figure 13.11. *Solutions to the constrained problem of minimizing $f_S(x)$ subject to $f_A(x) \le (1+\Delta_A) f_A(x_A^*)$, where x_A^* is the optimal solution of minimizing $f_A(x)$ without constraints. Solutions depicted in (a)–(i) correspond to $\Delta_A = 0.1, 0.2, \dots, 0.9$, respectively.*

The problem of minimizing $f_A(x)$ is an unconstrained problem with $n = 19{,}208$ variables. When tackled by Algencan, it is solved by the bound constraints solver Gencan. Setting $\varepsilon_{\text{opt}} = 10^{-8}$, Gencan took 565 iterations, 3,490 functional evaluations, 841 gradient evaluations, 565 Hessian evaluations, and 1,500.21 seconds of CPU time. Note that, in the case of unconstrained and bound-constrained problems, the number of (inner) iterations and the number of Hessian evaluations coincide if the inner-to-the-face strategy is the Newton's method with line search. At the solution x_A^* found, we have $f_A(x_A^*) \approx 1.04403487484171029 \times 10^{-2}$. Note that it is not very clear whether such precision would be needed in order to be used in the second step of the process of generating the mesh. A rough approximation $\bar{x}_A$, with $f_A(\bar{x}_A) \approx 1.08 \times 10^{-2}$, would have been found in less than half the time (177 inner iterations) setting $\varepsilon_{\text{opt}} = 10^{-4}$.

In the second stage of the mesh generation problem, we solved the problem of minimizing $f_S(x)$ subject to $f_A(x) \le (1+0.1) f_A(x_A^*)$ with $f_A(x_A^*) = 1.04403487484171029 \times 10^{-2}$.

Algencan solved the problem using 14 outer iterations, 75 inner iterations, 293 calls to subroutine `fcsub`, 167 calls to subroutine `gjacsub`, 105 calls to subroutine `hlpsub`, and 93.19 seconds of CPU time. In the case of this constrained problem, although the inner solver Gencan is using the Newton's method with line searches, the number of Hessians evaluations is larger than the number of inner iterations because some Hessians evaluations are used in unsuccessful (therefore, discarded) trials of the acceleration process. At the solution x_γ^* found, we have $f_S(x_\gamma^*) \approx 8.40936730531235827 \times 10^{-2}$ and $f_A(x_\gamma^*) \approx 1.14843878681980868 \times 10^{-2}$. Figures 13.12 and 13.13 illustrate the solution found. Note that, while the first-stage unconstrained problem appears to be a bit time-consuming (a drawback that can be circumvented by requiring a looser stopping-criterion tolerance), the constrained problem of the second stage is solved by Algencan relatively fast.

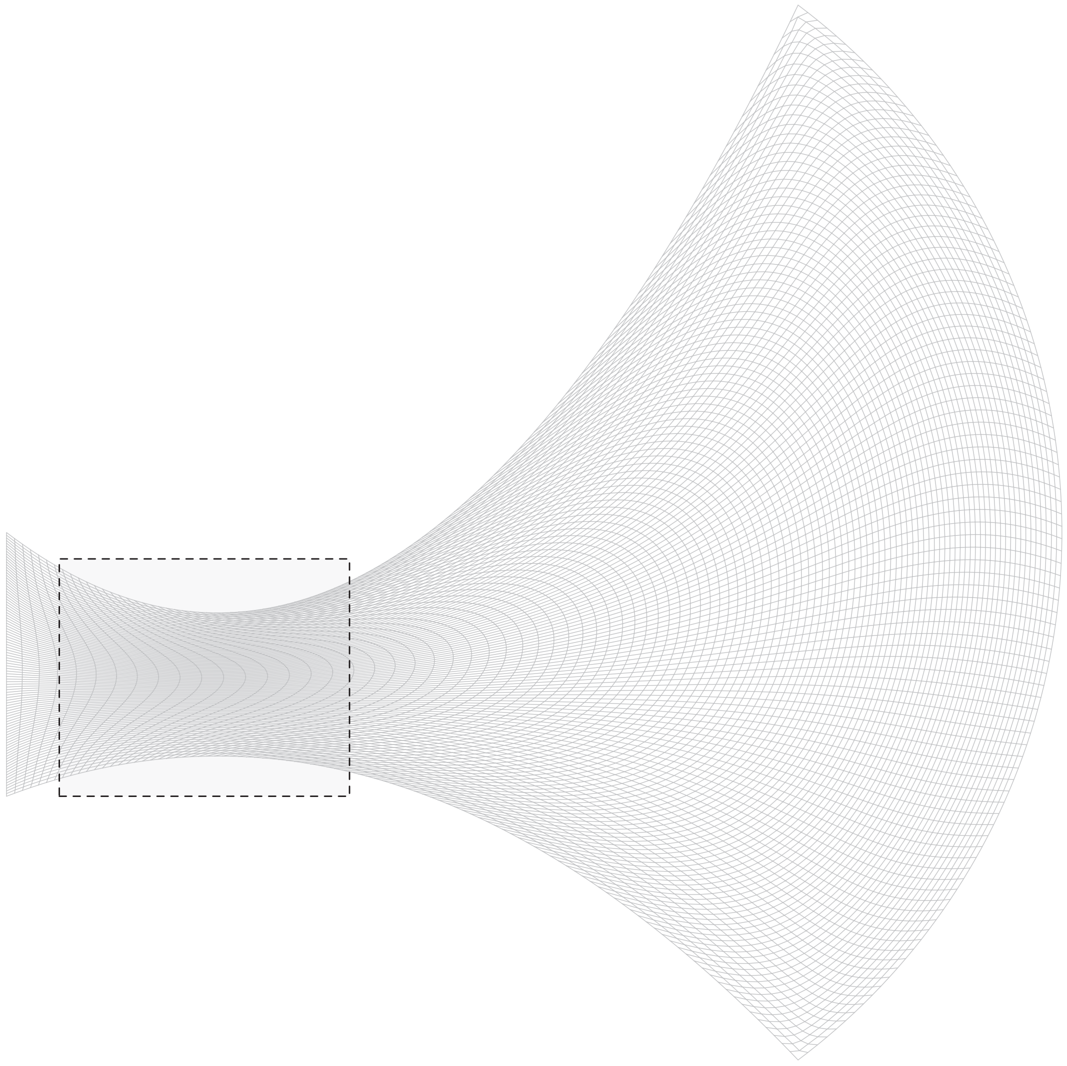

Figure 13.12. *Solution to the large horn-shaped 2D region mesh generation problem.*

13.5 ▪ Review and summary

In this chapter, we tackled practical problems using the Augmented Lagrangian software Algencan. Models were fully described, were the main code and the user-provided sub-

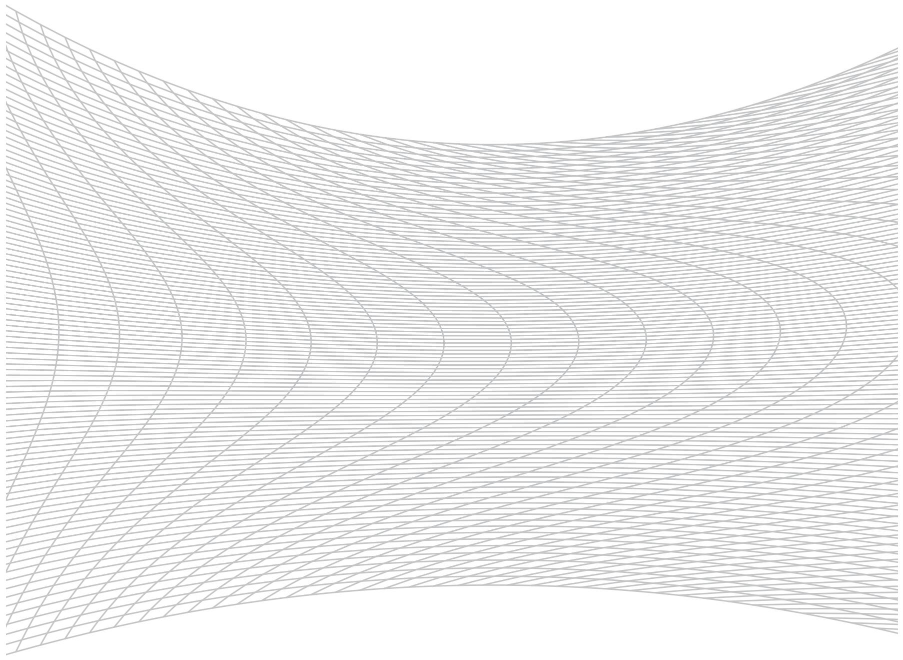

Figure 13.13. *Zoom of the shaded part of Figure 13.12.*

routines needed to solve each problem. Users were oriented toward clever choice of algorithmic parameters. Problems were used as examples, while the main focus was in the use of Algencan.

13.6 ▪ Further reading

The practical application of computing initial points for molecular dynamics simulations is fully described in [186, 192]. The Packmol package implements these ideas and is available at `http://www.ime.unicamp.br/~martinez/packmol/`.

The "sparse" implementation of the nonoverlapping objective function is described in detail in [65]. The problem of map projections appears to be a very controversial subject. In this chapter, we tackled the problem of drawing proportional maps from the optimization point of view, ignoring every possible subject that concerns cartographers, with the single purpose of illustrating the use of Algencan. A complete source of information on many approaches to this subject appears to be the book [240]. The main references to the control and mesh generation problems were given in the text.

13.7 ▪ Problems

13.1 The possibilities for problems in this chapter are enormous. Check the available codes related to the examples described in the present chapter. Reproduce the presented results. Modify the Algencan parameters, rerun, and compare the results.

13.2 Create your own examples. For each problem, tune the Algencan parameters to obtain something that you may consider a good approximation to a solution with a "satisfactory" performance of Algencan.

13.3 The unconstrained collection of Moré, Garbow, and Hillstrom (MGH) [204] describes 35 test functions $f : \mathbb{R}^n \to \mathbb{R}^m$. Use Algencan to solve the 35 feasibility