Résolution de problèmes d'optimisation non linéaire: IpOpt (Interior Point OPTimizer)

T. Haberkorn[1]

[1]MAPMO
Université d'Orléans

12ème journée CASCIMODOT, 1er Juillet 2010

Forme générale

- *IpOpt* sert à trouver un minimum (local) de problèmes d'optimisation de la forme:

$$\begin{cases} \min\limits_{x \in \mathbf{R}^n} f(x) \\ tq. \quad g_L \leq g(x) \leq g_U \\ \qquad x_L \leq x \leq x_U \end{cases}$$

avec $f(.)$ et $g(.)$ des fonctions $\mathcal{C}^2$,

- f est la fonction objectif,
- g correspond aux contraintes (linéaires et non linéaires) d'inégalités et d'égalites (si $g_l = g_U$),
- x_l et x_U sont des contraintes de boîtes,
- fait pour des problèmes de grande dimension

Un exemple élémentaire

Le problème

$$\begin{cases} \min\limits_{x\in\mathbb{R}^2} f(x) = -(x_2-2)^2 \\ tq \\ x_1^2 + x_2 - 1 = 0 \\ -1 \leqslant x_1 \leqslant 1 \end{cases}$$

Différentes implémentations

- En *C++* ou *fortran*, on définit le problème d'optimisation en donnant sa taille, ses contraintes de boîtes et diverses fonctions/procedures (objectif, contraintes, gradients, Hessien du Lagrangien).
- Définition sous *Ampl* (un langage de modélisation) suivant une sémantique spécifique.

Un exemple élémentaire

Le problème

$$\begin{cases} \min\limits_{x\in\mathbb{R}^2} f(x) = -(x_2-2)^2 \\ tq \\ x_1^2 + x_2 - 1 = 0 \\ -1 \leqslant x_1 \leqslant 1 \end{cases}$$

Différentes implémentations

- En *C++* ou *fortran*, on définit le problème d'optimisation en donnant sa taille, ses contraintes de boîtes et diverses fonctions/procédures (objectif, contraintes, gradients, Hessien du Lagrangien).
- Définition sous *Ampl* (un langage de modélisation) suivant une sémantique spécifique.

Un exemple élémentaire

Le problème

$$\begin{cases} \min\limits_{x\in\mathbb{R}^2} f(x) = -(x_2-2)^2 \\ tq \\ x_1^2 + x_2 - 1 = 0 \\ -1 \leqslant x_1 \leqslant 1 \end{cases}$$

Différentes implémentations

- En *C++* ou *fortran*, on définit le problème d'optimisation en donnant sa taille, ses contraintes de boîtes et diverses fonctions/procédures (objectif, contraintes, gradients, Hessien du Lagrangien).
- Définition sous *Ampl* (un langage de modélisation) suivant une sémantique spécifique.

Utilisation avec *C++* (1)

- Tout se trouve dans une classe *MyNLP*.

Tailles du problème

```
bool MyNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                Index& nnz_h_lag, IndexStyleEnum& index_style){
n = 2; // nombre de variables
m = 1; // nombre de contraintes
nnz_jac_g = 2; // nombre d'el. non nuls du Jacobien des contraintes
```

Bornes sur les contraintes et les boîtes

```
bool MyNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                            Index m, Number* g_l, Number* g_u){
x_l[0] = -1.0; x_u[0] = 1.0; // boite sur x1
x_l[1] = -1.0e19; x_u[1] = +1.0e19; // rien sur x2
g_l[0] = g_u[0] = 0.0; // la seule contrainte est = 0
```

Initialisation

```
bool MyNLP::get_starting_point(Index n, bool init_x, Number* x,
                bool init_z, Number* z_L, Number* z_U, Index m,
                bool init_lambda, Number* lambda){
x[0] = 0.5; x[1] = 1.5; // init des 2 parametres
// possibilite d'init des multiplicateurs et autres var. de l'algo
```

Utilisation avec *C++* (1)

- Tout se trouve dans une classe *MyNLP*.

Tailles du problème

```
bool MyNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                Index& nnz_h_lag, IndexStyleEnum& index_style){
n = 2; // nombre de variables
m = 1; // nombre de contraintes
nnz_jac_g = 2; // nombre d'el. non nuls du Jacobien des contraintes
```

Bornes sur les contraintes et les boîtes

```
bool MyNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                            Index m, Number* g_l, Number* g_u){
x_l[0] = -1.0; x_u[0] = 1.0; // boite sur x1
x_l[1] = -1.0e19; x_u[1] = +1.0e19; // rien sur x2
g_l[0] = g_u[0] = 0.0; // la seule contrainte est = 0
```

Initialisation

```
bool MyNLP::get_starting_point(Index n, bool init_x, Number* x,
                bool init_z, Number* z_L, Number* z_U, Index m,
                bool init_lambda, Number* lambda){
x[0] = 0.5; x[1] = 1.5; // init des 2 parametres
// possibilite d'init des multiplicateurs et autres var. de l'algo
```

Utilisation avec *C++* (1)

- Tout se trouve dans une classe *MyNLP*.

Tailles du problème

```
bool MyNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                Index& nnz_h_lag, IndexStyleEnum& index_style){
n = 2; // nombre de variables
m = 1; // nombre de contraintes
nnz_jac_g = 2; // nombre d'el. non nuls du Jacobien des contraintes
```

Bornes sur les contraintes et les boîtes

```
bool MyNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                            Index m, Number* g_l, Number* g_u){
x_l[0] = -1.0; x_u[0] = 1.0; // boite sur x1
x_l[1] = -1.0e19; x_u[1] = +1.0e19; // rien sur x2
g_l[0] = g_u[0] = 0.0; // la seule contrainte est = 0
```

Initialisation

```
bool MyNLP::get_starting_point(Index n, bool init_x, Number* x,
                bool init_z, Number* z_L, Number* z_U, Index m,
                bool init_lambda, Number* lambda){
x[0] = 0.5; x[1] = 1.5; // init des 2 parametres
// possibilite d'init des multiplicateurs et autres var. de l'algo
```

Utilisation avec *C++* (1)

- Tout se trouve dans une classe *MyNLP*.

Tailles du problème

```
bool MyNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                Index& nnz_h_lag, IndexStyleEnum& index_style){
n = 2; // nombre de variables
m = 1; // nombre de contraintes
nnz_jac_g = 2; // nombre d'el. non nuls du Jacobien des contraintes
```

Bornes sur les contraintes et les boîtes

```
bool MyNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                            Index m, Number* g_l, Number* g_u){
x_l[0] = -1.0; x_u[0] = 1.0; // boite sur x1
x_l[1] = -1.0e19; x_u[1] = +1.0e19; // rien sur x2
g_l[0] = g_u[0] = 0.0; // la seule contrainte est = 0
```

Initialisation

```
bool MyNLP::get_starting_point(Index n, bool init_x, Number* x,
                bool init_z, Number* z_L, Number* z_U, Index m,
                bool init_lambda, Number* lambda){
x[0] = 0.5; x[1] = 1.5; // init des 2 parametres
// possibilite d'init des multiplicateurs et autres var. de l'algo
```

Utilisation avec *C++* (2)

Objectif

```
bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{ obj_value = -(x[1] - 2.0) * (x[1] - 2.0); ...}
```

Gradient objectif

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{ grad_f[0] = 0.0; grad_f[1] = -2.0*(x[1] - 2.0);}
```

Contrainte

```
bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m,  Number* g)
{ g[0] = -(x[0]*x[0] + x[1] - 1.0);...}
```

Gradient des contraintes

```
bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x, Index m,
                       Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
if (values == NULL) { // complete iRow[] et jCol[] pour creux}
else {values[0] = -2.0 * x[0]; values[1] = -1.0;} ...}
```

Idem pour le Hessien *bool MyNLP::eval_h(...)*

Post-traitement de la solution

```
void MyNLP::finalize_solution(...){
// par exemple, ecriture dans un fichier ou affichage ...}
```

Utilisation avec *C++* (2)

Objectif

```
bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{ obj_value = -(x[1] - 2.0) * (x[1] - 2.0); ...}
```

Gradient objectif

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{ grad_f[0] = 0.0; grad_f[1] = -2.0*(x[1] - 2.0);}
```

Contrainte

```
bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m,  Number* g)
{ g[0] = -(x[0]*x[0] + x[1] - 1.0);...}
```

Gradient des contraintes

```
bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x, Index m,
                       Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
if (values == NULL) { // complete iRow[] et jCol[] pour creux}
else {values[0] = -2.0 * x[0]; values[1] = -1.0;} ...}
```

Idem pour le Hessien *bool MyNLP::eval_h(...)*

Post-traitement de la solution

```
void MyNLP::finalize_solution(...){
// par exemple, ecriture dans un fichier ou affichage ...}
```

Utilisation avec *C++* (2)

Objectif

```
bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{ obj_value = -(x[1] - 2.0) * (x[1] - 2.0); ...}
```

Gradient objectif

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{ grad_f[0] = 0.0; grad_f[1] = -2.0*(x[1] - 2.0);}
```

Contrainte

```
bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m,  Number* g)
{ g[0] = -(x[0]*x[0] + x[1] - 1.0);...}
```

Gradient des contraintes

```
bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x, Index m,
                       Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
if (values == NULL) { // complete iRow[] et jCol[] pour creux}
else {values[0] = -2.0 * x[0]; values[1] = -1.0;} ...}
```

Idem pour le Hessien *bool MyNLP::eval_h(...)*

Post-traitement de la solution

```
void MyNLP::finalize_solution(...){
// par exemple, ecriture dans un fichier ou affichage ...}
```

Utilisation avec *C++* (2)

Objectif

```
bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{ obj_value = -(x[1] - 2.0) * (x[1] - 2.0); ...}
```

Gradient objectif

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{ grad_f[0] = 0.0; grad_f[1] = -2.0*(x[1] - 2.0);}
```

Contrainte

```
bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m,  Number* g)
{ g[0] = -(x[0]*x[0] + x[1] - 1.0);...}
```

Gradient des contraintes

```
bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x, Index m,
                       Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
if (values == NULL) { // complete iRow[] et jCol[] pour creux}
else {values[0] = -2.0 * x[0]; values[1] = -1.0;} ...}
```

Idem pour le Hessien *bool MyNLP::eval_h(...)*

Post-traitement de la solution

```
void MyNLP::finalize_solution(...){
// par exemple, ecriture dans un fichier ou affichage ...}
```

Utilisation avec *C++* (2)

Objectif

```
bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{ obj_value = -(x[1] - 2.0) * (x[1] - 2.0); ...}
```

Gradient objectif

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{ grad_f[0] = 0.0; grad_f[1] = -2.0*(x[1] - 2.0);}
```

Contrainte

```
bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m,  Number* g)
{ g[0] = -(x[0]*x[0] + x[1] - 1.0);...}
```

Gradient des contraintes

```
bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x, Index m,
                       Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
if (values == NULL) { // complete iRow[] et jCol[] pour creux}
else {values[0] = -2.0 * x[0]; values[1] = -1.0;} ...}
```

Idem pour le Hessien *bool MyNLP::eval_h(...)*

Post-traitement de la solution

```
void MyNLP::finalize_solution(...){
// par exemple, ecriture dans un fichier ou affichage ...}
```

Utilisation avec *Ampl*

Définition 'naturelle' du problème

```
# declaration variable + boite + init
var x1 >= 1, <= -1, default 0.5;
var x2 default 1.5;

# objectif
minimize f: -(x2-2)^2;

# contrainte(s)
subject to g: x1^2 + x2 - 1 = 0;

# eventuellement initialisation particuliere (lecture fichier, calculs...)

# instruction d'appel du solveur + quelques options
option  solver ipopt;
option ipopt_options "imaxiter=1000 dtol=1.e-12 iscale=2 dfillinfact=2";

# appel du solveur
solve;

# post traitement
printf: " x1 = %24.16e\n", x1 > out.dat; # dans un fichier
printf: " x2 = %24.16e\n", x2; # a l'ecran
```

- Puis appel depuis l'exécutable *Ampl*.

Utilisation avec *Ampl*

Définition 'naturelle' du problème

```
# declaration variable + boite + init
var x1 >= 1, <= -1, default 0.5;
var x2 default 1.5;

# objectif
minimize f: -(x2-2)^2;

# contrainte(s)
subject to g: x1^2 + x2 - 1 = 0;

# eventuellement initialisation particuliere (lecture fichier, calculs...)

# instruction d'appel du solveur + quelques options
option  solver ipopt;
option ipopt_options "imaxiter=1000 dtol=1.e-12 iscale=2 dfillinfact=2";

# appel du solveur
solve;

# post traitement
printf: " x1 = %24.16e\n", x1 > out.dat; # dans un fichier
printf: " x2 = %24.16e\n", x2; # a l'ecran
```

- Puis appel depuis l'exécutable *Ampl*.

Algorithme : Réécriture et barrière

- Les inégalités $g_i^L \leq g_i(x) \leq g_i^U$ sont remplacées par:

$$g_i(x) - s_i = 0, \quad g_i^L \leq s_i \leq g_i^U,$$

avec s_i une nouvelle variable (variable d'écart).

- On réécrit le problème d'optimisation sous forme de contraintes d'égalités:

$$\begin{cases} \min\limits_{x \in \mathbf{R}^n} f(x) \\ tq. \quad c(x) = 0 \\ \quad\quad x_L \leq x \leq x_U \end{cases}$$

où x est augmentée des variables d'écarts.

- Puis on introduit les barrières logarithmiques:

$$\begin{cases} \min\limits_{x \in \mathbf{R}^n} \varphi_\mu(x) = f(x) - \mu \sum_{i=1}^n ln(x_i - x_i^L) - \mu \sum_{i=1}^n ln(x_i^U - x_i) \\ tq. \quad c(x) = 0 \end{cases}$$

Algorithme : Réécriture et barrière

- Les inégalités $g_i^L \leq g_i(x) \leq g_i^U$ sont remplacées par:

$$g_i(x) - s_i = 0, \quad g_i^L \leq s_i \leq g_i^U,$$

avec s_i une nouvelle variable (variable d'écart).

- On réécrit le problème d'optimisation sous forme de contraintes d'égalités:

$$\begin{cases} \min\limits_{x \in \mathbb{R}^n} & f(x) \\ tq. & c(x) = 0 \\ & x_L \leq x \leq x_U \end{cases}$$

où x est augmentée des variables d'écarts.

- Puis on introduit les barrières logarithmiques:

$$\begin{cases} \min\limits_{x \in \mathbb{R}^n} & \varphi_\mu(x) = f(x) - \mu \sum_{i=1}^n \ln(x_i - x_L) - \mu \sum_{i=1}^n \ln(x_U - x_i) \\ tq. & c(x) = 0 \end{cases}$$

Algorithme : Réécriture et barrière

- Les inégalités $g_i^L \leq g_i(x) \leq g_i^U$ sont remplacées par:

$$g_i(x) - s_i = 0, \quad g_i^L \leq s_i \leq g_i^U,$$

avec s_i une nouvelle variable (variable d'écart).

- On réécrit le problème d'optimisation sous forme de contraintes d'égalités:

$$\begin{cases} \min\limits_{x \in \mathbb{R}^n} f(x) \\ tq. \quad c(x) = 0 \\ \qquad x_L \leq x \leq x_U \end{cases}$$

où x est augmentée des variables d'écarts.

- Puis on introduit les barrières logarithmiques:

$$\begin{cases} \min\limits_{x \in \mathbb{R}^n} \varphi_\mu(x) = f(x) - \mu \sum_{i=1}^n ln(x_i - x_L) - \mu \sum_{i=1}^n ln(x_U - x_i) \\ tq.\ c(x) = 0 \end{cases}$$

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Algorithme : Pas primal-dual

Une fois la réécriture effectuée on procède en gros comme suit:

1. On se choisit un paramètre de pénalisation μ.
2. On applique les conditions nécessaires du premier ordre au problème réécrit $\Rightarrow$ se traduisent par un système d'équations non linéaires à résoudre.
3. Une méthode de type Newton est utilisée, dans laquelle on va calculer une direction pour améliorer la variable x et une pour les multiplicateurs de Lagrange λ.
4. On calcule ensuite indépendamment un pas sur x et un sur λ.
5. On ré-itère Newton jusquà avoir une solution satisfaisant les conditions du premier ordre avec une précision donnée.
6. On diminue μ et on recommence.

Comment se procurer IpOpt

Lien

le code source d'*IpOpt* est disponible à sur la page du projet *COIN-OR*:
https://projects.coin-or.org/Ipopt

Installation

1. Télécharger le code source d'*IpOpt*,
2. Télécharger les composantes tierces (scripts dispos pour ftp anonyme),
3. Compiler le tout avec votre compilateur C++ et fortran favori,
4. C'est fini.

NEOS

Serveur d'optimisation regroupant un grand nombre de logiciels d'optimisation:
http://www-neos.mcs.anl.gov/ $\Rightarrow$ exécuter codes Ampl/Gamms sur machine distante avec divers logiciels

A. Wächter and L. T. Biegler, *On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming 106(1), pp. 25-57, 2006.

Comment se procurer IpOpt

Lien

le code source d'*IpOpt* est disponible à sur la page du projet *COIN-OR*:
https://projects.coin-or.org/Ipopt

Installation

1. Télécharger le code source d'*IpOpt*,
2. Télécharger les composantes tierces (scripts dispos pour ftp anonyme),
3. Compiler le tout avec votre compilateur C++ et fortran favori,
4. C'est fini.

NEOS

Serveur d'optimisation regroupant un grand nombre de logiciels d'optimisation:
http://www-neos.mcs.anl.gov/ $\Rightarrow$ exécuter codes Ampl/Gamms sur machine distante avec divers logiciels

A. Wächter and L. T. Biegler, *On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming 106(1), pp. 25-57, 2006.

Comment se procurer IpOpt

Lien

le code source d'*IpOpt* est disponible à sur la page du projet *COIN-OR*:
https://projects.coin-or.org/Ipopt

Installation

1. Télécharger le code source d'*IpOpt*,
2. Télécharger les composantes tierces (scripts dispos pour ftp anonyme),
3. Compiler le tout avec votre compilateur C++ et fortran favori,
4. C'est fini.

NEOS

Serveur d'optimisation regroupant un grand nombre de logiciels d'optimisation:
http://www-neos.mcs.anl.gov/ $\Rightarrow$ exécuter codes Ampl/Gamms sur machine distante avec divers logiciels

A. Wächter and L. T. Biegler, *On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming 106(1), pp. 25-57, 2006.

Comment se procurer IpOpt

Lien

le code source d'*IpOpt* est disponible à sur la page du projet *COIN-OR*:
https://projects.coin-or.org/Ipopt

Installation

1. Télécharger le code source d'*IpOpt*,
2. Télécharger les composantes tierces (scripts dispos pour ftp anonyme),
3. Compiler le tout avec votre compilateur C++ et fortran favori,
4. C'est fini.

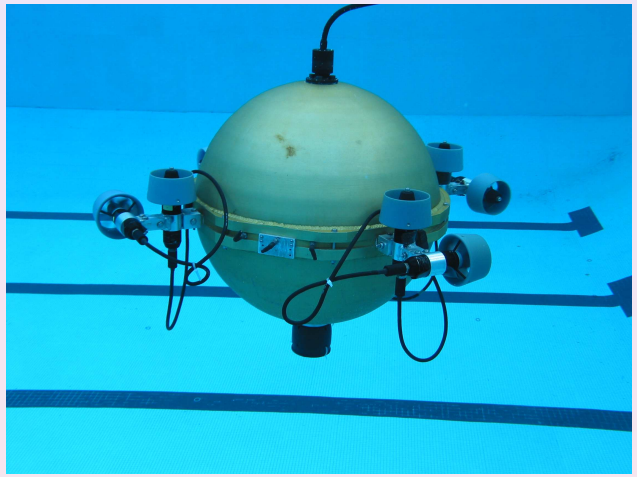
NEOS

Serveur d'optimisation regroupant un grand nombre de logiciels d'optimisation:
http://www-neos.mcs.anl.gov/ $\Rightarrow$ exécuter codes Ampl/Gamms sur machine distante avec divers logiciels

A. Wächter and L. T. Biegler, *On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming 106(1), pp. 25-57, 2006.

Sous-marin autonome

- Opéré par le *Autonomous Systems Lab* à l'Université d'Hawaï
- corps sphérique de 65 cm de diamètre
- 8 moteurs (4 hor./4 vert.)
- Masse $\approx$ 127 kg (0.3 kg Archimède)
- 6 degrés de liberté $\Rightarrow$ très manœuvrable



Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{aligned} M\dot{\nu} &= M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} &= J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ &+ D_\Omega(\Omega) + \tau_\Omega \end{aligned}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{aligned} M\dot{\nu} &= M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} &= J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ &+ D_\Omega(\Omega) + \tau_\Omega \end{aligned}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{aligned} M\dot{\nu} &= M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} &= J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ &+ D_\Omega(\Omega) + \tau_\Omega \end{aligned}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{array}{rcl} M\dot{\nu} & = & M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} & = & J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ & + & D_\Omega(\Omega) + \tau_\Omega \end{array}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{array}{rcl} M\dot{\nu} & = & M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} & = & J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ & + & D_\Omega(\Omega) + \tau_\Omega \end{array}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Modélisation

- Position et orientation $\eta = (x, y, z, \phi, \theta, \psi)^t$ dans repère inertiel.
- Repère mobile attaché au centre de gravité du corps rigide.
- Vitesses translationelles $\nu = (u, v, w)^t$ et angulaires $\Omega = (p, q, r)^t$ dans le repère mobile.
- Dynamique des vitesses:

$$\dot{\eta} = \begin{pmatrix} R & 0 \\ 0 & \Theta_1 \end{pmatrix} \begin{pmatrix} \nu \\ \Omega \end{pmatrix}$$

- Dynamique des accélérations:

$$\begin{array}{rcl} M\dot{\nu} & = & M\nu \times \Omega + D_\nu(\nu) + R^t B + \varphi_\nu \\ J\dot{\Omega} & = & J\Omega \times \Omega + M\nu \times \nu - r_{CB} \times R^t B \\ & + & D_\Omega(\Omega) + \tau_\Omega \end{array}$$

- Contrôles φ_ν et τ_Ω résultantes de l'action des 8 moteurs.

Critère

Problème de contrôle optimal

- Les critères intéressant sont typiquement la minimisation du temps de transfert ou la minimisation de la consommation pour aller d'un état à un autre.
- On obtient un problème de contrôle optimale de la forme:

$$(OCP)\begin{cases} \min\limits_{\varphi_\nu,\tau_\Omega} t_f \ (ou\ consommation) \\ tq. \quad \dot{\chi}(t) = f(\chi,\varphi_\nu,\tau_\Omega),\ \forall t \in [0,t_f] \\ \qquad \chi(0) = \chi_0 \\ \qquad \chi(t_f) = \chi_f \\ \qquad (\varphi_\nu(t),\tau_\Omega) \in TCM \cdot (\prod_{i=1}^{8}[\alpha_i,\beta_i]) \in \mathbf{R}^6 \end{cases}$$

avec $\chi = (\eta_1, \eta_2, \nu, \Omega) \in \mathbf{R}^{12}$ l'état.

Critère

Problème de contrôle optimal

- Les critères intéressant sont typiquement la minimisation du temps de transfert ou la minimisation de la consommation pour aller d'un état à un autre.
- On obtient un problème de contrôle optimale de la forme:

$$(OCP)\begin{cases} \underset{\varphi_\nu,\tau_\Omega}{\min}\ t_f\ (\textit{ou consommation}) \\ tq. \quad \begin{array}{l} \dot{\chi}(t) = f(\chi,\varphi_\nu,\tau_\Omega),\ \forall t\in[0,t_f] \\ \chi(0)=\chi_0 \\ \chi(t_f)=\chi_f \\ (\varphi_\nu(t),\tau_\Omega)\in TCM\cdot(\prod_{i=1}^{8}[\alpha_i,\beta_i])\in\mathbb{R}^6 \end{array} \end{cases}$$

avec $\chi=(\eta_1,\eta_2,\nu,\Omega)\in\mathbb{R}^{12}$ l'état.

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

- Avantages : taille du problème d'optimisation réduite ($6 * N$ au lieu de $18 * N$) et uniquement 12 contraintes qui correspondent à $\chi(t_f) = \chi_f$.
- Inconvénients : les 12 équations différentielles à intégrer sont souvent raides et doivent être résolues précisément.
- On utilise cette approche avec variables adjointes qui nous donnera les conditions nécessaires d'optimalité.

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

- Avantage : taille du problème d'optimisation réduite ($6 * N$ au lieu de $18 * N$ et uniquement 12 contraintes qui correspondent à $\chi(t_f) = \chi_f$).
- Inconvénient : les 12 contraintes sont extrémement non linéaire, l'optimisation devient beaucoup moins robuste.
- On utilise cette approche par exemple si on veut restreindre le nombre de commutations des contrôles (contrainte opérationelle du véhicule).

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

- Avantage : taille du problème d'optimisation réduite ($6 * N$ au lieu de $18 * N$ et uniquement 12 contraintes qui correspondent à $\chi(t_f) = \chi_f$).
- Inconvénient : les 12 contraintes sont extrémement non linéaire, l'optimisation devient beaucoup moins robuste.
- On utilise cette approche par exemple si on veut restreindre le nombre de commutations des contrôles (contrainte opérationelle du véhicule).

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

- Avantage : taille du problème d'optimisation réduite ($6 * N$ au lieu de $18 * N$ et uniquement 12 contraintes qui correspondent à $\chi(t_f) = \chi_f$).
- Inconvénient : les 12 contraintes sont extrémement non linéaire, l'optimisation devient beaucoup moins robuste.
- On utilise cette approche par exemple si on veut restreindre le nombre de commutations des contrôles (contrainte opérationelle du véhicule).

Discrétisations

(*OCP*) n'est pas un problème d'optimisation, mais on peut l'approximer comme tel.

Première discrétisation

- On discrétise létat $\chi(.)$ et les contrôles φ_ν et τ_Ω:

$$\Rightarrow \chi_i,\ \varphi_{\nu,i},\ \tau_{\Omega,i},\ i = 1, \cdots, N$$

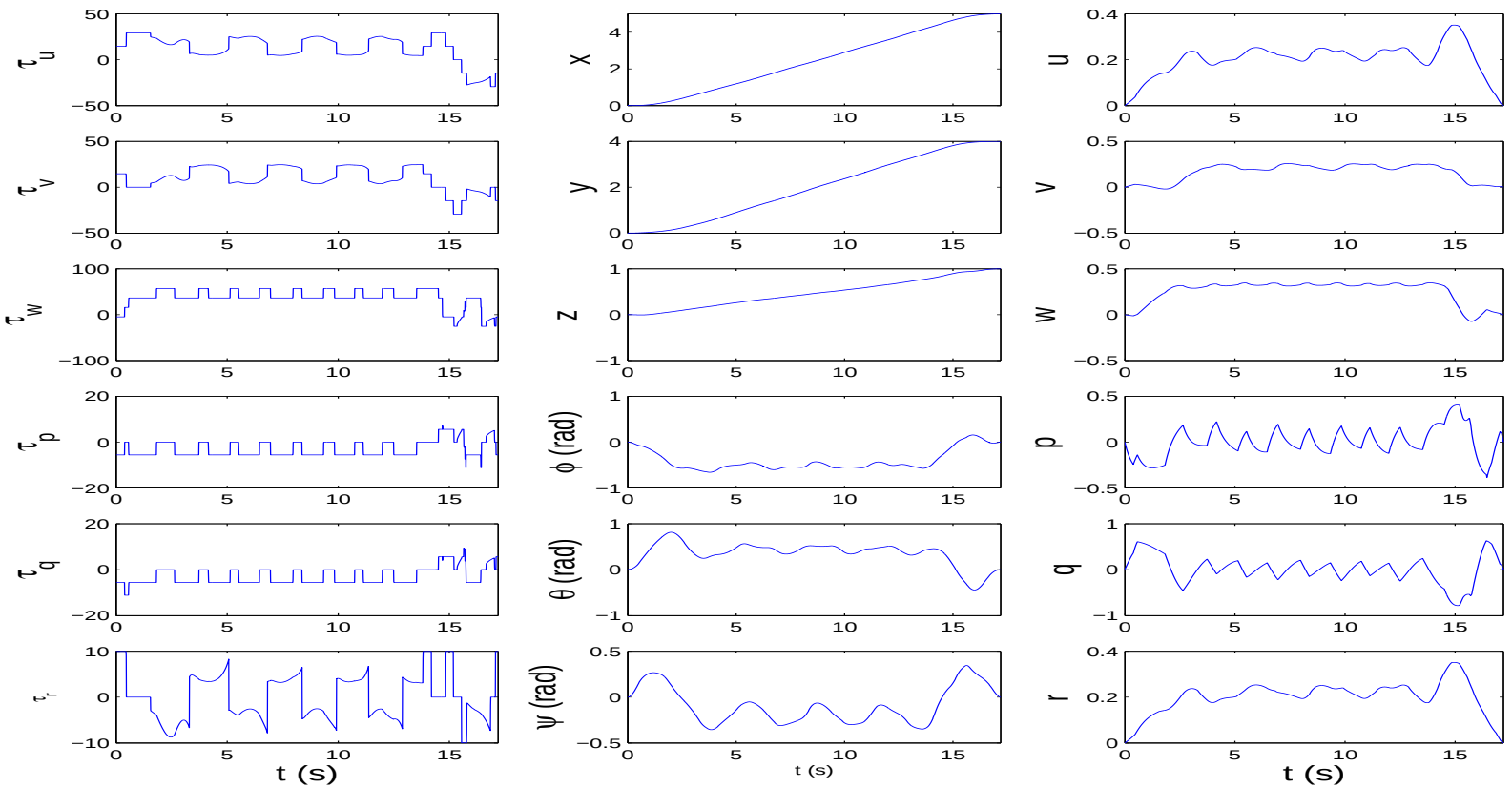
  $(12 + 6) * N$ inconnues
- On réécrit la dynamique comme des contraintes reliant un point de la discrétisation au suivant: par exemple avec un schéma tout simple d'Euler.
- Les contraintes intiales et finales sont sous la bonne forme

Discrétisation en contrôle

Ne discrétiser que les contrôles φ_ν et τ_Ω et intégrer la dynamique à partir de χ_0.

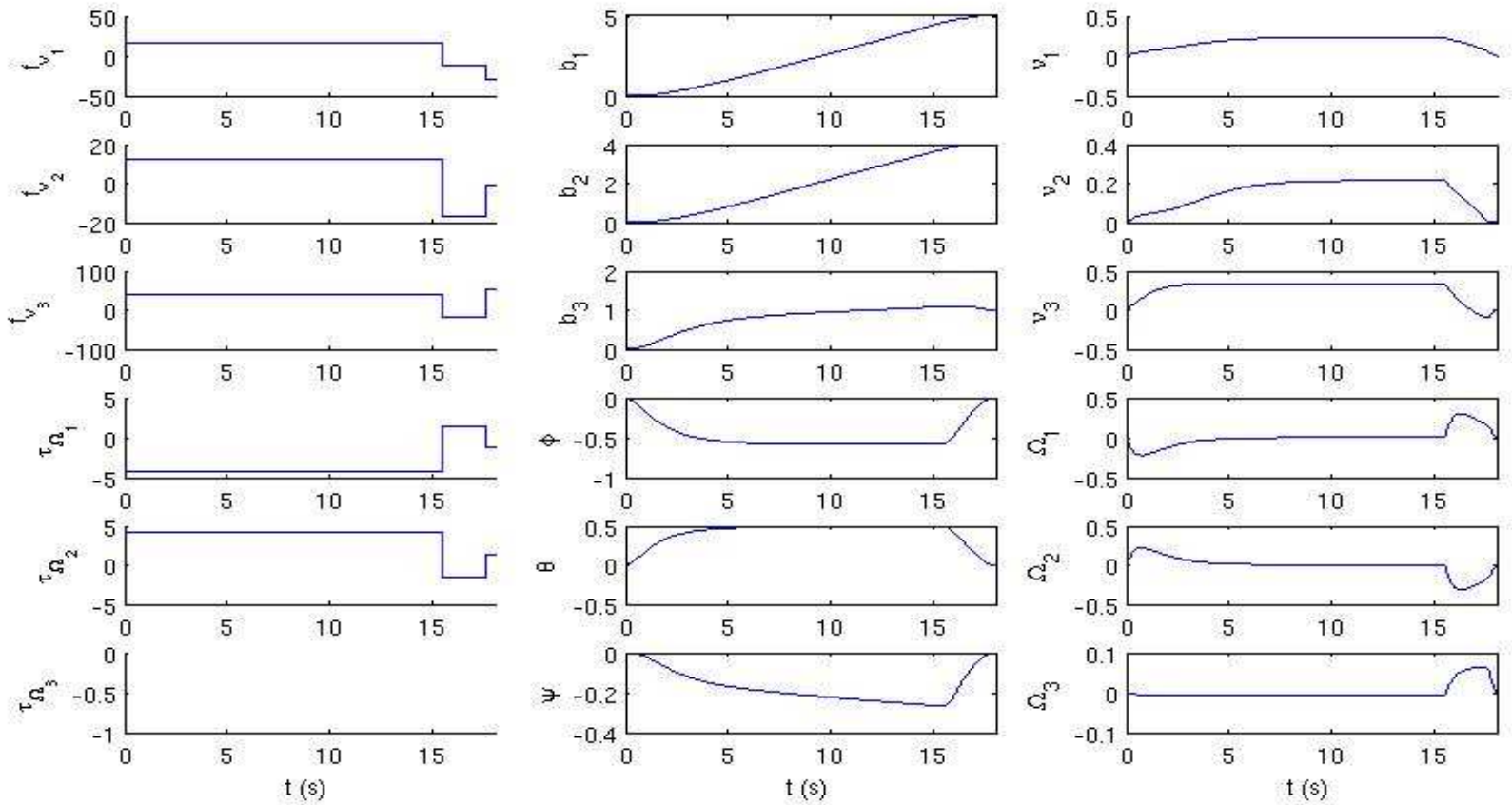
- Avantage : taille du problème d'optimisation réduite ($6 * N$ au lieu de $18 * N$ et uniquement 12 contraintes qui correspondent à $\chi(t_f) = \chi_f$).
- Inconvénient : les 12 contraintes sont extrémement non linéaire, l'optimisation devient beaucoup moins robuste.
- On utilise cette approche par exemple si on veut restreindre le nombre de commutations des contrôles (contrainte opérationelle du véhicule).

Solution en temps minimum, $\chi_f = (5, 4, 1, 0, \cdots, 0)$



- Beaucoup de commutations.
- Une dizaine de minutes avec *IpOpt*.

Limitation du nombre de commutations



- 2 temps de commutation mais $t_f \approx t_f^{\min}$
- Quelques secondes avec *IpOpt*.