MTH8408 : Méthodes d'optimisation et contrôle optimal

Laboratoire 4: Optimisation sans contraintes et méthodes itératives

Tangi Migot et Paul Raynaud

In [1]:
```
using LinearAlgebra, Krylov, NLPModels, Printf, Logging, SolverCore, Test, ADNLPMod
```

```
[ Info: Precompiling ADNLPModels [54578032-b7ea-4c30-94aa-7cbd1cce6c9a]
```

In [9]:
```
#Test problem:
FH(x) = [x[2]+x[1].^2-11, x[1]+x[2].^2-7]
x0H = [10., 20.]
###########################
#Utilise FH et x0H pour créer un ADNLSModel
himmelblau_nls = ADNLPModels.ADNLSModel(FH, x0H, 2)
###########################
```

Out[9]:
```
ADNLSModel - Nonlinear least-squares model with automatic differentiation backend
ADModelBackend{
  ForwardDiffADGradient,
  ForwardDiffADHvprod,
  EmptyADbackend,
  EmptyADbackend,
  EmptyADbackend,
  ForwardDiffADHessian,
  EmptyADbackend,
}
  Problem name: Generic
   All variables: ████████████████████ 2      All constraints: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
All residuals: ████████████████████ 2
            free: ████████████████████ 2                 free: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                lower: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0            non
linear: ████████████████████ 2
           upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                upper: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
nnzj: (  0.00% sparsity)   4
         low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0              low/upp: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
nnzh: (  0.00% sparsity)   3
           fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                fixed: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
          infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               infeas: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
            nnzh: (  0.00% sparsity)   3               linear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                    nonlinear: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
                                                         nnzj: (------% sparsity)

  Counters:
             obj: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 grad: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
cons: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
        cons_lin: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0             cons_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
jcon: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           jgrad: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                  jac: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0              j
ac_lin: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
         jac_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                jprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0            jpr
od_lin: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
       jprod_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               jtprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0           jtpr
od_lin: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
      jtprod_nln: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0                 hess: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
hprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
           jhess: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0               jhprod: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0             re
sidual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
    jac_residual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0       jprod_residual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0      jtprod_re
sidual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
   hess_residual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0       jhess_residual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0       hprod_re
sidual: ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅ 0
```

Exercice 1: Gauss-Newton

Dans cet exercice, on complète une implémentation de la méthode Gauss-Newton avec région de confiance (paramétrée par Δ) discutée en cours.

Il faut compléter les morceaux:

- utiliser les fonctions des NLSModels pour obtenir F et sa jacobienne (ici on utilise pas la jacobienne mais juste le produit jacobienne-vecteur). Parcourez la documentation de NLPModels pour déterminer la fonction adéquat, indice les fonctions pour les NLSModels indiquent des `nls` au lieu de `nlp` dans la documentation.
- Utiliser la fonction `lsmr` du package `Krylov.jl` pour résoudre le système linéaire avec une contrainte de `radius`. Lisez la documentation de `lsmr`.

In [10]:
```
function gauss_newton(nlp      :: AbstractNLSModel,
                      x        :: AbstractVector,
                      ϵ        :: AbstractFloat;
                      η₁       :: AbstractFloat = 1e-3,
                      η₂       :: AbstractFloat = 0.66,
                      σ₁       :: AbstractFloat = 0.25,
                      σ₂       :: AbstractFloat = 2.0,
                      max_eval :: Int = 1_000,
                      max_time :: AbstractFloat = 60.,
                      max_iter :: Int = typemax(Int64)
                      )
    ######################################################
    Fx = NLPModels.residual(himmelblau_nls, x)
    Jx = NLPModels.jac_residual(himmelblau_nls, x)
    ######################################################
    normFx = norm(Fx)

    Δ = 1.

    iter = 0

    el_time = 0.0
    tired   = neval_residual(nlp) > max_eval || el_time > max_time
    status  = :unknown

    start_time = time()
    too_small  = false
    normdual   = norm(Jx' * Fx)
    optimal    = min(normFx, normdual) ≤ ϵ

    @info log_header([:iter, :nf, :primal, :status, :nd, :Δ],
    [Int, Int, Float64, String, Float64, Float64],
    hdr_override=Dict(:nf => "#F", :primal => "‖F(x)‖", :nd => "‖d‖"))

    while !(optimal || tired || too_small)

        #################################
        #Compute a direction satisfying the trust-region constraint
        (d, stats)  = lsmr(-Jx, Fx, radius = Δ)
        #################################

        too_small = norm(d) < 1e-15
        if too_small #the direction is too small
            status = :too_small
        else
            xp      = x + d
            ###########################
```

```julia
            Fxp     = NLPModels.residual(himmelblau_nls, xp)
            ###########################
            normFxp = norm(Fxp)

            Pred = 0.5 * (normFx^2 - norm(Jx * d + Fx)^2)
            Ared = 0.5 * (normFx^2 - normFxp^2)

            if Ared/Pred < η₁
                Δ = max(1e-8, Δ * σ₁)
                status = :reduce_Δ
            else #success
                x  = xp
                ###########################
                Jx = NLPModels.jac_residual(himmelblau_nls, x)
                ###########################
                Fx = Fxp
                normFx = normFxp
                status = :success
                if Ared/Pred > η₂ && norm(d) >= 0.99 * Δ
                    Δ *= σ₂
                end
            end
        end

        @info log_row(Any[iter, neval_residual(nlp), normFx, status, norm(d), Δ])

        el_time      = time() - start_time
        iter   += 1

        many_evals   = neval_residual(nlp) > max_eval
        iter_limit   = iter > max_iter
        tired        = many_evals || el_time > max_time || iter_limit
        normdual     = norm(Jx' * Fx)
        optimal      = min(normFx, normdual) ≤ ϵ
    end

    status = if optimal
        :first_order
    elseif tired
        if neval_residual(nlp) > max_eval
            :max_eval
        elseif el_time > max_time
            :max_time
        elseif iter > max_iter
            :max_iter
        else
            :unknown_tired
        end
    elseif too_small
        :stalled
    else
        :unknown
    end

    return GenericExecutionStats(nlp; status, solution = x,
                                 objective = normFx^2 / 2,
```

```julia
                                    dual_feas = normdual,
                                    iter = iter,
                                    elapsed_time = el_time)
end
```

Out[10]: gauss_newton (generic function with 1 method)

In [11]:
```julia
stats = gauss_newton(himmelblau_nls, himmelblau_nls.meta.x0, 1e-6)
@test stats.status == :first_order
```

```
[ Info:   iter      #F     ‖F(x)‖          status      ‖d‖          Δ
[ Info:      0       2    3.8e+02         success    1.0e+00    2.0e+00
[ Info:      1       3    3.1e+02         success    2.0e+00    4.0e+00
[ Info:      2       4    1.9e+02         success    4.0e+00    8.0e+00
[ Info:      3       5    4.5e+01         success    7.7e+00    8.0e+00
[ Info:      4       6    9.5e+00         success    3.4e+00    8.0e+00
[ Info:      5       7    1.6e+00         success    1.3e+00    8.0e+00
[ Info:      6       8    1.2e-01         success    3.5e-01    8.0e+00
[ Info:      7       9    8.8e-04         success    3.0e-02    8.0e+00
[ Info:      8      10    5.3e-08         success    2.3e-04    8.0e+00
```

Out[11]: **Test Passed**

Exercice 2: Méthode Levenberg-Marquard inexacte

Dans cet exercice, on complète une implémentation de la méthode Levenberg-Marquardt. Pour compléter le code `lm_param` on va utiliser les fonctions suivantes:

- `dsol` qui calcul la solution du système $\min_x \frac{1}{2}\|J(x) d + F(x)\| + \lambda \|x\|^2$ avec la fonction `lsqr` du package `Krylov.jl`.
- `multi_sol` qui pour un entier nl donné et un μ va résoudre le problème de dsol pour nl valeurs de λ (autour de la valeur μ). Par exemple, pour $\mu=10^{-6}$ et $nl=3$, on prendra $\lambda=10^{-7}, 10^{-6},10^{-5}$. Parmis les `nl` directions calculées, on retourne celle qui donne la plus petite valeur de $\|F(x+d)\|^2$.

In [12]:
```julia
function dsol(Fx, Jx, λ)
    (d, stats) = lsqr(-Jx, Fx, λ)
    return d
end
```

Out[12]: dsol (generic function with 1 method)

In [13]:
```julia
function multi_sol(nlp, x, Fx, Jx, λ, τ; nl = 3)
    λ=zeros(nl)
    if nl % 2 == 0
        for i=1:nl
            λ[i] = τ/(10^(nl/2 - i + 2))
        end
    else
        for i=1:nl
            λ[i] = τ/(10^((nl-1)/2 - i + 1))
```

```julia
        end
    end

    for valeur in λ
        print(valeur)
        d = dsol(Fx, Jx, valeur)
        x_new = x + d
        norm_sq = norm(Fx(x_new))^2
        if norm_sq < min_norm_sq
            min_norm_sq = norm_sq
            best_d = d
        end
    end

    return best_d
end
```

Out[13]: multi_sol (generic function with 1 method)

In [7]:

```julia
function lm_param(nlp      :: AbstractNLSModel,
                  x        :: AbstractVector,
                  ϵ        :: AbstractFloat;
                  η₁       :: AbstractFloat = 1e-3,
                  η₂       :: AbstractFloat = 0.66,
                  σ₁       :: AbstractFloat = 10.0,
                  σ₂       :: AbstractFloat = 0.5,
                  max_eval :: Int = 10_000,
                  max_time :: AbstractFloat = 60.,
                  max_iter :: Int = typemax(Int64)
                  )
    ######################################################
    Fx = NLPModels.residual(himmelblau_nls, x)
    Jx = NLPModels.jac_residual(himmelblau_nls, x)
    ######################################################
    normFx   = norm(Fx)
    normdual = norm(Jx' * Fx)

    iter = 0
    λ = 0.0
    λ₀ = 1e-6
    η = 0.5
    τ = η * normdual

    el_time = 0.0
    tired   = neval_residual(nlp) > max_eval || el_time > max_time
    status  = :unknown

    start_time = time()
    too_small  = false
    optimal    = min(normFx, normdual) ≤ ϵ

    @info log_header([:iter, :nf, :primal, :status, :nd, :λ],
    [Int, Int, Float64, String, Float64, Float64],
    hdr_override=Dict(:nf => "#F", :primal => "‖F(x)‖", :nd => "‖d‖"))

    while !(optimal || tired || too_small)
```

```
    ###########################
    # (d, stats)  = lsqr(Jx, -Fx, λ = λ, atol = τ)
    d = multi_sol(nlp, x, Fx, Jx, λ, τ)
    ###########################

    too_small = norm(d) < 1e-16
    if too_small #the direction is too small
        status = :too_small
    else
        xp      = x + d
        ###########################
        Fxp     = NLPModels.residual(himmelblau_nls, xp)
        ###########################
        normFxp = norm(Fxp)

        Pred = 0.5 * (normFx^2 - norm(Jx * d + Fx)^2 - λ*norm(d)^2)
        Ared = 0.5 * (normFx^2 - normFxp^2)

        if Ared/Pred < η₁
            λ = max(λ₀, σ₁ * λ)
            status = :increase_λ
        else #success
            x  = xp
            ###########################
            Jx = NLPModels.jac_residual(himmelblau_nls, x)
            ###########################
            Fx = Fxp
            normFx = normFxp
            status = :success
            if Ared/Pred > η₂
                λ = max(λ * σ₂, λ₀)
            end
        end
    end

    @info log_row(Any[iter, neval_residual(nlp), normFx, status, norm(d), λ])

    el_time      = time() - start_time
    iter        += 1
    many_evals   = neval_residual(nlp) > max_eval
    iter_limit   = iter > max_iter
    tired        = many_evals || el_time > max_time || iter_limit
    normdual     = norm(Jx' * Fx)
    optimal      = min(normFx, normdual) ≤ ϵ

    η = λ == 0.0 ? min(0.5, 1/iter, normdual) : min(0.5, 1/iter)
    τ = η * normdual
end

status = if optimal
    :first_order
elseif tired
    if neval_residual(nlp) > max_eval
        :max_eval
    elseif el_time > max_time
```

```julia
            :max_time
        elseif iter > max_iter
            :max_iter
        else
            :unknown_tired
        end
    elseif too_small
        :stalled
    else
        :unknown
    end

    return GenericExecutionStats(nlp; status, solution = x,
                                 objective = normFx^2 / 2,
                                 dual_feas = normdual,
                                 iter = iter,
                                 elapsed_time = el_time)
end
```

Out[7]: lm_param (generic function with 1 method)

In [14]:
```julia
stats = lm_param(himmelblau_nls, himmelblau_nls.meta.x0, 1e-6)
@test stats.status == :first_order
```

```
821.663449959897
[ Info:   iter      #F     ‖F(x)‖           status        ‖d‖          λ
```

```
MethodError: no method matching lsqr(::SparseArrays.SparseMatrixCSC{Float64, Int64}, ::Vector{Float64}, ::Float64)

Closest candidates are:
  lsqr(::Any, ::AbstractVector{FC}; window, M, N, ldiv, sqd, λ, radius, etol, axtol, btol, conlim, atol, rtol, itmax, timemax, verbose, history, callback, iostream) where {T<:AbstractFloat, FC<:Union{Complex{T}, T}}
   @ Krylov C:\Users\Ulrizpascuit\.julia\packages\Krylov\pv2NF\src\lsqr.jl:158


Stacktrace:
 [1] dsol(Fx::Vector{Float64}, Jx::SparseArrays.SparseMatrixCSC{Float64, Int64}, λ::Float64)
   @ Main .\In[12]:2
 [2] multi_sol(nlp::ADNLSModel{Float64, Vector{Float64}, Vector{Int64}}, x::Vector{Float64}, Fx::Vector{Float64}, Jx::SparseArrays.SparseMatrixCSC{Float64, Int64}, λ::Float64, τ::Float64; nl::Int64)
   @ Main .\In[13]:15
 [3] multi_sol(nlp::ADNLSModel{Float64, Vector{Float64}, Vector{Int64}}, x::Vector{Float64}, Fx::Vector{Float64}, Jx::SparseArrays.SparseMatrixCSC{Float64, Int64}, λ::Float64, τ::Float64)
   @ Main .\In[13]:1
 [4] lm_param(nlp::ADNLSModel{Float64, Vector{Float64}, Vector{Int64}}, x::Vector{Float64}, ϵ::Float64; η₁::Float64, η₂::Float64, σ₁::Float64, σ₂::Float64, max_eval::Int64, max_time::Float64, max_iter::Int64)
   @ Main .\In[7]:41
 [5] lm_param(nlp::ADNLSModel{Float64, Vector{Float64}, Vector{Int64}}, x::Vector{Float64}, ϵ::Float64)
   @ Main .\In[7]:1
 [6] top-level scope
   @ In[14]:1
```

Exercice 3: Rocket Control

Dans les cellules ci-dessous nous introduisons un modèle de contrôle optimal (cf. https://en.wikipedia.org/wiki/Optimal_control) pour le contrôle d'une fusée dont une version discrétisée a été modélisé avec JuMP:

Le lien vers le tutoriel: https://nbviewer.jupyter.org/github/jump-dev/JuMPTutorials.jl/blob/master/notebook/modelling/rocket_control.ipynb

In [ ]:
```julia
using JuMP, Ipopt

# Create JuMP model, using Ipopt as the solver
rocket = Model(optimizer_with_attributes(Ipopt.Optimizer, "print_level" => 0))

# Constants
# Note that all parameters in the model have been normalized
# to be dimensionless. See the COPS3 paper for more info.
h_0 = 1    # Initial height
v_0 = 0    # Initial velocity
m_0 = 1    # Initial mass
g_0 = 1    # Gravity at the surface
```

```
T_c = 3.5  # Used for thrust
h_c = 500  # Used for drag
v_c = 620  # Used for drag
m_c = 0.6  # Fraction of initial mass left at end

c     = 0.5 * sqrt(g_0 * h_0)  # Thrust-to-fuel mass
m_f   = m_c * m_0             # Final mass
D_c   = 0.5 * v_c * m_0 / g_0    # Drag scaling
T_max = T_c * g_0 * m_0        # Maximum thrust

n = 800   # Time steps

@variables(rocket, begin
    Δt ≥ 0, (start = 1/n) # Time step
    # State variables
    v[1:n] ≥ 0            # Velocity
    h[1:n] ≥ h_0          # Height
    m_f ≤ m[1:n] ≤ m_0    # Mass
    # Control
    0 ≤ T[1:n] ≤ T_max    # Thrust
end)

# Objective: maximize altitude at end of time of flight
@objective(rocket, Max, h[n])

# Initial conditions
@constraints(rocket, begin
    v[1] == v_0
    h[1] == h_0
    m[1] == m_0
    m[n] == m_f
end)

# Forces
# Drag(h,v) = Dc v^2 exp( -hc * (h - h0) / h0 )
@NLexpression(rocket, drag[j = 1:n], D_c * (v[j]^2) * exp(-h_c * (h[j] - h_0) / h_0
# Grav(h)   = go * (h0 / h)^2
@NLexpression(rocket, grav[j = 1:n], g_0 * (h_0 / h[j])^2)
# Time of flight
@NLexpression(rocket, t_f, Δt * n)

# Dynamics
for j in 2:n
    # h' = v

    # Rectangular integration
    # @NLconstraint(rocket, h[j] == h[j - 1] + Δt * v[j - 1])

    # Trapezoidal integration
    @NLconstraint(rocket,
        h[j] == h[j - 1] + 0.5 * Δt * (v[j] + v[j - 1]))

    # v' = (T-D(h,v))/m - g(h)

    # Rectangular integration
```

```julia
    # @NLconstraint(rocket, v[j] == v[j - 1] + Δt *(
    #                  (T[j - 1] - drag[j - 1]) / m[j - 1] - grav[j - 1]))

    # Trapezoidal integration
    @NLconstraint(rocket,
        v[j] == v[j-1] + 0.5 * Δt * (
            (T[j] - drag[j] - m[j] * grav[j]) / m[j] +
            (T[j - 1] - drag[j - 1] - m[j - 1] * grav[j - 1]) / m[j - 1]))

    # m' = -T/c

    # Rectangular integration
    # @NLconstraint(rocket, m[j] == m[j - 1] - Δt * T[j - 1] / c)

    # Trapezoidal integration
    @NLconstraint(rocket,
        m[j] == m[j - 1] - 0.5 * Δt * (T[j] + T[j-1]) / c)
end
```

In [ ]:

```julia
# Solve for the control and state
println("Solving...")
status = optimize!(rocket)

# Display results
# println("Solver status: ", status)
println("Max height: ", objective_value(rocket))
```

In [ ]:

```julia
value.(h)[n]
```

In [ ]:

```julia
# Can visualize the state and control variables
using Gadfly
```

In [ ]:

```julia
h_plot = plot(x = (1:n) * value.(Δt), y = value.(h)[:], Geom.line,
                Guide.xlabel("Time (s)"), Guide.ylabel("Altitude"))
m_plot = plot(x = (1:n) * value.(Δt), y = value.(m)[:], Geom.line,
                Guide.xlabel("Time (s)"), Guide.ylabel("Mass"))
v_plot = plot(x = (1:n) * value.(Δt), y = value.(v)[:], Geom.line,
                Guide.xlabel("Time (s)"), Guide.ylabel("Velocity"))
T_plot = plot(x = (1:n) * value.(Δt), y = value.(T)[:], Geom.line,
                Guide.xlabel("Time (s)"), Guide.ylabel("Thrust"))
draw(SVG(6inch, 6inch), vstack(hstack(h_plot, m_plot), hstack(v_plot, T_plot)))
```

Questions:

```
- i) Transformer le modèle JuMP utilisé ci-dessus en un NLPModel en
utilisant le package `NLPModelsJuMP`.
- ii) Résoudre ce nouveau modèle avec `Ipopt` en utilisant
`NLPModelsIpopt`.
- iii) Calcul séparément la différence entre les h,v,m,T, Δt
calculés.
- iv) Est-ce que le contrôle T atteint ses bornes ?
- v) Reproduire les graphiques ci-dessous avec la solution calculée
via `NLPModelsIpopt`.
```

In [ ]:

```julia
using NLPModels, LinearAlgebra, NLPModelsJuMP, NLPModelsIpopt
```