CONTROL PREDICTIVO BASADO EN MODELO CON ENTRADA FINITA PARA SEGUIMIENTO DE CAMINO ACELERADO CON GPU

Román Comelli[†], Ernesto Kofman[†], Carlos Bederián[‡] y Nicolás Wolovick[‡]

[†]*CIFASIS, CONICET-UNR, Ocampo y Esmeralda, 2000 Rosario, Argentina*
[‡]*FaMAF, UNC, Ciudad Universitaria, 5000 Córdoba, Argentina*

Resumen: En este trabajo se presentan dos implementaciones simples para placa de video (GPU, por sus siglas en inglés) de una estrategia de Control Predictivo Basado en Modelo con Conjunto de Entrada Finito para seguimiento de camino con un robot tipo auto. Las pruebas realizadas sobre una GPU Nvidia Jetson TX2 muestran la factibilidad de usar los algoritmos desarrollados en aplicaciones de control con restricciones de tiempo real.

Palabras clave: *control predictivo basado en modelo, entrada finita, gpu, seguimiento de camino, tiempo real*
2000 AMS Subject Classification: 65Y05 - 90C10 - 70E60

1. INTRODUCCIÓN

En el Centro Internacional Franco-Argentino de Ciencias de la Información y de Sistemas (CIFASIS) se está desarrollando un robot desmalezador para el campo cuyo prototipo puede verse en la Figura 1. Su objetivo final es recorrer plantaciones de soja en forma autónoma identificando malezas y aplicándoles agroquímicos localmente [5]. Una de las tareas que tiene que resolver el robot para lograr esto es la de poder seguir un camino predeterminado, controlando coordinadamente velocidad y dirección. Como la primera de estas variables es baja y prácticamente constante, se usa un control simple PI independiente para ella. Con respecto a la dirección, se ha optado por utilizar una estrategia de Control Predictivo Basado en Modelo (MPC, por sus siglas en inglés) [6]. Este tipo de controladores, considerando restricciones en el estado y la entrada, usa un modelo del sistema para determinar la acción de control óptima a aplicar para conseguir la salida deseada, reduciendo un cierto costo. Como en general la entrada es continua, la correspondiente optimización suele resolverse mediante algoritmos iterativos que computan el gradiente de dicho costo.

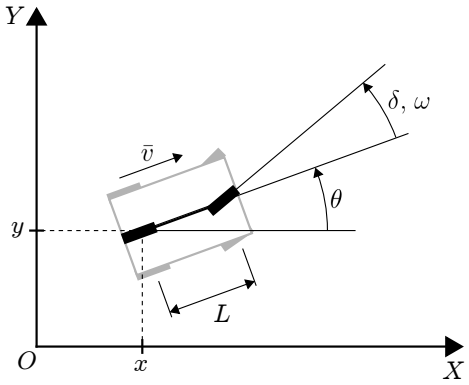
Figura 1: Robot desmalezador.

En particular nuestro prototipo tiene la característica de que su dirección se mueve a velocidad aproximadamente constante, por lo que puede considerarse que las entradas posibles de aplicar en cada instante de control son únicamente tres: mantener la dirección como estaba o girar, a la izquierda o a la derecha, un cierto ángulo fijo. En este contexto de conjunto de entrada finito, el problema de optimización de MPC puede resolverse simulando, hasta cierto tiempo futuro, todas las evoluciones posibles del sistema con las distintas combinaciones de entrada factibles. Debe notarse que como cada simulación es independiente, este enfoque resulta adecuado para arquitecturas de paralelismo masivo. Asumiendo un horizonte de control H_c, en este caso sería necesario realizar 3^{H_c} simulaciones. Luego, calculando para cada una el correspondiente costo, puede uno quedarse con la secuencia de entradas de control óptima, es decir, aquella que lo minimiza. Terminado esto, correspondería aplicar la primera acción de dicha secuencia de control y luego, como en toda técnica de horizonte móvil, esperar al próximo paso del controlador para volver a repetir todo el procedimiento, con mediciones actualizadas de las variables que haya disponibles.

La forma descripta de proceder se denomina MPC con Conjunto Finito de Control (FCS-MPC) [4] y en este trabajo se presenta una implementación, y una variante de la misma, para Placa de Video (GPU) de la parte central de este esquema de control. Hay que mencionar que aunque ya se ha usado FCS-MPC computándose en una GPU [2, 3], la aplicación, las condiciones y/o la metodología han sido distintas.

2. MODELO Y PARÁMETROS DEL CONTROLADOR

Un modelo útil para representar robots tipo auto es el de bicicleta, que es simple y por lo tanto adecuado para usarse en esquemas como el de FCS-MPC. Este modelo, dado en (1), se esquematiza en la Figura 2.



$$\begin{cases} x(k+1) = x(k) + v(k)\cos(\theta(k))\Delta t, \\ y(k+1) = y(k) + v(k)\sin(\theta(k))\Delta t, \\ \theta(k+1) = \theta(k) + \frac{v(k)}{L}\tan(\delta(k))\Delta t, \\ \delta(k+1) = \delta(k) + \omega(k)\Delta t. \end{cases} \tag{1}$$

Figura 2: Esquema del modelo de bicicleta.

Las variables de estado son la posición de la rueda trasera $\boldsymbol{p} = [x, y]^T \in \mathbb{R}^2$, la orientación $\theta \in [-\pi, \pi)$ y el ángulo de la dirección $\delta \in [\delta_{min}, \delta_{max}]$. Las entradas de este modelo serían la velocidad lineal de la rueda trasera v y la velocidad angular de la dirección $\omega \in \{-\omega_f, 0, \omega_f\}$, siendo ω_f una velocidad angular fija. Sin embargo, dado que en este caso solo ω es de interés para el control, los valores $v(k)$ serán constantes predefinidas en lugar de variables manipulables o entradas. Finalmente, Δt es el período de muestreo y L es la distancia entre ruedas.

Respecto del controlador, además del ya mencionado horizonte de control H_c, que representa la cantidad de pasos desde el comienzo de la ventana de simulación en los que las acciones de control pueden cambiar, otro parámetro es el horizonte de predicción H_p, es decir, el número total de pasos de la simulación. A su vez, otro aspecto fundamental de cualquier estrategia de MPC es la función de costo. En particular, se utilizó la de la Ecuación (2). Para mayor información se recomienda ver [1], donde se explica también cómo se calcula la referencia que aquí ya se asume conocida (no es adecuada para computarse en GPU).

$$J = \sum_{i=1}^{H_p} \left(\|\boldsymbol{p}(i) - \boldsymbol{p_{ref}}(i)\| + |(\theta(i) - \theta_{ref}(i))\, v_{ref}(i)\Delta t| \right). \tag{2}$$

3. Implementación en CUDA

Las etapas de procesamiento a implementar en GPU eran, para cada simulación: generación de la secuencia de entrada, cómputo de la evolución, obtención de los costos y selección del mínimo. Conservar la separación entre estas partes requería mantener el resultado de cada una en memoria por lo que se optó por combinar las primeras tres etapas en una única. Esto es lo mostrado en el Fragmento de Código 1.

```
__global__ void PredictAndEvaluate(const real * __restrict__ ref,
    const real * __restrict__ meas, real * __restrict__ costs) {
  uint idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (idx < simulations) {
    real x[4];
    memcpy(x, meas, states * sizeof(real));
    uint num = idx;
    uint den = simulations / 3; // 3 for the cardinality of the control
    int control;
    real cost = 0;
    for (unsigned int i = 0; i < prediction_horizon; ++i) {
      if (i < control_horizon) { // Generate control locally, depending on idx
        control = num / den - 1;
        num %= den;
        den /= 3;}
      else { control = 0; }
      x[0] += step_time * ref[Get2DIdx(3, i)] * CudaCos(x[2]);
```

```
      x[1] += step_time * ref[Get2DIdx(3, i)] * CudaSin(x[2]);
      cost += CudaSqrt((x[0] - ref[Get2DIdx(0, i + 1)]) * (x[0] -
          ref[Get2DIdx(0, i + 1)]) + (x[1] - ref[Get2DIdx(1, i + 1)]) * (x[1] -
          ref[Get2DIdx(1, i + 1)]));
      x[2] += step_time * ref[Get2DIdx(3, i)] * CudaTan(x[3]) / wheelbase;
      x[2] = WrapToPi(x[2]);
      x[3] += step_time * angular_speed * control;
      x[3] = SaturateSteering(x[3]);
      cost += CudaFabs(WrapToPi(x[2] - ref[Get2DIdx(2, i + 1)]) * step_time *
          ref[Get2DIdx(3, i + 1)]);}
    costs[idx] = cost;}}
```

Fragmento de Código 1: *Kernel* de la primera implementación.

El *kernel* recibe las referencias $\boldsymbol{p_{ref}}$, θ_{ref} y v_{ref} en el argumento `ref`, y la medición del estado, es decir x, y, θ y δ, en `meas`. Por su parte, en `costs` se devuelven todos los costos obtenidos. Lo que se hace en el código es, luego de ubicarse en la simulación siendo ejecutada, inicializar variables y pasar al ciclo que calcula la evolución del estado. Dentro de este `for`, primero se determina la acción de control a aplicar asignándole -1, 0 o 1 a la variable `control` según el índice del hilo y el número del paso. Luego se van obteniendo las variables de estado y en simultáneo se calcula el aporte al costo. En particular, también se mantienen acotados los ángulos θ (en $[-\pi, \pi)$) y δ (hasta ángulo máximo por limitación física). Como puede verse, si bien los costos se almacenan finalmente en la memoria principal, tanto las entradas de velocidad angular como los estados (solo correspondientes al último instante de tiempo) están únicamente en la memoria de cada hilo, presumiblemente en registros por ser pocas variables.

El paso final de hallar el índice del costo mínimo (lo único necesario para reconstruir el valor de la entrada de interés), se hace mediante un llamado a la función `cublasIsamin` (o `cublasIdamin` para `double`) de la librería CUBLAS, por fuera del *kernel* presentado.

3.1. Variante con descarte

En esta implementación, algunas simulaciones cuyos costos necesariamente serán mayores o iguales a otros, son descartadas. La existencia de saturación en el ángulo de la dirección es lo que permite saber esto. A modo de ejemplo, supongamos que el robot tiene su dirección saturada hacia la izquierda. En esta situación, de las tres acciones de control posibles, (intentar) girar a la izquierda ocasionaría el mismo efecto que mantener la dirección tal como está. Extendiendo esta idea puede decirse que toda secuencia de entradas que, estando el robot con el ángulo de su dirección saturado, sean tendientes a "saturar más" dicha variable, pueden descartarse ya que tendrán, en el mejor de los casos, el mismo costo que otra que frente a esta situación mantenga la dirección como estaba. La implementación de esto se muestra en el Fragmento de Código 2. Si la situación mencionada se detecta, el *thread* deja de simular y define su costo como infinito. En este caso, la función `SaturateSteering` es levemente diferente a la de la primera implementación.

```
__global__ void PredictAndEvaluate(...) {...
  if (idx < simulations) {...
    for (unsigned int i = 0; i < prediction_horizon; ++i) {...
      x[3] += step_time * angular_speed * control;
      if (SaturateSteering(x[3])) {
        cost = INFINITY;
        break;}...}...}}
```

Fragmento de Código 2: *Kernel* con descarte.

4. Experimentos

Las pruebas aquí presentadas se realizaron con una GPU Nvidia Jetson TX2 (CC 6.2, arquitectura Pascal, 256 CUDA cores, 8 GB de RAM, 1300 MHz). Los experimentos consistieron en medir tiempo total promedio de ejecución variando el tamaño de bloque, la referencia de camino y el tamaño del problema

(H_c), manteniendo $H_p = 25$. El objetivo era determinar si alguna implementación era mejor y hasta cuánto se podía subir H_c considerando como tiempo máximo de ejecución 200 ms (período de control del robot).

Tabla 1: Tiempo de cómputo en ms para distintas referencias de camino.

(a) Camino derecho.

H_c	Orig.	Descarte	CPU
9	0.398	0.338	45.0
10	0.568	0.605	145
11	1.33	1.44	-
12	3.73	4.07	-
13	10.9	12.0	-
14	33.0	36.0	-
15	100	109	-

(b) Camino con curva.

H_c	Orig.	Descarte	CPU
9	0.320	0.328	45.0
10	0.570	0.554	145
11	1.33	1.28	-
12	3.73	3.49	-
13	10.9	10.0	-
14	32.9	29.7	-
15	99.9	88.7	-

(c) Camino con curva pronunciada.

H_c	Orig.	Descarte	CPU
9	0.359	0.265	45.0
10	0.571	0.394	145
11	1.33	0.728	-
12	3.73	1.85	-
13	10.9	5.10	-
14	32.9	14.8	-
15	100	44.1	-

En la Tabla 1 se observan los tiempos promedio obtenidos, discriminados por referencia. Las pruebas con distintos tamaños de bloque indicaron que usar 256 para este parámetro era adecuado en todos los casos, por lo que sólo se presentan mediciones correspondientes a dicho tamaño de bloque. A los fines comparativos, también se agregaron mediciones de una implementación vectorizada en Matlab R2017b, obtenidas con un procesador Intel Core i5-10400. Debe notarse que los tiempos aproximadamente se triplican al incrementarse en uno el horizonte de control, por lo que H_c pudo subirse hasta 15 para GPU y hasta 10 para CPU. Al haber cuatro variables de estado, la cantidad de variables calculadas en cada instante de control son $4 \cdot H_p \cdot 3^{H_c}$, sin contar el cómputo del costo y demás variables auxiliares.

Puede observarse que mientras que la implementación original requiere aproximadamente el mismo tiempo en todos los casos, para la variante con descarte esto cambia considerablemente. Cuando el ángulo de la dirección está cerca de la saturación los tiempos disminuyen considerablemente (referencia de camino con curva pronunciada) mientras que, en el caso contrario, se incrementan (referencia con camino derecho). Estos resultados sugieren la posibilidad de usar una versión combinada de ambos enfoques que decida entre una u otra implementación dependiendo de la referencia recibida en cada instante de tiempo. Si bien en aplicaciones de tiempo real lo que importa es el peor caso (y haciendo esto no se estaría mejorando en este sentido), una menor utilización de la GPU sería favorable para tenerla disponible para otras tareas.

5. Conclusiones

Se presentaron una implementación, y una variante de la misma, para GPU de un algoritmo de FCS-MPC para seguimiento de camino. Mediante experimentos con una GPU Nvidia Jetson TX2, se verificó la factibilidad de usar dichas implementaciones en un controlador de tiempo real. Como trabajo futuro se plantea incorporar a estos algoritmos las modificaciones necesarias para garantizar estabilidad y hacer experimentos sobre el robot verdadero. Se plantea también continuar mejorando la implementación.

Referencias

[1] Comelli, Román, Ernesto Kofman y Diego Feroldi: *MPC con Conjunto Finito de Control para Seguimiento de Camino*. En *Actas de XIX Reunión de Trabajo en Procesamiento de la Información y Control (RPIC 2021)*, páginas 1–6, 2021.

[2] Kozubik, Michal y Pavel Vaclavek: *Speed Control of PMSM with Finite Control Set Model Predictive Control Using General-purpose Computing on GPU*. En *Proc. of XLVI Annual Conference of the IEEE Industrial Electronics Society (IECON 2020)*, páginas 379–383, 2020.

[3] Phung, Duc Kien, Bruno Hérissé, Julien Marzat y Sylvain Bertrand: *Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit*. IFAC-PapersOnLine, 50(1):11883–11888, 2017.

[4] Picasso, Bruno, Stefania Pancanti, Alberto Bemporad y Antonio Bicchi: *Receding-Horizon Control of LTI Systems with Quantized Inputs1*. IFAC Proceedings Volumes, 36(6):259–264, 2003.

[5] Pire, Taihú, Martín Mujica, Javier Civera y Ernesto Kofman: *The Rosario Dataset: Multisensor Data for Localization and Mapping in Agricultural Environments*. Intl. J. of Robotics Research, 38(6):633–641, 2019.

[6] Rawlings, James B., David Q. Mayne y Moritz Diehl: *Model predictive control: theory, computation, and design*, volumen 2. Nob Hill Publishing Madison, WI, 2017.