Park 轉換

Park 轉換 (Park Transformation) 是電機工程中一種關鍵的數學工具，常用於電機控制與向量控制 (Field-Oriented Control, FOC)。其基本概念是將二維靜態正交坐標 ($\alpha - \beta$ 座標) 轉換至二維旋轉坐標 ($\mathbf{d} - \mathbf{q}$ 座標)，以便簡化交流電機的控制與分析。

幾何原理與座標系統

在交流電機控制中，為了便於分析與控制，我們通常使用下列兩種不同的坐標系：

- $\alpha - \beta$ **座標系**：靜態的直角坐標系，通常來自於 **Clarke 轉換** ，(i_α) 和 (i_β) 代表兩個正交的電壓或電流分量。
- **d-q 座標系**：一個相對於轉子磁場同步旋轉的坐標系，其中 **d 軸(Direct Axis)** 與磁場對齊， **q 軸 (Quadrature Axis)** 則與 **d 軸** 垂直。

核心概念就是將 **Clarke 轉換** 後的 $\alpha - \beta$ 正交靜止座標，透過代入旋轉的角速度 (θ) 參數進行同步旋轉，使其投影到 $\mathbf{d} - \mathbf{q}$ 旋轉座標系 。在 **馬達上的角速度** 指的是 **電角速度(Electrical Angular Velocity)**

這樣本來觀察的弦波，就會變成直流信號，因為我們觀察的參考座標時刻隨著弦波同步旋轉，達到相對靜止的效果。

Park 轉換的推導過程

幾何推導

考慮一個向量($\mathbf{I}$) 在 $\alpha - \beta$ 坐標系中的表示為：

$$\mathbf{I} = I_\alpha \hat{\imath} + I_\beta \hat{\jmath}$$

其中 $(\hat{\imath})$ 與 $(\hat{\jmath})$ 分別是 (α) 與 (β) 軸的單位向量。

現在，假設 $\mathbf{d} - \mathbf{q}$ 坐標系相對於 $\alpha - \beta$ 坐標系旋轉了一個角度 (θ) ，我們希望將 $(\mathbf{I})$ 重新表示在 $\mathbf{d} - \mathbf{q}$ 坐標系上。

d-q 坐標軸的表達式

當 dq 坐標系相對於 $\alpha - \beta$ 坐標系 **順時針旋轉** (θ) 角時，其新的基底向量 (單位向量) 可以表示為：

$$\begin{aligned}\hat{d} &= \cos\theta\hat{\imath} + \sin\theta\hat{\jmath} \\ \hat{q} &= -\sin\theta\hat{\imath} + \cos\theta\hat{\jmath}\end{aligned}$$

這兩個向量的推導來自於標準二維旋轉矩陣的概念。若一個點 (x, y) **逆時針旋轉** (θ) 角後的新坐標 (x', y') 滿足：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

則相對於 $\alpha - \beta$ 坐標系， $\mathbf{d} - \mathbf{q}$ 坐標的單位向量由此獲得，可以觀察到，就是只是旋轉方式的不同，一個 **逆時針方向**，一個 **順時針方向**。

向量投影計算

由於 $\mathbf{d} - \mathbf{q}$ 軸的方向已知，$(\mathbf{I})$ 在 $\mathbf{d} - \mathbf{q}$ 坐標上的投影可以通過 **內積** 計算：

$$i_d = \mathbf{I} \cdot \widehat{d} = i_\alpha \cos\theta + i_\beta \sin\theta$$
$$i_q = \mathbf{I} \cdot \widehat{q} = -i_\alpha \sin\theta + i_\beta \cos\theta$$

這兩個方程式可以用矩陣形式表示為：

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix}$$

旋轉矩陣的特性

上述矩陣是一個 **正交旋轉矩陣**，其特性如下：

1. 行向量與列向量皆為單位正交向量，確保坐標變換不改變向量長度。
2. 逆變換是其轉置，即：

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

這表示若已知 $\mathbf{d} - \mathbf{q}$ 坐標，可以逆推回 $\alpha - \beta$ 坐標。

物理意義與應用

Park 轉換的主要優勢是簡化交流電機的控制。經過變換後：

- **d 軸** 分量對應於磁場方向的電壓或電流，用於控制磁場強度。因為 **d 軸** 與轉子磁場對齊，調節 $\mathbf{i_d}$ 影響磁鏈強度，進而影響電機勵磁。
- **q 軸** 分量對應於垂直於磁場的電壓或電流，用於控制轉矩。由於轉矩是由定子電流與轉子磁場的交互作用產生，因此調節 $\mathbf{i_q}$ 可以直接影響轉矩。

這樣，原本複雜的三相交流電機數學模型轉變為類似直流電機的控制方式，使向量控制更加直觀。

Park 的程式碼實現

下面是ODrive 使用的代碼，位於 **FieldOriented::get_alpha_beta_output()** 內，函式內進行了 **Park轉換** 與計算完畢後的 **逆Park轉換** 。

```
ODriveIntf::MotorIntf::Error FieldOrientedController::get_alpha_beta_output(
   uint32_t output_timestamp, std::optional<float2D>* mod_alpha_beta,
   std::optional<float>* ibus) {

if (!vbus_voltage_measured_.has_value() || !Ialpha_beta_measured_.has_value()) {
   // FOC didn't receive a current measurement yet.
   return Motor::ERROR_CONTROLLER_INITIALIZING;
} else if (abs((int32_t)(i_timestamp_ - ctrl_timestamp_)) > MAX_CONTROL_LOOP_UPDATE_TO_CURRE
   // Data from control loop and current measurement are too far apart.
   return Motor::ERROR_BAD_TIMING;
}

// TODO: improve efficiency in case PWM updates are requested at a higher
// rate than current sensor updates. In this case we can reuse mod_d and
// mod_q from a previous iteration.

if (!Vdq_setpoint_.has_value()) {
   return Motor::ERROR_UNKNOWN_VOLTAGE_COMMAND;
} else if (!phase_.has_value() || !phase_vel_.has_value()) {
   return Motor::ERROR_UNKNOWN_PHASE_ESTIMATE;
} else if (!vbus_voltage_measured_.has_value()) {
   return Motor::ERROR_UNKNOWN_VBUS_VOLTAGE;
}
```

```
auto [Vd, Vq] = *Vdq_setpoint_;
float phase = *phase_;
float phase_vel = *phase_vel_;
float vbus_voltage = *vbus_voltage_measured_;

std::optional<float2D> Idq;

// Park transform
if (Ialpha_beta_measured_.has_value()) {
   auto [Ialpha, Ibeta] = *Ialpha_beta_measured_;
   float I_phase = phase + phase_vel * ((float)(int32_t)(i_timestamp_ - ctrl_timestamp_) / (
   float c_I = our_arm_cos_f32(I_phase);
   float s_I = our_arm_sin_f32(I_phase);
   Idq = {
         c_I * Ialpha + s_I * Ibeta,
         c_I * Ibeta - s_I * Ialpha
   };
   Id_measured_ += I_measured_report_filter_k_ * (Idq->first - Id_measured_);
   Iq_measured_ += I_measured_report_filter_k_ * (Idq->second - Iq_measured_);
} else {
   Id_measured_ = 0.0f;
   Iq_measured_ = 0.0f;
}


float mod_to_V = (2.0f / 3.0f) * vbus_voltage;
float V_to_mod = 1.0f / mod_to_V;
float mod_d;
float mod_q;

if (enable_current_control_) {
   // Current control mode

   if (!pi_gains_.has_value()) {
         return Motor::ERROR_UNKNOWN_GAINS;
   } else if (!Idq.has_value()) {
         return Motor::ERROR_UNKNOWN_CURRENT_MEASUREMENT;
   } else if (!Idq_setpoint_.has_value()) {
         return Motor::ERROR_UNKNOWN_CURRENT_COMMAND;
   }

   auto [p_gain, i_gain] = *pi_gains_;
   auto [Id, Iq] = *Idq;
   auto [Id_setpoint, Iq_setpoint] = *Idq_setpoint_;

   float Ierr_d = Id_setpoint - Id;
   float Ierr_q = Iq_setpoint - Iq;

   // Apply PI control (V{d,q}_setpoint act as feed-forward terms in this mode)
   mod_d = V_to_mod * (Vd + v_current_control_integral_d_ + Ierr_d * p_gain);
   mod_q = V_to_mod * (Vq + v_current_control_integral_q_ + Ierr_q * p_gain);

   // Vector modulation saturation, lock integrator if saturated
   // TODO make maximum modulation configurable
   float mod_scalefactor = 0.80f * sqrt3_by_2 * 1.0f / std::sqrt(mod_d * mod_d + mod_q * moc
   if (mod_scalefactor < 1.0f) {
         mod_d *= mod_scalefactor;
         mod_q *= mod_scalefactor;
         // TODO make decayfactor configurable
         v_current_control_integral_d_ *= 0.99f;
         v_current_control_integral_q_ *= 0.99f;
   } else {
         v_current_control_integral_d_ += Ierr_d * (i_gain * current_meas_period);
         v_current_control_integral_q_ += Ierr_q * (i_gain * current_meas_period);
   }

} else {
   // Voltage control mode
   mod_d = V_to_mod * Vd;
   mod_q = V_to_mod * Vq;
}

// Inverse park transform
float pwm_phase = phase + phase_vel * ((float)(int32_t)(output_timestamp - ctrl_timestamp_)
float c_p = our_arm_cos_f32(pwm_phase);
float s_p = our_arm_sin_f32(pwm_phase);
float mod_alpha = c_p * mod_d - s_p * mod_q;
float mod_beta = c_p * mod_q + s_p * mod_d;

// Report final applied voltage in stationary frame (for sensorless estimator)
final_v_alpha_ = mod_to_V * mod_alpha;
final_v_beta_ = mod_to_V * mod_beta;

*mod_alpha_beta = {mod_alpha, mod_beta};

if (Idq.has_value()) {
   auto [Id, Iq] = *Idq;
   *ibus = mod_d * Id + mod_q * Iq;
   power_ = vbus_voltage * (*ibus).value();
}

return Motor::ERROR_NONE;
}
```

而下面的程式碼來自 STM MCSDK v6.2， 因為他的Clarke轉換出來的 $\mathbf{i}_\beta$ 方向相反，所以它轉換方程式也需要旋轉

```
/**
* @brief  This function transforms stator values alpha and beta, which
*         belong to a stationary qd reference frame, to a rotor flux
*         synchronous reference frame (properly oriented), so as q and d.
*                   d= alpha *sin(theta)+ beta *cos(Theta)
*                   q= alpha *cos(Theta)- beta *sin(Theta)
* @param  Input: stator values alpha and beta in alphabeta_t format.
* @param  Theta: rotating frame angular position in q1.15 format.
* @retval Stator values q and d in qd_t format
*/
__weak qd_t MCM_Park(alphabeta_t Input, int16_t Theta)
{
qd_t Output;
int32_t d_tmp_1;
int32_t d_tmp_2;
int32_t q_tmp_1;
int32_t q_tmp_2;
int32_t wqd_tmp;
int16_t hqd_tmp;
Trig_Components Local_Vector_Components;

Local_Vector_Components = MCM_Trig_Functions(Theta);

/* No overflow guaranteed */
q_tmp_1 = Input.alpha * ((int32_t )Local_Vector_Components.hCos);

/* No overflow guaranteed */
q_tmp_2 = Input.beta * ((int32_t)Local_Vector_Components.hSin);

/* Iq component in Q1.15 Format */
#ifndef FULL_MISRA_C_COMPLIANCY_MC_MATH
/* WARNING: the below instruction is not MISRA compliant, user should verify
   that Cortex-M3 assembly instruction ASR (arithmetic shift right) is used by
   the compiler to perform the shift (instead of LSR logical shift right) */
wqd_tmp = (q_tmp_1 - q_tmp_2) >> 15; //cstat !MISRAC2012-Rule-1.3_n !ATH-shift-neg !MISRAC2
#else
wqd_tmp = (q_tmp_1 - q_tmp_2) / 32768;
#endif
```

```c
/* Check saturation of Iq */
if (wqd_tmp > INT16_MAX)
{
   hqd_tmp = INT16_MAX;
}
else if (wqd_tmp < (-32768))
{
   hqd_tmp = ((int16_t)-32768);
}
else
{
   hqd_tmp = ((int16_t)wqd_tmp);
}

Output.q = hqd_tmp;

if (((int16_t)-32768) == Output.q)
{
   Output.q = -32767;
}
else
{
   /* Nothing to do */
}

/* No overflow guaranteed */
d_tmp_1 = Input.alpha * ((int32_t )Local_Vector_Components.hSin);

/* No overflow guaranteed */
d_tmp_2 = Input.beta * ((int32_t )Local_Vector_Components.hCos);

/* Id component in Q1.15 Format */
#ifndef FULL_MISRA_C_COMPLIANCY_MC_MATH
/* WARNING: the below instruction is not MISRA compliant, user should verify
   that Cortex-M3 assembly instruction ASR (arithmetic shift right) is used by
   the compiler to perform the shift (instead of LSR logical shift right) */
wqd_tmp = (d_tmp_1 + d_tmp_2) >> 15; //cstat !MISRAC2012-Rule-1.3_n !ATH-shift-neg !MISRAC20
#else
wqd_tmp = (d_tmp_1 + d_tmp_2) / 32768;
#endif

/* Check saturation of Id */
if (wqd_tmp > INT16_MAX)
{
   hqd_tmp = INT16_MAX;
}
else if (wqd_tmp < (-32768))
{
   hqd_tmp = ((int16_t)-32768);
}
else
{
   hqd_tmp = ((int16_t)wqd_tmp);
}

Output.d = hqd_tmp;

if (((int16_t)-32768) == Output.d)
{
   Output.d = -32767;
}
else
{
   /* Nothing to do */
}

   return (Output);
}
```

下面則是 ARM CMSIS DSP Pack 內的轉換函式 **Park轉換**

```c
/**
 * @brief Floating-point Park transform
 * @param[in]  Ialpha  input two-phase vector coordinate alpha
 * @param[in]  Ibeta   input two-phase vector coordinate beta
 * @param[out] pId     points to output   rotor reference frame d
 * @param[out] pIq     points to output   rotor reference frame q
 * @param[in]  sinVal  sine value of rotation angle theta
 * @param[in]  cosVal  cosine value of rotation angle theta
 * @return     none
 *
 * The function implements the forward Park transform.
 *
 */
__STATIC_FORCEINLINE void arm_park_f32(
float32_t Ialpha,
float32_t Ibeta,
float32_t * pId,
float32_t * pIq,
float32_t sinVal,
float32_t cosVal)
{
  /* Calculate pId using the equation, pId = Ialpha * cosVal + Ibeta * sinVal */
  *pId = Ialpha * cosVal + Ibeta * sinVal;

  /* Calculate pIq using the equation, pIq = - Ialpha * sinVal + Ibeta * cosVal */
  *pIq = -Ialpha * sinVal + Ibeta * cosVal;
}
```

結論

透過幾何分析與數學推導，我們得到了 Park 轉換的標準矩陣。這一變換是交流電機控制中的重要工具，使得三相電機的控制變得更為簡單高效。