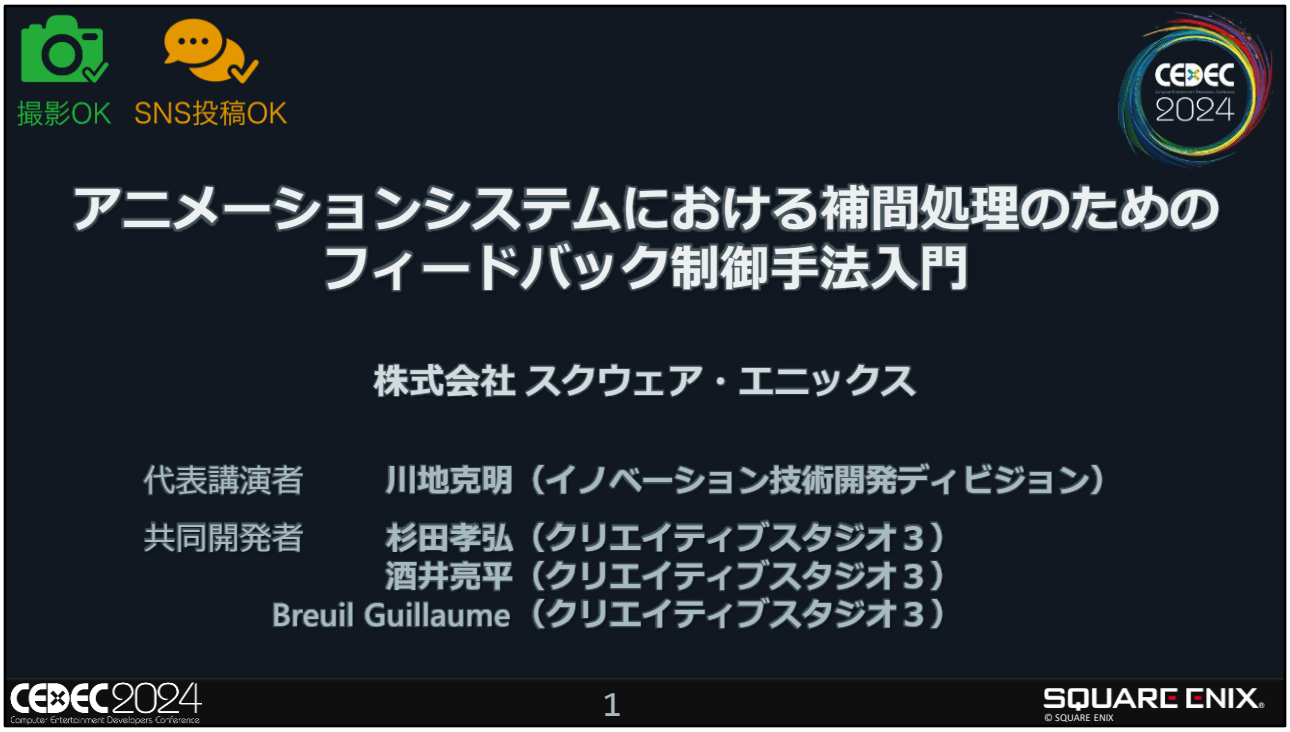
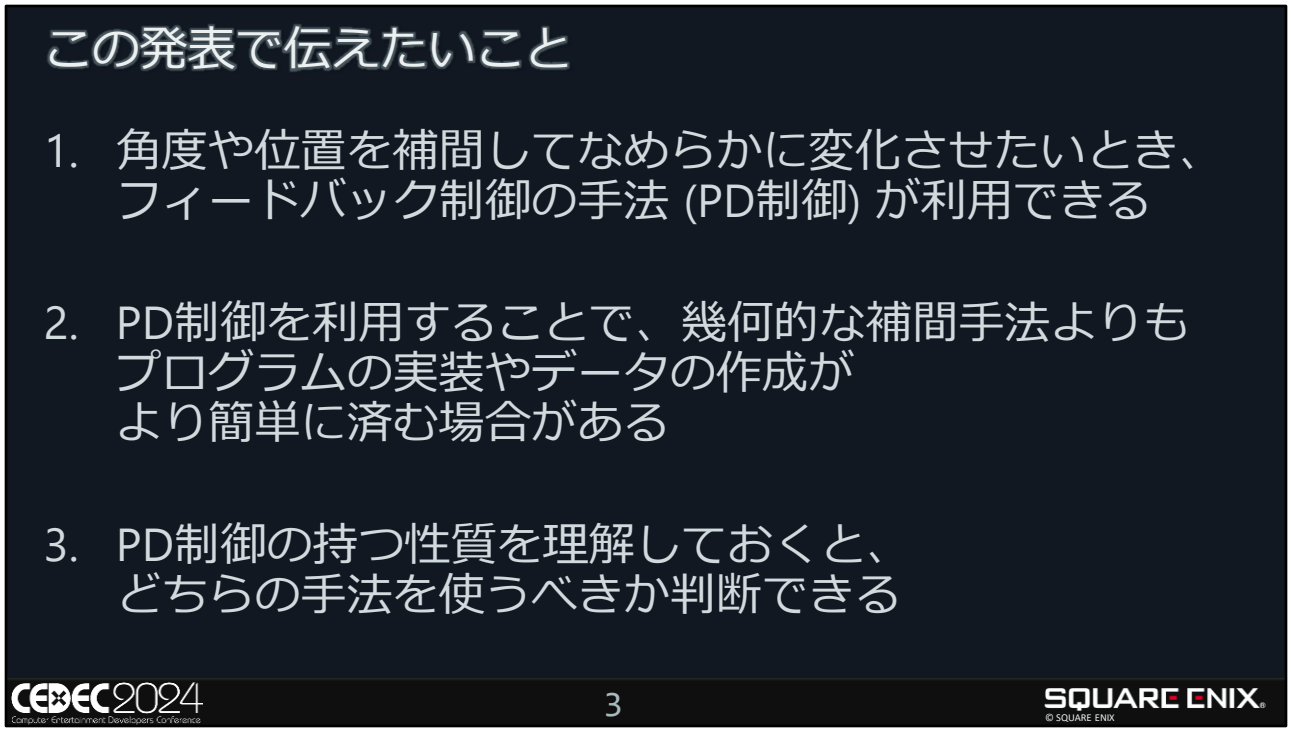


本セッションでは、アニメーションシステムにおける補間処理のためのフィードバック制御手法入門と題しまして、(株) スクウェア・エニックスより川地が代表して講演を行います。

本発表に関するご質問・ご意見には
CEDECオンラインライブ・チャットをご利用ください



それでは最初に、この発表の全体として伝えたいことについて、3点にまとめてお話します。

(1) まずはじめに、この発表ではフィードバック制御の手法について紹介します。
その手法の中でも、PD制御と呼ばれる手法について詳しく紹介し、
このPD制御が、アニメーションシステムにおいて角度や位置をなめらかに変化させる、いわゆる補間の処理に利用できることについて説明します。

(2) こうした角度や位置の補間を行う方法としては、フィードバック制御以外の手法としては幾何的な補間手法が広く使われています。

幾何的な補間手法とは、ブレンドカーブなどを用いて二つのアニメーションをなめらかにつなぐ方法です。
実績のある方法で、幾何的な補間の性質についてはよく知られています。

この発表では、PD制御による補間の持つ性質についても詳しく説明することで、
ゲームで必要とする補間結果を得るためのプログラムやデータの作成が、
幾何的な補間よりも簡単になる場合があることを説明します。

(3) また、PD 制御による補間には得意な分野とそうでない不得意な分野があります。

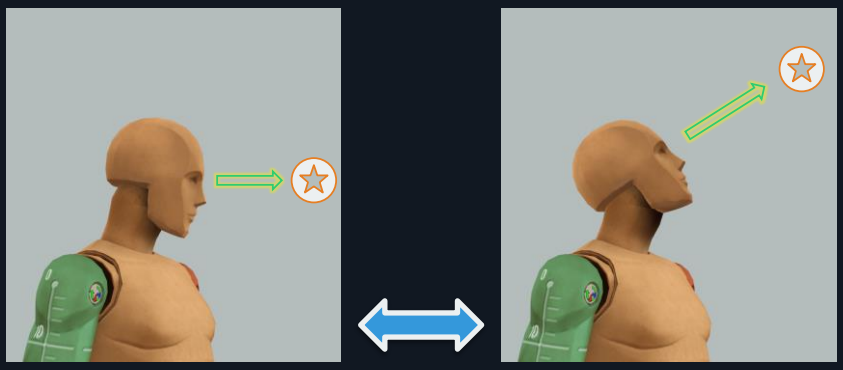
発表の中では、PD制御の長所と短所の両方について説明して、

アニメーションの補間処理を行いたいときに、
PD処理を使うべきなのか、そうでないのかの判断を可能にする助けとなるようにします。

次のスライドからは、アニメーションの補間処理について具体的に見ていきたいと思います。

角度の変化

(例) 注視する対象物が切り替わったとき、
首のジョイント角度を変化させる



アニメーションプログラムで角度の補間を行いたい場合には、
まずジョイントの角度が二つの状態の間で変化していることになります。

このスライドでは、首のジョイントの角度が変化している様子を図で表しています。

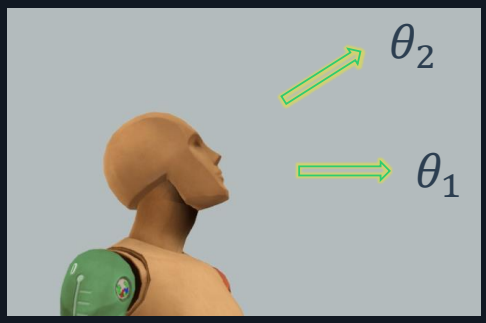
左側の状態では目の前にある対象物に注目しており、
首のジョイントが水平方向を向く角度になっています。

注目する対象物が上のほうの位置に切り替わったときには、
これにつれて首のジョイントは上を向く角度になります。

アニメーションにおける補間とは、この2つの状態の間をなめらかにつなぐ処理になります。

補間によるなめらかな角度変化の実現

- 2つの角度 θ_1, θ_2 をなめらかに補間する
  - 注視位置1に対する首の角度 θ_1
  - 注視位置2に対する首の角度 θ_2



実際にアニメーションにおける補間処理を行うと、
スライドで表示している動画のように、2つの対象物を注目する姿勢を
なめらかに切り替えるような動きが得られます。

ここで、首のジョイントの2つの状態における角度に記号を割り当てると、
1番目の、水平方向を注目している首の角度を θ1,
2番目の、上の方を注目している首の角度をθ2 として、
アニメーションにおける補間処理は、この2つの角度 θ1とθ2の間をなめらかに補間する処理だと言えます。

スライドで表示している動画は、2つの角度 θ1とθ2の間をなめらかに補間した結果を首のジョイントに適用したものとなっています。

Look-at IK

FINAL FANTASY XVI, SQUARE ENIX, 2023

首の角度を上下左右に変化させ、
注視対象をなめらかに追従する

CEDEC2024 Computer Entertainment Developers Conference

SQUARE ENIX

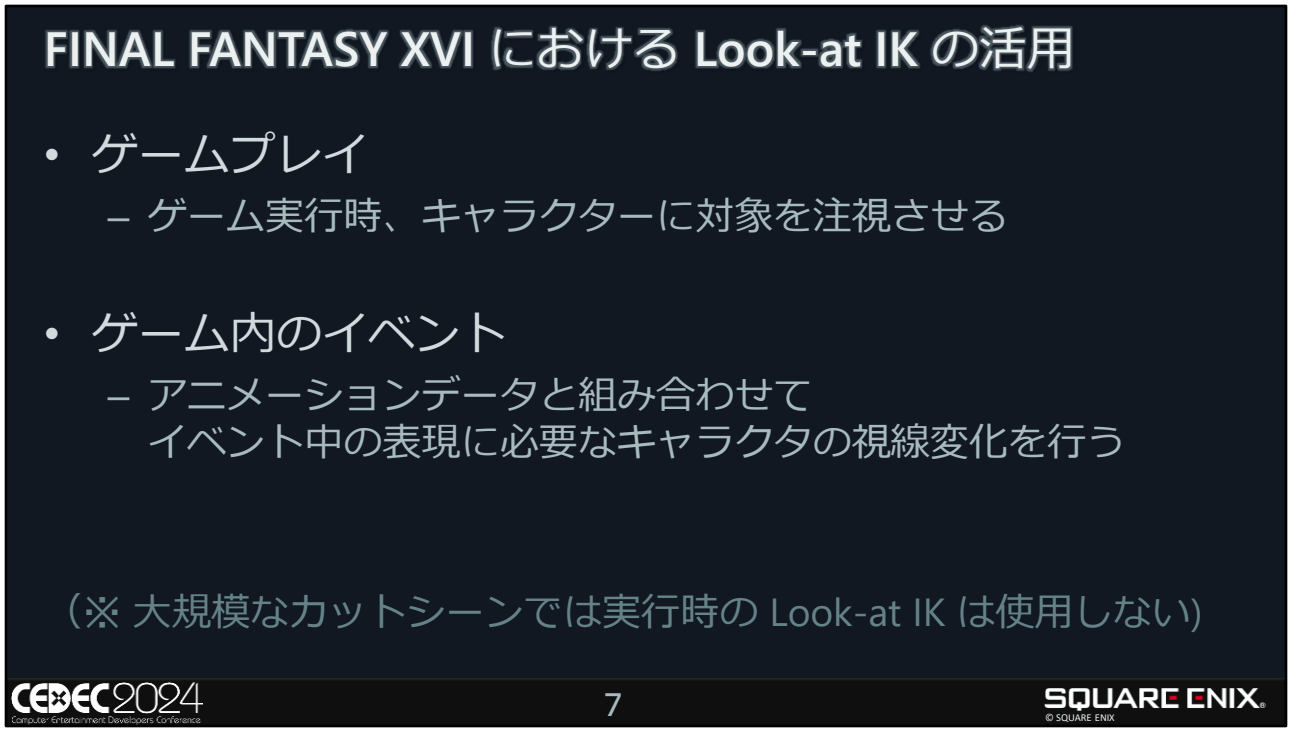
このような補間処理は、いわゆる Look-at IK 機能の実装に利用することができます。

この発表で説明するフィードバック制御によるアニメーション補間は、
昨年発売した FINAL FANTASY XVI において、
キャラクタの視線方向を変化させる Look-at IK の実装に利用されています。

このスライドでは、FF16 に登場するボスキャラクターが、
プレイヤーキャラクターを注視するようすを表しています。

プレイヤーキャラクタがボスキャラクタのまわりを移動すると、
首の角度を上下左右に大きく変化させて、
プレイヤーの方向に視線が追従していることがわかると思います。

中段の図はプレイヤーがボスの右側に移動したとき、
下段の図はプレイヤーがボスの後ろ側に移動したときのようすで、
それぞれ大きく首の角度が変わっています。



このように、FF16 ではプレイヤーキャラクタを操作できるゲームプレイの最中に、登場するキャラクタの注視する方向に変える表現を行うために Look-at IK を利用しています。

またこれに加えて、ゲーム内のイベントでも Look-at IK を活用しています。

イベントとは、ゲームのストーリーの進行に応じて発生する状態で、プレイヤーはキャラクタが操作できない状態で、ストーリーに沿ったキャラクタどうしの会話などが行われる状態のことです。

このようなイベントを作成するときには、キャラクタのアニメーションに対して Look-at IK を適用し、視線が変化する表現を行います。

アニメーションのデータについては、そのイベントに専用のものを用意することもありますが、できるだけ汎用的に再利用できるアニメーションを組み合わせることでイベントの生産性を高めることができます。

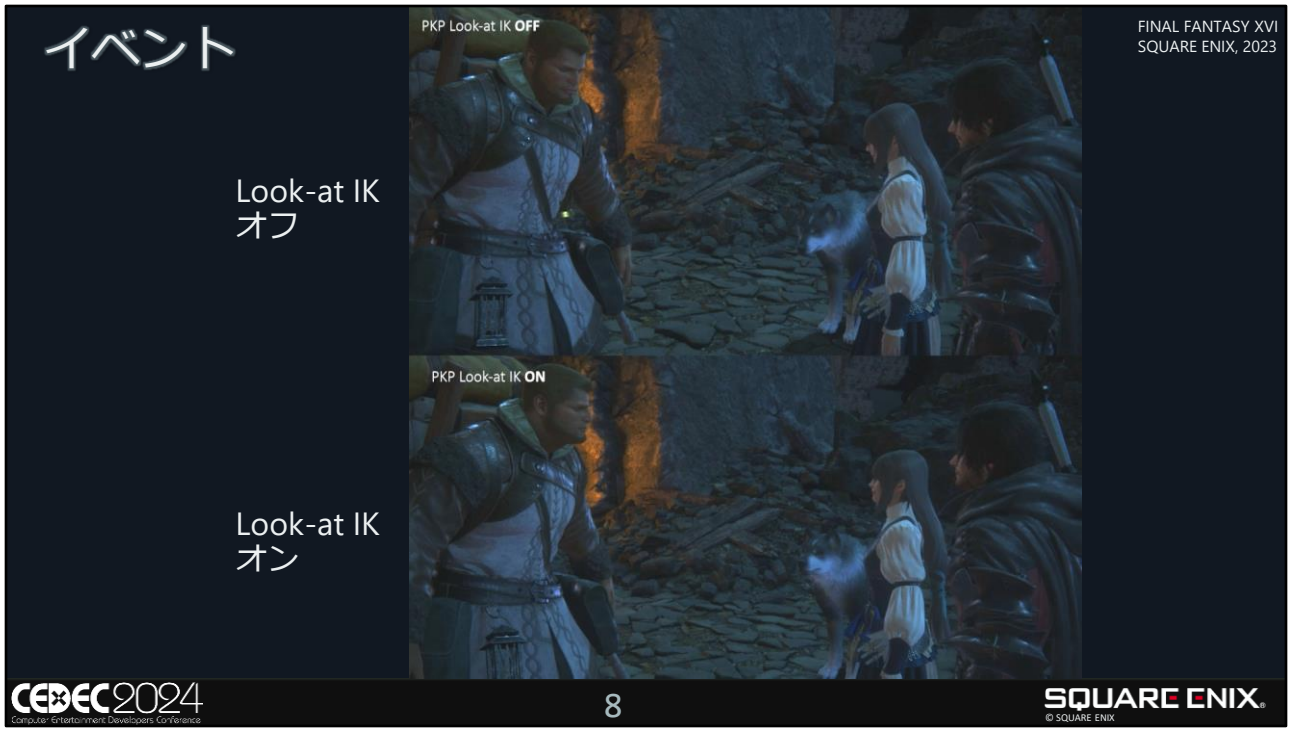
イベントにおいて Look-at IK を利用すると、一つのアニメーションデータを幅広いケースに対応させることができるので、イベントの生産性がより高くなることが期待できます。

なお、ストーリーが大きく進行するときには、イベントではなくカットシーンの再生が

行われることがあります。

このようなカットシーンでは、キャラクタのアニメーションはそのカットシーン専用のデータが用意されるので、Look-at IK が使われることはありません。

次のスライドでは、Look-at IK を活用して作られたFF16のイベントの一つを見ていきます。



この動画は、FF16のイベントで使用されている Look-at IK の効果を分かりやすく見るために、IK をオンにしたときとオフにしたときを比較した動画です。

下の動画は IK をオンにした状態の動画で、IK の効果で背の高さが異なるキャラクタどうしが自然に視線を合わせる表現が行われていることがわかります。

また、会話の区切りで IK がオフになったときには視線方向がなめらかに元の方向にもどり、自然に次のアニメーションにつながっています。

このように、Look-at IK がイベントで活用されている様子がごらんいただけたと思います。

次のスライドからは、こうした IK の効果を実装するために、フィードバック制御の手法をどのようにして適用するか、ということについて詳しく説明していきます。

フィードバック制御

- 前進する船の向きをなめらかに変えたいとき、船の舵をどのように操作すればよいか?

まず フィードバック制御、とはそもそもどういったものなのかについて説明します。

具体的な制御の例題としては、大きな船が前に進んでいるとき、その向きを変えたいといった場合があります。

このような船の向きを変えるためには、船の舵を操作する必要があります。

フィードバック制御では、船の舵の操作と、船の向きの変化の関係を数学的に定式化します。

このような定式化によって、舵の操作をどのように行えばよいかを決めることができます。

舵によって船の向きを変えるケースを定式化すると、次のスライドのようになります。

制御の定式化

- 制御対象
  - 現在の方向 θ
  - 目標の方向 θ_g

- 方向が等しくなる ($\theta = \theta_g$) ように舵輪を回す
  - 舵輪をどこまで回してもよいか?
  - どれぐらいの速さで回せばよいか?

制御の対象となるのは船が向いている方向 θ です。これに対して、目標とする方向 θg を定めます。

制御の目標は、θ = θg となるように舵を回すことです。

注意すべき点は、舵輪の角度が船の角度に直結していないということです。

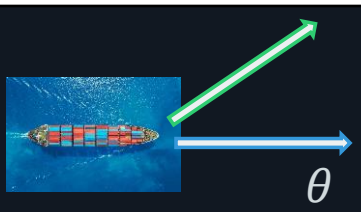
もし舵を反時計回り・左に回したら、遅れて船の向きがだんだん左に変わります。ただし、船の向きが左に曲がっていくのが思ったより遅いからといって、舵をたくさん左に切ればよいというものでもありません。

左に切りすぎてしまった場合、目標の角度を行き過ぎてしまい、右に切りなおさないといけない場合がでてきます。

このように、舵の角度と船の角度が直結していないような場合に、舵をどこまで回すか、また、どれぐらいの速さで回すかという問題になります。こうした問題は、自動制御という分野で研究されています。

古典制御理論

- PD 制御 (1自由度)
  フィードバック制御の手法の一つ



  - 制御対象　　物体の位置・角度
  - 位置・角度　　慣性 (重さ) による運動に従う
  - 制御の方法　　位置・角度を直接操作することはできず、外部から力を加えて間接的に変化させる

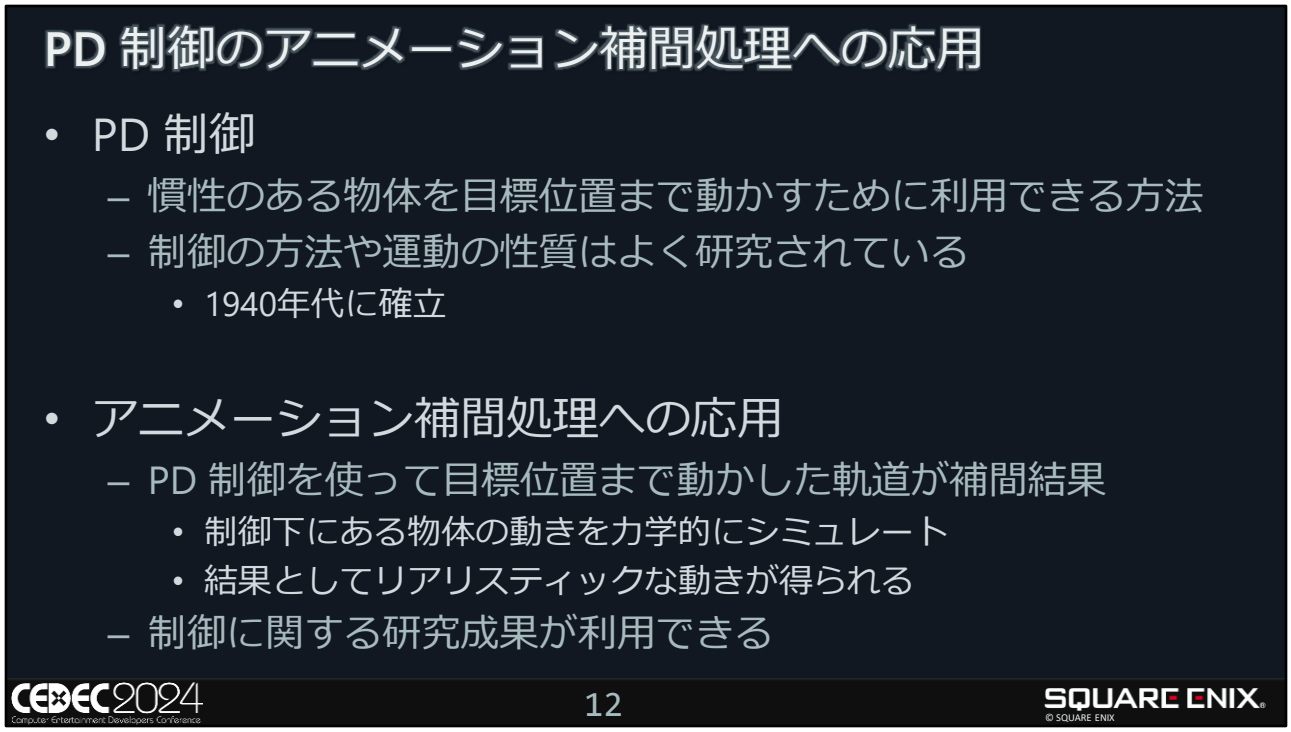
この講演で説明する PD 制御は、フィードバック制御の手法の一つです。自動制御の分野の中では古典制御理論という分類になります。

船の舵を切る問題の場合には、制御の対象は船の角度θ一つだけなので一自由度の制御となります。これはアニメーションの補間における定式化でも同じです。

制御対象となる一自由度の位置や角度は、慣性や重さによって決まる運動に従って変化します。運動を変化させるためには、物体の外部から力を加える必要があります。

ただし、位置や角度については直接値を変化させることはできず、力を加えることによって物体の速度を変化させ、その結果として位置を変化させるという、間接的な方法を取る必要があります。

またこのような制御手法は、物体の位置の制御だけではなく、製鉄所で鉄を溶かすための高炉・窯の温度制御などにも使われています。



以上のように、PD制御は慣性や重さのある物体に力を加えて、目標の位置まで動かすために利用できます。

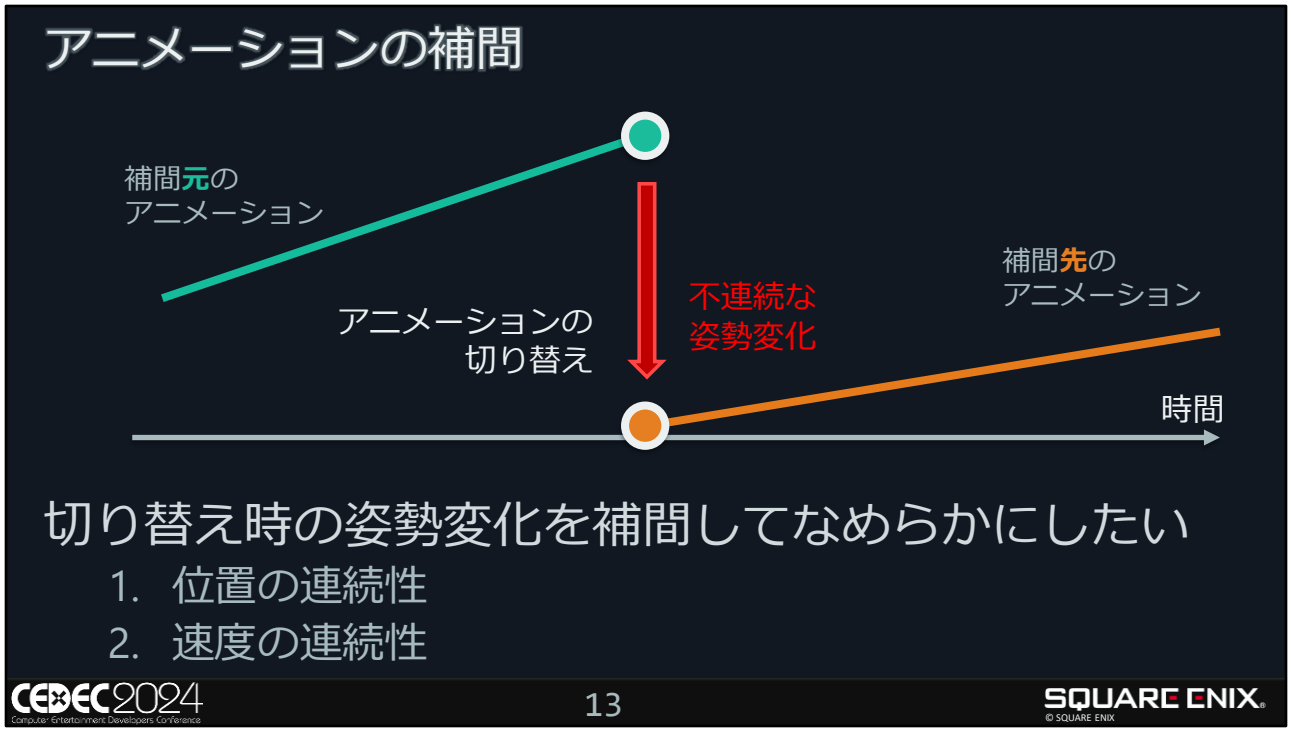
この制御についてはよく研究されており、1940年代にその理論的な基礎が確立されています。そのため、力を加える制御の方法や、結果として得られる運動の性質がよくわかっているといえます。

PD制御を使って物体を目標位置まで動かす方法は、アニメーションの補間処理に応用することができます。PD制御によって物体を動かした位置の変化、得られる軌道が、補間の結果となります。

このような補間を行う場合は、PD制御の対象となっている物体の動きを力学的にシミュレートすることになります。

重さのある物体の物理シミュレーションが補間処理のアルゴリズムとなるので、結果としてリアリスティックでなめらかな動きが得られるという特徴があります。

また、PD制御の性質はこれまでにも広く研究し活用されているので、これらの研究成果を利用することができます。



このスライドからは、アニメーションの補間について説明します。

アニメーションの補間は、異なるアニメーションを切り替えて再生するときに必要になります。

スライドでは、左側の緑色のアニメーションから、右側のオレンジ色のアニメーションへの切り替えが生じるようすを表しています。

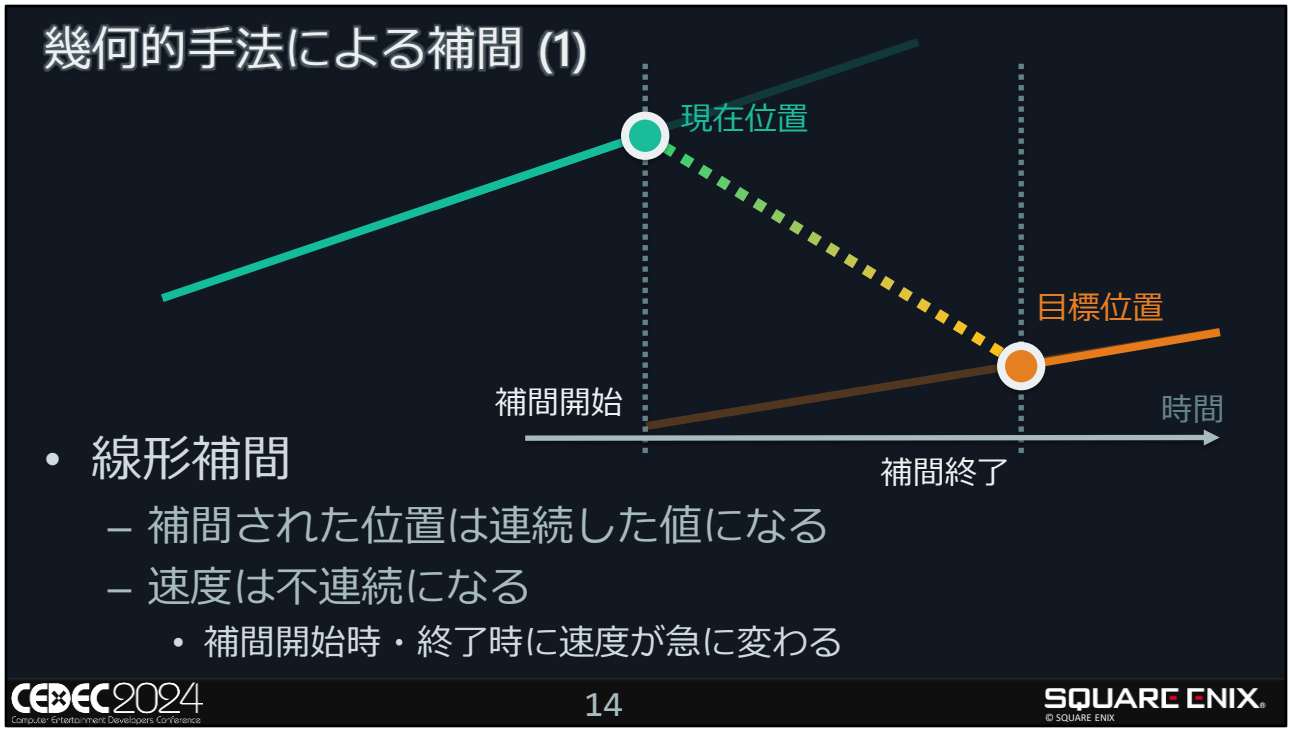
横軸は時間で、縦軸はあるジョイントの角度を表しているものと考えてください。

このとき、ふたつのアニメーションは、切り替えが起こる瞬間に角度が不連続になっています。

そのため、アニメーションを切り替えた瞬間、キャラクタの姿勢も不連続に変化してパッとポーズが飛んでしまうように見えます。

このような不連続な姿勢の変化を防ぐために、アニメーションを切り替えた時に補間を行い、姿勢の変化をなめらかにする処理が必要になります。

なめらかな姿勢変化に必要な性質とは、まず位置や角度が連続していること、そして、速度が連続していることです。



一番単純なアニメーションの補間手法は、線形補間です。

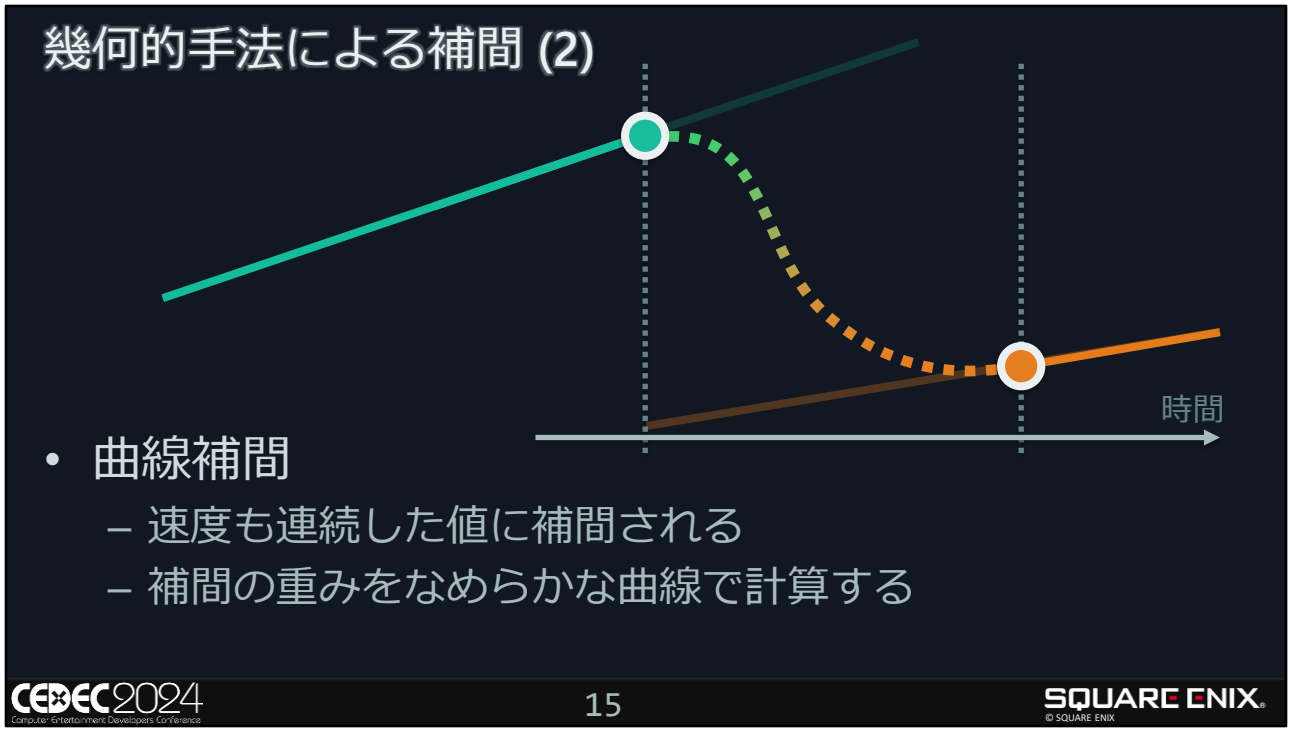
線形補間では、補間を行う開始点から終了点まで、補間の重みを線形に変化させます。

このとき、スライドに示したように、補間された位置や角度が不連続に変化し、キャラクタの見た目がパッと飛ぶことはなくなります。

ただし、補間された速度は連続にはなりません。

スライドのグラフで言えば、補間が開始した瞬間や補間が終了した瞬間で速度が急に変わり、不連続になります。

このとき、速度が不連続な瞬間では、キャラクタの見た目として、急に別の方向に動き出すような見た目になり、不自然な感じになります。



幾何的な補間手法でも、曲線による補間を用いることで、位置と速度の両方について連続した補間結果を得ることができます。

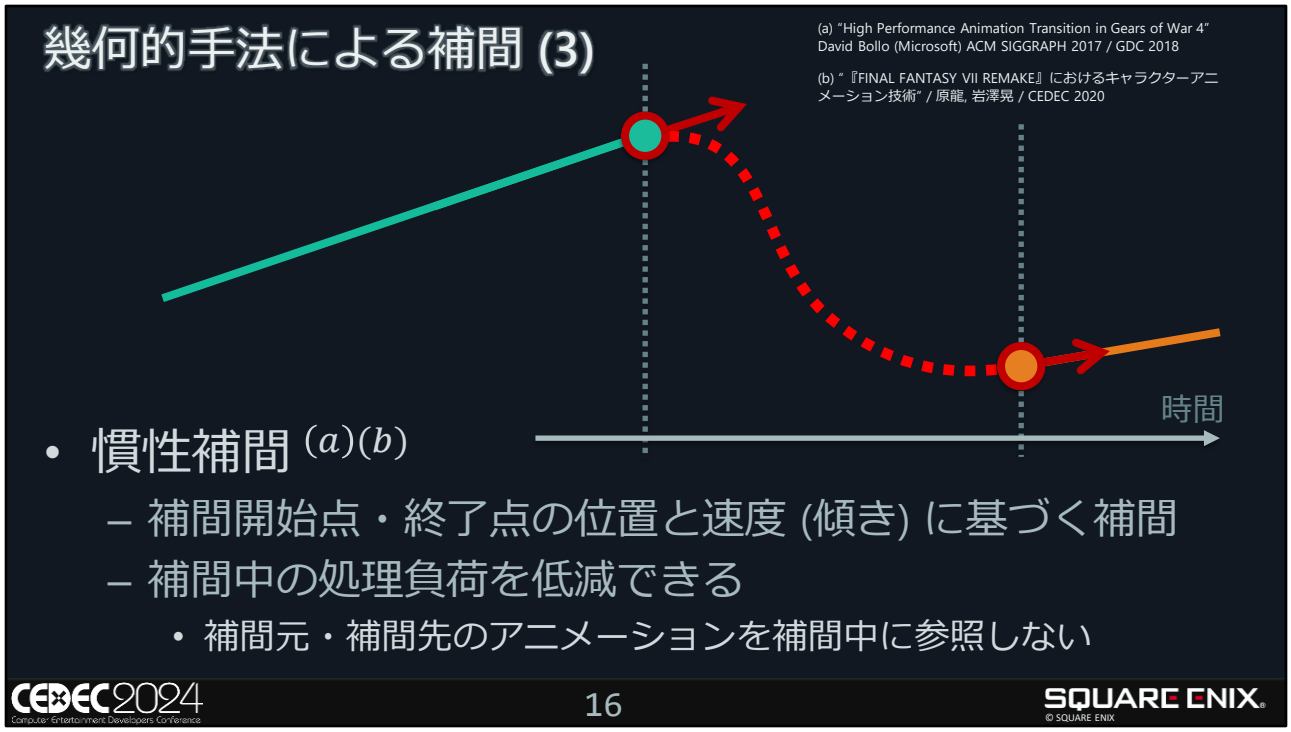
補間を行う時間の区間で、曲線の方程式を利用して補間の重みをなめらかに変化させることで、補間の結果の位置と速度の両方が連続した値になります。

スライドのグラフでは、補間の開始から終了まで、補間された結果がなめらかな曲線となっていることがわかると思います。

このとき、キャラクタの動きにも位置や速度がぱっと変化することはなく、補間結果としてなめらかな動きが得られます。

ただし、前のスライドの線形補間も同じですが、二つのアニメーションの重みづけを変化させて補間を行う方法では、補間を行っている間、補間元と補間先の両方のアニメーションの再生をつづけなければならないという特徴があります。

この特徴により、補間によってアニメーションの処理負荷が増えてしまうという問題が生じます。



補間処理を行っている間にアニメーション再生の負荷が増える問題に対応できる手法として、慣性補間という手法が提案されています。

この手法は、アニメーションの補間を行うとき、二つのアニメーションの重みづけを行いません。

その代わり、補間の開始点と終了点の位置と速度を保存しておき、この2つの点の間を位置と速度に基づいてなめらかに補間します。

こうすることで、元になるアニメーションを再生しつづけなくても補間の処理を可能にしています。

この手法は実際のゲームタイトルでも使われており、最近の Unreal Engine でもアニメーションシステムの機能として組み込まれています。

幾何的手法による補間の性質

- 開始点・終了点の間で値をなめらかにつなげる
- 補間が終わるまでの時間長さを指定しておく
  - 補間開始点から終了点までの時間
- 補間の時間長さはどのように決めればよいか？
  ⇒ 補間対象のアニメーションの性質に応じて
  長さを調節する

以上で見てきたように、幾何的な手法によるアニメーションの補間では、補間の開始点から終了点まで、対象となる値をなめらかにつなぐ処理を行います。

このとき、補間が終わるまでの時間の長さは、補間処理のパラメータの一つとしてあらかじめ指定しておきます。

そのため、もし補間結果が適切でない場合には、補間の時間長さについても調整が必要になる場合があります。

補間の時間長さをどれぐらいにするのが適切なのかについては、補間処理の対象になるアニメーションによって異なっており、それぞれのアニメーションについて長さを調節することになります。

【比較】幾何的な補間 vs. フィードバック制御による補間

- 幾何的な補間
  - 指定した時間長さでなめらかに補間できる
- フィードバック制御 (PD制御) による補間
  - 必ずしも一定時間内に補間が終わるとは限らない
    - (長所) 状況に応じて補間時間が自動的に伸縮する
    - (短所) 補間時間の厳密な長さを保証できない
  - 力学的なモデルによるなめらかな補間結果が得られる
    - (長所) 補間の途中でも簡単に目標位置を変えることができる
    - (短所) フレームレートが異なると補間結果が完全には一致しない
    - (短所) 時間軸を遡る補間は難しい

以上のスライドで見てきたように、幾何的な補間手法を用いた場合には、補間の時間長さを適切に指定しておくことで、アニメーションの間をなめらかに補間することが可能になっています。

(1) 一方で、PD制御を利用したアニメーションの補間では、いくつかの点で幾何的な補間とは性質がことなっています。

まず、PD制御を利用して補間を行うと、一定の時間で補間が終わるとは限らないという特徴があります。

もちろん、PD制御を調節するパラメータによって、補間に要する時間を長くしたり短くしたりといった調整は可能ですし、補間時間のおおまかな予測もできます。

ただし、補間が開始した瞬間の速度が大きい場合や、補間先の値が変化する場合などには、補間時間はこれに応じて伸びたり縮んだりします。

これは、PD制御を利用したときに、状況に応じて補間時間が自動的に調整されるような性質をもっていることが原因です。

この性質自体はPD制御の長所でもありますが、補間の時間を厳密に決めておきたい用途には向いていないとも言えます。

(2) 次に、PD制御で補間を行う場合には、力学的なモデルを用いて補間結果を計算しています。

そのため、補間中にダイナミックに補間先の目標位置が変化する場合でも、特に工夫せずになめらかな補間結果を得ることができます。この性質は、注目する位置が常に変化する可能性がある Look-at IK に応用するためには大変便利な性質です。

その一方で、補間処理の結果は力学的なシミュレーションの結果から得られるので、同じ補間を行ってもフレームレートが異なる場合には、補間結果が完全に一致することはありません。

これは 30FPS と 60FPS とで補間結果の値が2倍異なるという意味ではなくて、力学的なモデル化による時間発展の系を計算するという性質から、どうしても結果に誤差が生じるということです。

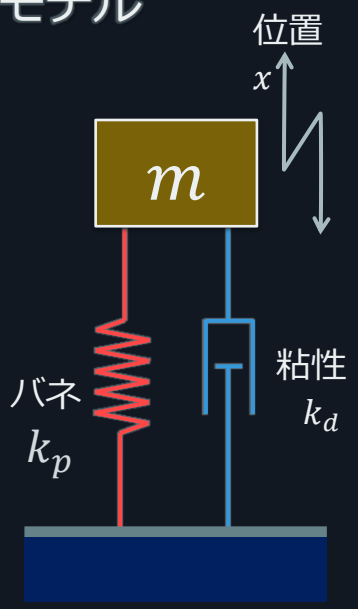
したがって、ネットワークゲームなどで、別々の計算機の上でアニメーションが厳密に一致しなければならないといった場合には問題が生じます。

また、同様の理由から、時間軸をさかのぼるような補間を行うことは難しくなります。

以上で挙げたPD制御の長所と短所については、次のスライドからそれぞれ詳しく説明していきます。

フィードバック制御 (PD制御) の力学的モデル

1. 制御対象の物体 （重さ m）
   - 物体の位置 x を目標に追従させる
2. バネ （強さ k_p）
   - 目標位置とのズレをなくす方向に働く
   - k_p を大きくすると素早く追従する
3. 粘性抵抗 （強さ k_d）
   - 物体の移動速度と反対方向に働く
   - k_d を大きくすると追従運動が安定する



このスライドの図は、PD 制御の力学的なモデルを表しています。

制御の対象となるのは重さのある物体で、物体の位置を目標位置に近づけることが制御の目的です。

(1) 物体は上下に動くことができて、物体の位置は記号 x で表します。
また、物体の重さを表す記号を m とします。

(2) この物体は、動かない壁に対してバネと粘性抵抗でつながれています。

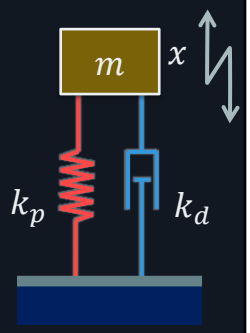
バネは物体の位置と目標位置とのズレをなくすように働きます。バネの強さを表すばね定数の記号を kp で表し、kp が大きいほどバネの力が強くなり、素早く目標に追従します。

(3) もう一つの要素は、粘性抵抗です。

粘性抵抗は、物体の速度に比例して働く力です。物体の速度が速ければ速いほど、移動に逆らう力を働かせます。粘性抵抗の強さを kd で表し、kd が大きいほど速度に抵抗する力が強くはたらき、目標に追従する運動が安定します。

PD制御の運動方程式

- $m\ddot{x} = F$
  $= k_p(x_g - x) - k_d\dot{x}$
  - 物体の加速度 $\ddot{x}$ は、物体に加える力 F に比例する
  - 目標位置 x_g と現在位置 x のズレが大きいほど
    目標位置に引き寄せるバネの力をより大きく働かせる
  - 物体の移動速度 $\dot{x}$ が大きいほど、
    速度を減衰させる粘性抵抗力をより大きく働かせる



PD 制御の力学的モデルについて、運動方程式を立てるとスライドの式のようになります。

(1) まず、制御対象の物体の加速度 x.. は、物体に加える力 F に比例します。次に、物体に加える力 F は、バネの力と粘性抵抗の力が合わさった力になります。

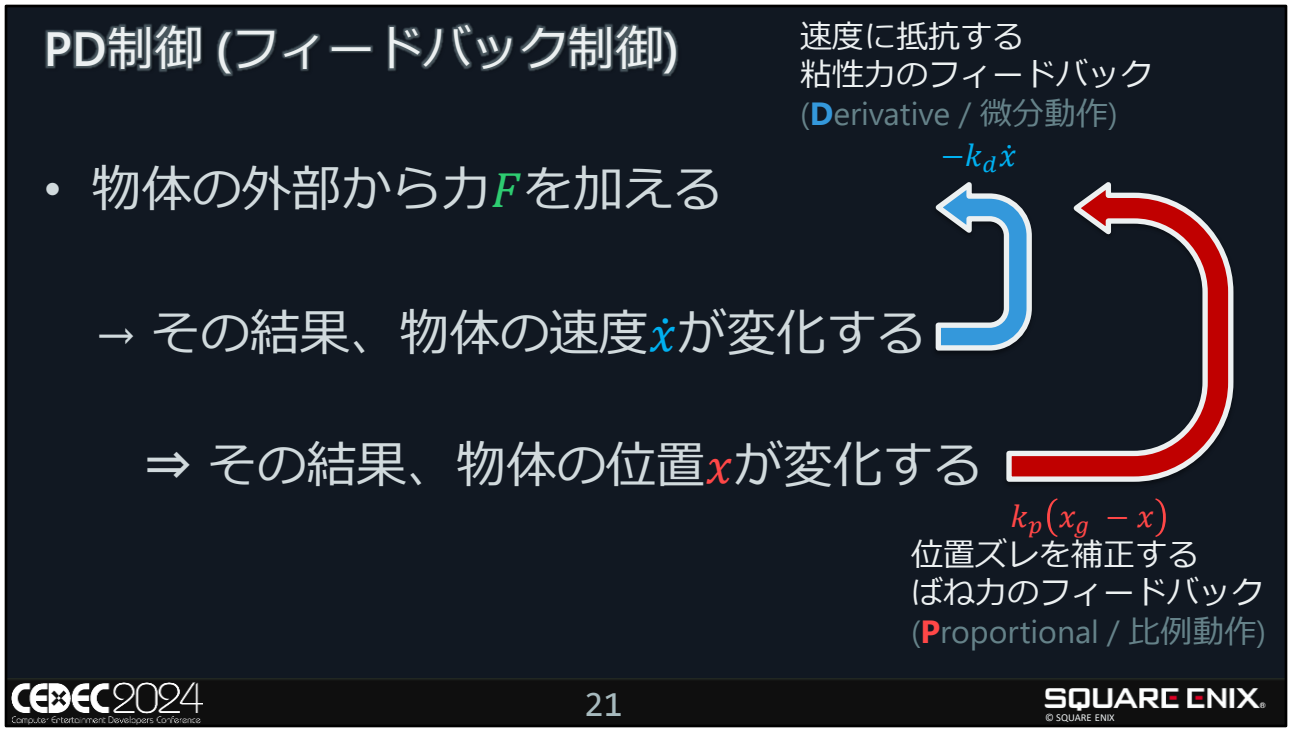
(2) バネの力は、目標位置 xg と現在位置 x とのズレに比例して大きくなります。

比例定数は kp で、kp が大きいほど目標位置に引き寄せるバネの力が大きく働きます。

(3) 粘性抵抗の力は、物体の速度に比例して大きくなります。

比例定数は kd で、粘性抵抗は物体の速度と反対向きに働くので式としては -kd x. と表せます。

このように、バネkpの力と粘性抵抗kd に力によって、物体に加わる力 F を表すことができます。



ここまでで PD 制御の力学モデルについて見てきました。

(1) 物体の運動について少しおさらいをすると、運動を決めるのは物体に外部から働く F です。

(2) 外部から力が働くと、その結果として物体の速度 x. が変化します。

(3) 物体の速度が変化すると、結果として物体の位置 x が変化します。

(4) PD 制御では、物体の位置に応じて、その位置のズレを補正するバネの力を働かせます。バネの力は位置のズレに比例して大きくなるので、これを比例動作といいます。英語でいうと比例は proportional になって、PD 制御の P はここから来ています。

働かせるバネの力は、力学モデルの運動を決める外部からの力 F の一部としてフィードバックされることになります。フィードバック制御という呼び方はこのような仕組みを表しています。

(5) また同様に、物体の速度に応じて、速度に抵抗する粘性抵抗の力を働かせます。粘性抵抗は速度に比例して大きくなります。速度は位置の微分なので、これを微分動作といいます。英語でいうと微分は derivative で、PD 制御の D に相当します。

バネの力と同様に、微分動作の力は外部からの力にフィードバックされます。これが PD 制御で力のフィードバックが行われるの全体像になります。

ゲームプログラムにおけるPD制御

- ゲームではPD制御の利用は簡単
  - 重さ m, 力の強さ k_p, k_d を好き勝手に決められる

    ⇔ 現実世界では重さやバネの強さ等は実物に依存している

- ゲームプログラムでの実装
  - 重さは $m = 1$ に固定して計算式から消去
  - 調整パラメータは2つ: (1) バネ k_p (2) 粘性抵抗 k_d
    - PD制御による補間結果の動きを調整するパラメータとなる

このような PD 制御の方法をゲーム中で利用するのは特に難しいことではありません。

現実世界でも PD 制御は広く利用されていますが、実体のあるものを PD 制御で動かす場合、重さやバネの強さは実物によって決まります。

そのため、たとえばバネの強さは、実際に手に入るバネの性能に依存しているので、好き勝手な強さのバネを使うことはできません。

一方で、ゲームで PD 制御を利用する場合にはパラメータに値を代入するだけなので、必要に応じて値を自由に変えることができます。

また、ゲームでは物体の位置や速度は変数として保持しているので、いつでも正確な値を知ることができます。

現実世界の制御対象では、位置や速度の計測にノイズや遅れがあったりしますが、ゲームではこうした制約はありません。

(2) 実際に PD 制御をゲームに実装するときには、値を自由に決めることができるという性質を利用して、物体の重さ m については 1 に固定しておくとよいです。

m = 1 にしておくことで、計算式から記号 m を消すことができます。

その場合、PD 制御によってアニメーションの補間を行うときの調整パラメータは、バネ kp と粘性抵抗 kd の2つになります。

補間結果の動きはこの 2 つのパラメータによって決まることになります。

実装例：

```
struct PD {
  float x, v;     // 現在位置xと現在速度ẋ（現在の状態）
  float kp, kd;   // バネと粘性抵抗（調整パラメータ）

  /* mẍ = kp(xg − x) − kdẋ, m = 1 */
  void update(float dt,                  // 時間刻み
              float xg) {                // 目標位置
    float F = kp * (xg - x) - kd * v;    // (1) mẍを計算
    v += dt * F;                         // (2) 速度を更新
    x += dt * v;                         // (3) 位置を更新
  }
};
```


PD 制御の力学モデルをプログラムとして実装すると、このスライドのようになります。

全体としては PD という名前の構造体になっていて、float 型のメンバ変数が4つあります。

最初の2つのメンバ変数は制御対象の物体の状態を表していて、x が現在の位置、v が現在の速度を表します。

次の2つのメンバ変数は、PD 制御の調整パラメータです。kp はバネ定数、kd が粘性抵抗の定数です。

現在の状態を更新するときは、メンバ関数の update を呼び出します。

update の引数は時間刻み dt と目標位置 xg です。

update に与えられた引数と、調整パラメータのメンバ変数から物体に働く力 F を計算できます。プログラムでは (1) の部分です。

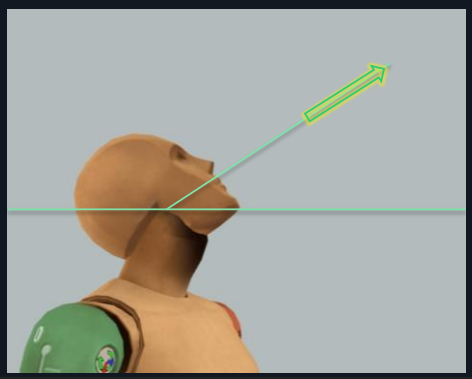
次に、(2) で力 F にしたがって速度を更新します。

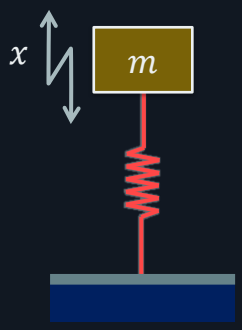
最後に、(3) で更新された速度によって位置の更新を行います。

これで PD 制御の実装が完成です。

バネ k_p だけを働かせた場合 (粘性抵抗なし)

→ 目標位置 x_g で止まることができず、
目標位置 x_g の前後を行ったり来たりする





次に、PD制御の力学モデルが実際にどのような運動を行うかを見ていきます。

まずはじめに、もしPD制御のDの部分をなくしたらどうなるかを考えます。

D の部分は粘性抵抗で、これをなくすとバネだけが働いていることになります。

このとき、物体はバネの力によって目標位置の方向に引き寄せられます。

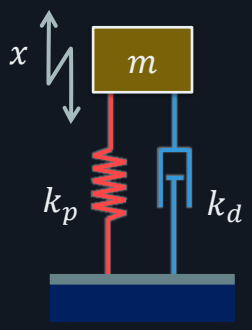
ただし、粘性抵抗による減衰がないので、物体が止まることができません。

そうすると、物体は目標位置を中心として、その前後を行ったり来たりする往復運動を続けることになります。

これがPD制御で粘性抵抗が必要になる理由です。

バネ k_p と粘性抵抗 k_d の併用 (PD制御)

- 物体の位置 x は目標位置 x_g に近づく
- バネ k_p と粘性抵抗 k_d の
  強さに応じて近づき方が変わる
  1. 臨界減衰
  2. 不足減衰
  3. 過減衰



先程のスライドで説明したように、物体を目標位置に近づけるためには、バネと粘性抵抗の両方の力が必要になります。

バネは目標とのズレを補正する力です。粘性抵抗は速すぎる速度を抑える力です。

このバネと粘性抵抗による力を加えて物体の位置を制御すると、最終的に物体は目標位置に近づきます。ただし、その近づき方は、バネの力と粘性抵抗の力の強さに応じて、近づき方が変化します。

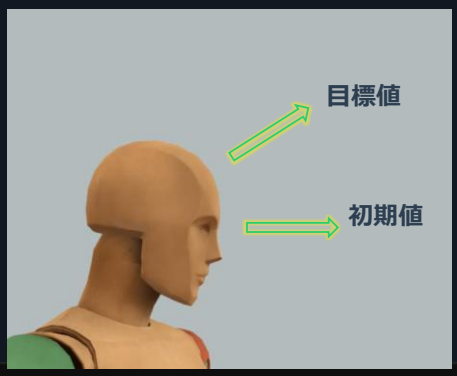
古典制御理論の研究によって、PD制御における目標位置への近づき方は3種類に分類されています。

その3種類は、臨界減衰、不足減衰、過減衰と呼ばれています。

次のスライドからは3種類の近づき方についてもう少し詳しく見ていきます。

目標への近づき方（1） 臨界減衰

- $k_d = 2\sqrt{k_p}$ のときを臨界減衰という
  - 目標値を通り過ぎず、到達するまでの時間が短い



設定例

- $k_p = 40$
- $k_d = 12.5$

まず近づき方の1番目として、臨界減衰について見ていきます。

臨界減衰は別の呼び方として臨界制動と呼ばれることもありますが、このスライドでは臨界減衰と呼びます。

PD制御の性質が臨界減衰となる条件は、kd = 2 sqrt(kp) となることです。

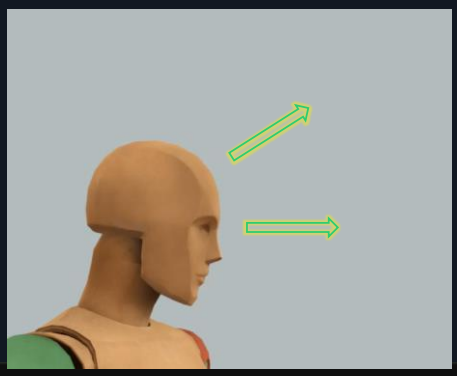
このとき、制御対象の値は、目標を通り過ぎることはなく、目標まで短い時間で到達することができます。

スライドの動画では、首のジョイントの角度をPD制御の対象として、最初に水平方向を見ていた角度が、上の方を見上げる目標値まで臨界減衰で近づく様子をあらわしています。

目標の角度を通り過ぎることなく、目標の角度の近くに到達していることがごらんいただけると思います。

目標への近づき方（2）不足減衰

- 臨界減衰よりも粘性抵抗 k_d を減少させたとき
  - 目標値を少し通り過ぎてから戻ってくる
  - $k_d = \sqrt{2k_p}$ は応答性が高く、よく使われる設定



設定例

- $k_p = 40$
- $k_d = 8$

次に、不足減衰について見ていきます。

先程のスライドでは臨界減衰について説明しましたが、不足減衰と臨界減衰を比べると、臨界減衰よりも粘性抵抗を小さくしたものが不足減衰となります。

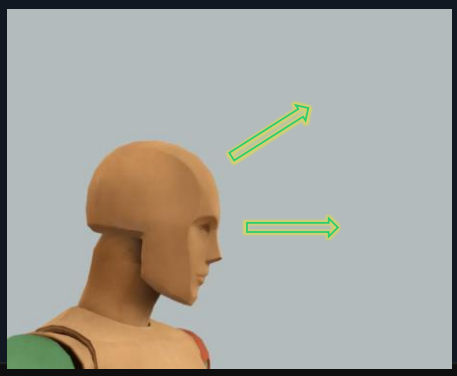
不足減衰では物体の速度に対する抵抗力が弱くなっているため、目標値に近づいたときにちょうど目標で止まることはできずに、少し通り過ぎてから戻って来る動きをします。

スライドの動画は、このような不足減衰によるPD制御の一例です。上を向く目標に対して、すこし通り過ぎてから戻っているのがわかると思います。

このような不足減衰の中でも、kd = sqrt(2kp) は目標に対する応答性能が高く、PD制御の設定としてよく使われる値です。スライドの動画もこの設定を利用しています。

目標への近づき方（3）過減衰

- 臨界減衰よりも粘性抵抗 k_d を増加させたとき
  - なかなか目標にたどり着かない
    (臨界減衰より遅い)



設定例

- $k_p = 40$
- $k_d = 15$

3番目は、過減衰という近づき方です。

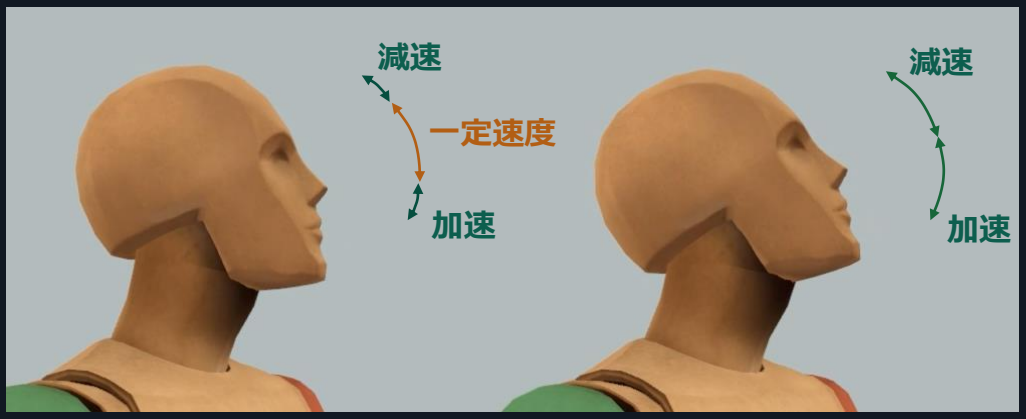
過減衰は、臨界減衰よりも粘性抵抗を大きく設定したときの近づき方になります。

臨界減衰と比較すると、同じ速度に対して粘性抵抗による力がより強く働くことになるので、目標値への近づき方は臨界減衰よりも遅くなり、なかなか目標に近づかなくなります。

動きの安定感を増したいときには、このように粘性抵抗を大きめに設定することができます。

速度の制限

- 補間結果における速度を一定以下に抑える



また、PD 制御を利用したときの目標への近づき方を調整するとき、ゲームに特有の実装として、速度の制限を加えることができます。

速度の制限は、バネや粘性抵抗によって速度を変化させたあと、その速度が一定以下になるように値を直接修正する実装です。

このスライドの動画は、PD 制御を利用して上を向かせるときに、首の角速度が一定以下になるように制約を加えた様子をあらわしています。

このような効果は、たとえば補間された動きに対して、ロボットダンスのような性質、一定速度で動く性質を加えたいときに使うことができます。

速度制限の実装

```
void PD::update(float dt, float xg,
                float vMax)          // 最大速度
{
  /* mẍ = k_p(x_g − x) − k_d ẋ,  m = 1 */
  float F = kp * (xg - x) - kd * v;// (1) mẍを計算

  v += dt * F;                       // (2) 速度を更新
  v = std::clamp(v, -vMax, vMax);    // (2+) 速度を制限

  x += dt * v;                       // (3) 位置を更新
}
```


PD制御のプログラムに対して、速度の制限を加える実装は非常に簡単です。

PD制御の update 関数は、力を計算して速度を更新し、更新した速度で位置を更新するという順番になります。

ここで、速度の最大値を与えるために、update 関数の引数に最大速度を表す vMax を指定できるようにします。

Update 関数の2番目の部分で速度を更新したあと、一定の範囲に速度が収まるように vMax で値を clamp する処理を足すことで、速度の制限を行うことができます。

PD制御による補間の性質

1. 力学的モデルによるなめらかな補間結果が得られる

2. 補間に要する時間は動的に変化する
   - 補間が始まった瞬間の状態に依存して補間時間が変化する
   - 調整パラメータでおおよその補間時間が決まる

3. 目標位置に限りなく近づいて行くが、
   数学的には決して目標位置に一致しない (漸近する)

これまでのスライドで、PD制御によってアニメーションの補間を行った結果がどのような性質を持つかということがわかってきたと思います。

1番目の性質としては、PD制御は力学的モデルに基づいているので、補間の結果としてなめらかなアニメーションが得られるということがあります。

2番目の性質としては、PD制御の力学的モデルの性質から、補間に必要となる時間、つまり、目標位置に近づくために必要な時間が初期状態に応じて変化するということが挙げられます。

これは、補間が開始したときに物体がある程度の速度をはじめから持っていた場合を考えるとわかりやすいと思います。

ただし、PD制御の力学的モデルではバネと粘性抵抗によって目標に近づける力を調整しているので、PD制御による動きのおおまかな性質はこれらの調整パラメータによって決まります。特に、補間にかかる時間は主にバネの強さが大きな影響を与えています。

目標位置近くに達するまでの時間

- 臨界減衰のとき
  - おおむね $2\pi/\sqrt{k_p}$ 秒ぐらいで目標付近に達する
  - (例) $k_p = 40$ の臨界減衰では約1秒

  (ただし初期速度や粘性抵抗 k_d に依存して実際の秒数は変動する)

- PD制御の調整パラメータ
  - バネの強さ k_p を直接指定してもらうのではなく、目標付近に達するまでの**秒数**で調節できると直感的

では実際にバネの強さがどれぐらいPD制御による補間に必要な時間に影響するかについて見ていきます。

結論からいえば、臨界減衰のときにだいたい目標位置に達するまでの秒数は2pi/sqrt(kp) ぐらいということが知られています。

数字の例を上げれば、先にお見せしたスライドの動画ではどれも kp = 40 と設定していましたが、このとき約1秒ぐらい補間の時間がかかることになります。

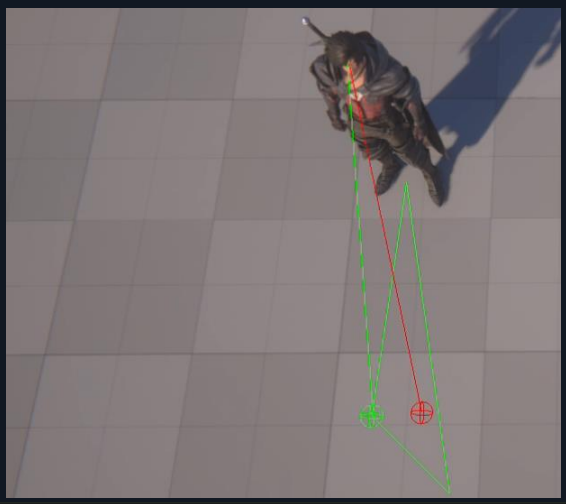
ただし、繰り返しになりますが、物体の初期速度や粘性抵抗の大きさに依存して、実際の補間の秒数は変化します。粘性抵抗を大きくすると速度が遅くなるので、秒数は伸びることになります。

このような補間の時間については、ゲームで実際に PD 制御をアニメーションの補間に利用するときは、アニメーションの担当者などが時間を調節できるようにすると思います。

そのような場合には、kp をそのまま調整可能なパラメータとして見せるよりも、このスライドで説明した補間時間をパラメータとして調節してもらったほうが直感的になります。プログラムに与える kp は秒数から計算することができるので、プログラムの内部的には kp に変換した値を覚えておくことになります。

PD制御の追従性

- 目標が移動しつづける場合は一定の遅れが生じる (定常偏差)



FINAL FANTASY XVI, SQUARE ENIX, 2023

PD制御のもう一つの特徴として、目標を追従するときの性質があります。

具体的な性質としては、もし目標が移動し続けている場合には、PD制御によってこれを追従すると、つねに一定の遅れを生じながら追従するという性質です。

このような遅れのことを定常偏差といいます。

このスライドの動画では、PD制御における定常偏差が生じているようすを示しています。

緑色の丸が目標位置を表していて、この目標位置が移動し続けているために、赤丸で表される追従結果は少し遅れて緑色の○を追いかける結果となっています。

FF16 では、Look-at IK にこのような遅れがあっても特に問題が出るような IK の使い方はしていませんでした。

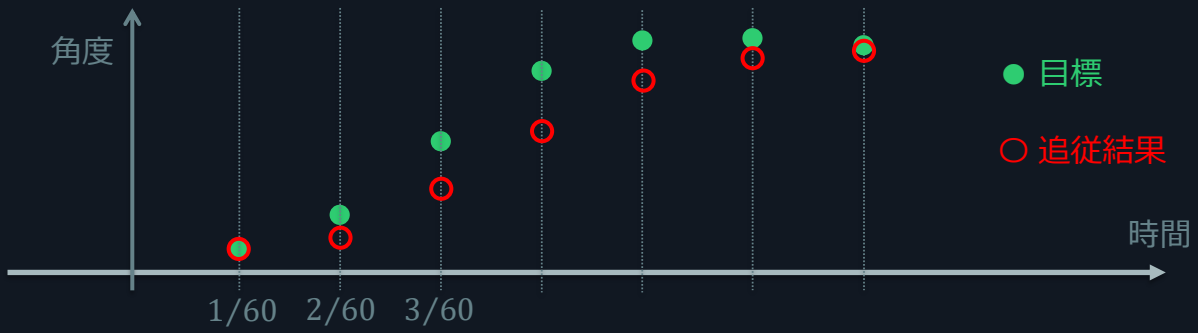
ですが、もし定常偏差を小さくする必要のあるゲームであれば、PD制御の代わりにPID制御という方法があるので、こちらを使うことを検討したほうがよいと思います。

ちなみに、PID制御のIは積分を表していて、動画のような定常偏差を解消するように働きますが、この発表では PID については扱いません。

アニメーションデータとPD制御

"Misconception of PD Control in Animation"
B. F. Allen and P. Faloutsos
Eurographics / ACM SIGGRAPH Symposium on Computer Animation 2012

- PD制御でアニメーションデータを再現できる？
  - 毎フレームの関節角度を目標として追従



⇒ 追従結果はアニメーションデータから**遅れる**

PD制御をアニメーションの補間に利用したときに遅れが出るという性質があって、これに関係する問題があるので説明します。

その問題というのは、動きのあるアニメーションデータにPD制御を適用したときの結果です。

(1) いままでのPD制御の性質から、PD制御の目標値としてアニメーションデータを流し込んだらいいのではないか、ということは考えられると思います。

つまり、アニメーションデータとして毎フレーム変化する関節角度があるとき、この角度、スライドのグラフで緑色の〇としてあらわされている値を、毎フレーム目標値として与えるということです。

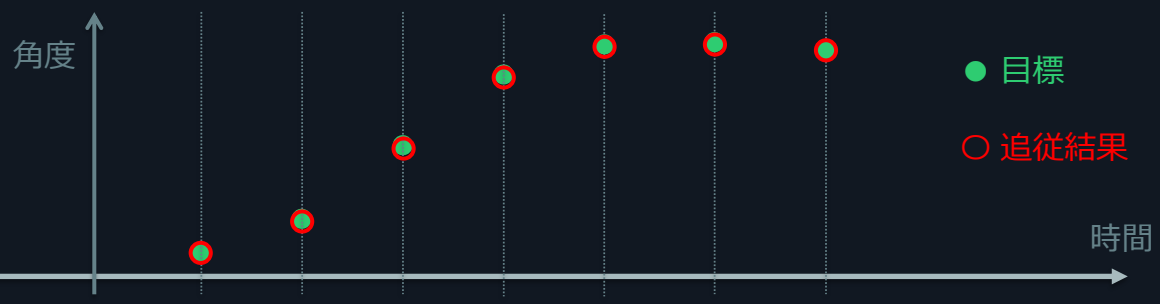
(2) このときPD制御による追従結果がどうなるかというと、前のスライドの動画でお見せしたように、一定の遅れを生じながら追従する結果になります。

スライドのグラフで見ると、赤丸で表される追従結果は、緑色の目標値から少しずつ遅れている結果になるということです。

関節角度の追従遅れをなくしたい

"Misconception of PD Control in Animation"
B. F. Allen and P. Faloutsos
Eurographics / ACM SIGGRAPH Symposium on Computer Animation 2012

- バネk_pと粘性抵抗k_dを調節して追従性を高めたら？
  - 入力のアニメーションデータは高精度で再現される
  - なめらかな補間結果を得られる利点が損なわれる



このような遅れをなくすためには、バネの強さなどのパラメータを調節することができます。

具体的には、バネを強くして追従をより速くするという方法になります。

このようにすると、変化していく目標に対しても、追従の結果がほとんどズレなくなります。

ただし、このような調整は大きな問題があります。つまり、追従性を高めることによって、PD制御による補間結果がなめらかにならないという問題が出てきます。

もともと、PD 制御をアニメーションの補間に利用した目的は、Look-at IKのように目標値がさまざまに変化するときでも、補間の結果としてなめらかな動きが得られるという性質を利用するためでした。

したがって、目標値に対して非常に高い追従性能を持たせることは、このような目的とは相反する使い方ということになります。

関節角度の PD 制御

"Misconception of PD Control in Animation"
B. F. Allen and P. Faloutsos
Eurographics / ACM SIGGRAPH Symposium on Computer Animation 2012

- 関節角度を直接の目標値とするのは悪手
- アニメーションの角度を Look-at IK で変えたい場合
  - IK による 角度変化 (オフセット) を制御目標にするとよい
    1. 実現したい角度 = アニメーションの角度 + オフセット角度
    2. オフセット角度を PD 制御における目標として追従させる
    3. 追従結果をキャラクタの姿勢に適用する

  ⇒ 元のアニメーションを忠実に再生しつつ、
  IK による姿勢変化をなめらかに加えることができる

PD制御のこのような性質から、なめらかなアニメーションの補間結果を得るためには、アニメーションデータの関節角度を直接の目標値として追従させる方法は望ましくないということがわかります。

(a) では実際に Look-at IK のように目標とする角度をなめらかに追従させるにはどうすればよいかということについて考えます。

結論から言えば、アニメーションデータの角度そのものではなく、IKによる角度の変化を目標値として使うとよいということになります。

(b) もう少し詳しく説明すると、IK で実現したい関節の角度を、アニメーションデータが持っている角度と、そこからの差分の角度に分けます。

ここでは差分の角度のことをオフセット角度と呼びます。

(c) このオフセット角度をPD制御における目標値として追従させることにより、IKによる角度が不連続に変化する場合でも、補間結果としてなめらかな角度の変化が得られるようになります。

(d) 最後に、オフセット角度の追従結果を、もともとのアニメーションデータの角度に加えてキャラクタの姿勢に適用することで、最終的なポーズを計算します。

このように、IKによる角度変化を追従の目標値とすることによって、元のアニメーションを忠実に再生しながら、IKによる姿勢変化をなめらかに加えていくことが可能になります。

PD制御による補間の性質

1. 力学的モデルによるなめらかな補間結果が得られる

2. 補間に要する時間は動的に変化する
   - 補間が始まった瞬間の状態に依存して補間時間が変化する
   - 調整パラメータでおおよその補間時間が決まる

3. 目標位置に限りなく近づいて行くが、
   数学的には決して目標位置に一致しない (漸近する)

さらに、押さえておくべきPD制御の重要な性質として、厳密には目標に達することはないという性質があります。

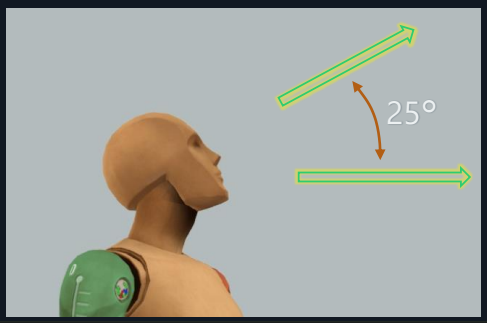
もう少し正確にいえば、PD制御によって目標値を追従させると目標値に限りなく近づいてはいくものの、もし目標値が全く変化しない場合であっても、数学的にはいつまでたっても決して目標値に一致することはないという性質があります。このような挙動のことを、「目標値に漸近する性質がある」といいます。

次のスライドからは、PD制御を IK に応用したときにこの漸近する性質がどのように影響するかについて見ていきます。

IK の効果と PD 制御による目標への漸近

(例) 見上げる姿勢を Look-at IK で作る

- [IK オフ] 初期状態の視線は水平方向
- [IK オン] 水平から上方に 25° の目標位置



オフセット = 25°

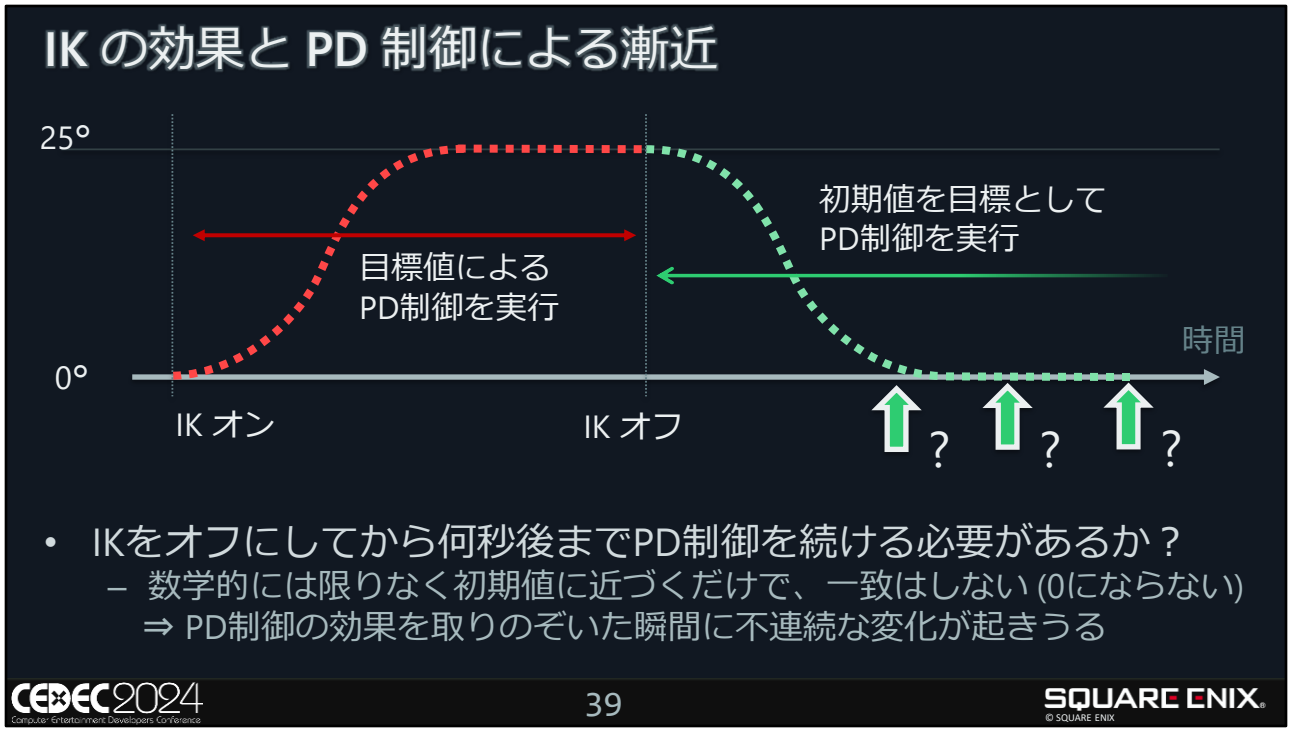
オフセット = 0° (水平)

このスライドの動画は、キャラクタの首のジョイントの角度をPD制御の対象として、上を見上げる姿勢に変化する動きを Look-at IK で作ったようすを表しています。

IK がオフのときは、目標位置が与えられていないのでアニメーションデータの姿勢をそのまま再現するために、ジョイントの角度のオフセットは0度です。

IK をオンにしたときは、上を見上げるために 25度のオフセットを目標値として与えています。

PD制御でこの目標値を追従することで、オフセット角度が0度から25度までなめらかに変化する動きを作り出すことができます。



IK のオンオフによる首の角度の変化をグラフとして表すと、このスライドのようになります。

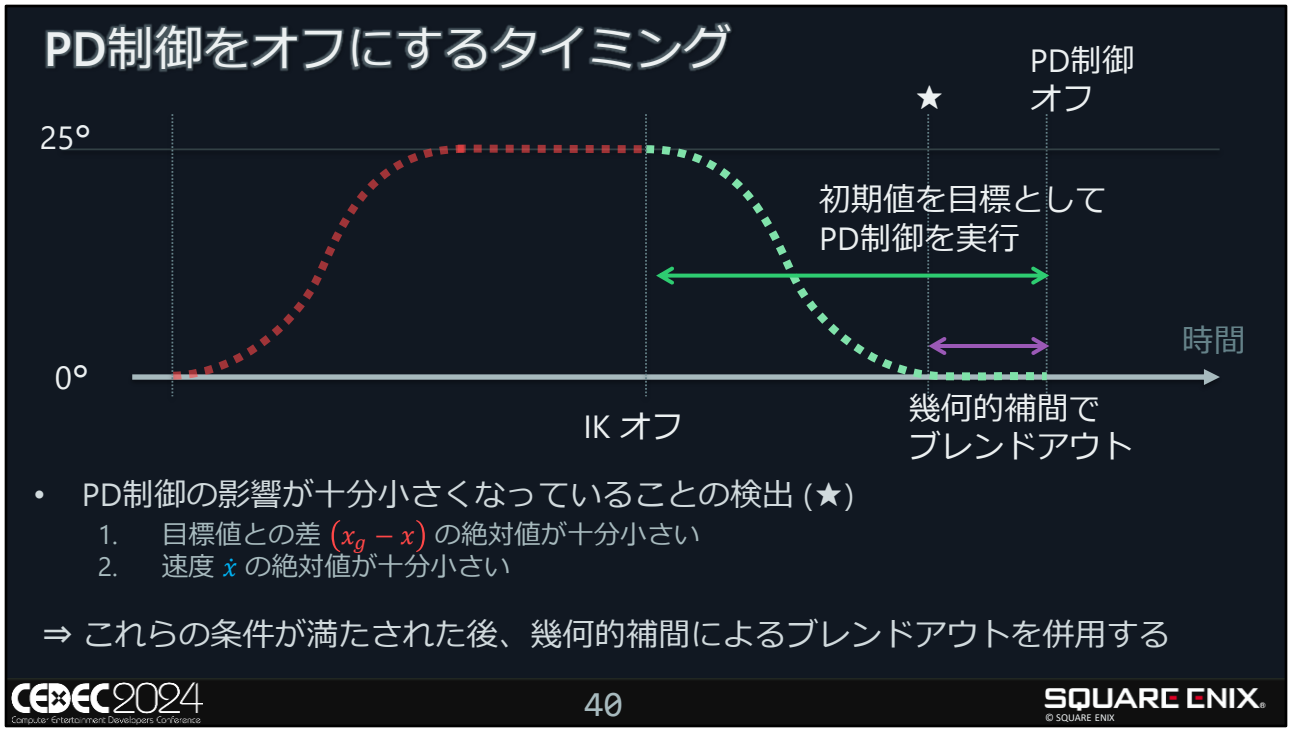
最初はIKがオフの状態から始まるので、オフセットが0度から始まります。

IKをオンにすると、グラフの赤い点線のように0度から25度までなめらかに変化します。これは目標値を25度として PD 制御による目標値への追従が行われた結果となります。

次にIKをオフにすると、グラフの緑色の点線のようにもとの0度に向かって角度がなめらかに変化します。PD制御がは、0度を目標値として実行されます。

このとき、IKをオフにしたあと、最終的にPD制御をいずれどこかで止める必要があります。問題は、いつPD制御を止めてもよいか、何秒後に止めればよいかというところです。

PD制御の性質として説明したように、数学的には決してオフセットの値は0度になりません。したがって、PD制御をやめてオフセットの角度を取り除いた瞬間に、キャラクタの姿勢に不連続な変化が起こってしまう可能性があります。



では実際どうすればよいかというところですが、一つの方法として、PD制御を利用したときの、目標値0にどんどん近くなっていく性質を利用することはできます。

スライドの例で言えば、IKがオフになってからある程度の時間が経つと、追従結果の値は0に近くなります。このとき、もし十分0に近くなっていれば、そこからさらに通常の幾何的補間、ブレンドアウトを併用することで、IK の効果による角度のオフセットを完全に0にすることができます。

十分0に近いかどうかの検出は、2つの条件をチェックすることで可能です。

1つは、追従結果と目標値との差の絶対値が十分小さいこと、
もう一つは、追従結果が変化する速度の絶対値が十分小さいことです。

これらの2つの条件が満たされていれば、その時点から幾何的補間を併用してブレンドアウトさせます。目標値との差は小さくなっているので、このときのブレンドアウトは線形補間で十分です。

このようにして、オフセットの角度が不連続に変化することを避けながら PD 制御自体をオフにすることができます。

PD制御の対象量

- 1次元量の補間
  - 位置
  - 回転角
- ベクトルや回転を表す量の補間
  - 3次元単位ベクトル
  - 四元数 (quaternion)

(1) これまでのスライドでは、位置や角度などの1次元の量に関する PD 制御について紹介してきました。

(2) 次のスライドからは、発展的な話題として、3次元の単位ベクトルやクォータニオンを補間したいときにどうするかについて説明します。

3次元単位ベクトルを PD 制御で補間

- 厳密な補間方法
  - 単位ベクトル (x, y, z) が存在する単位球面上で補間
- 簡略化した方法
  1. 単位ベクトルの成分 x, y, z を、成分毎に別々に補間
  2. ベクトルの長さを1に正規化

⇒ Look-at IK の補間のためには簡略な方法で十分

(※ 完全に正反対のベクトル同士での補間はできないことに注意)

まず3次元の単位ベクトルについてです。

3次元の単位ベクトルについて PD 制御で補間したい場合には、厳密には単位ベクトルが存在する単位球面上で補間を行う必要があります。

しかし実際に Look-at IK でアニメーションの補間を行う場合には、単位ベクトルの x, y, z 成分をそれぞれ1次元のPD制御で補間して、そのあとにベクトルの長さが1になるように正規化する方法で実用的には十分です。

もちろん球面上の角速度が一定でないといけないような用途には不十分ですが、そういった厳密さが必要ない場合にはこうした簡略化した方法が利用可能です。

ただし、完全に反対方向を向いたベクトル同士は補間できないので、その点注意が必要です。

四元数 (quaternion) を PD 制御で補間 (1/2)

- 厳密な補間方法
  - 球面線形補間 (slerp)
    1. 角度成分 w は、余弦 $\cos\theta$ から角度 θ に戻してから補間
    2. 補間の重みについて正弦 $\sin\varphi$ の計算を含む
- 簡略化した方法
  - x, y, z, w の成分毎に別々に補間してから正規化

  ⇒ Look-at IK の補間のためにはこれで十分

次にクォータニオンをPD制御で補間したい場合について考えます。

結論から言えば、単位ベクトルと同様に、実用的にはクォータニオンの4つの要素ごとに1次元のPD制御で補間し、最後に正規化することで実用的には十分な結果が得られます。

数学的に厳密な方法について説明しておくと、厳密には球面線形補間を利用する方法があります

回転成分 w については余弦で表されているので、一旦角度に戻してから補間し、再び余弦を計算します。

また、補間の重みを計算するときに正弦関数を計算する必要があります。

ですが、Look-at IK のアニメーションの補間を行うには、簡略化した方法でも十分実用的な見た目を得ることができているので、こちらを利用しています。

四元数 (quaternion) を PD 制御で補間 (2/2)

- 角速度・角度の制限
  - 四元数の w 成分は、回転角度の余弦を表している
    - $w = \cos\frac{\theta}{2}$
    - w 成分から角度 θ を計算し、θ を制限し、再び w 成分に戻す
  - 角度 θ を計算せずに済ませる (Look-at IK)
    - θ の値域を $-\pi < \theta < \pi$ とすれば、この区間で $\cos\frac{\theta}{2}$ は単調増加
    - 最大角度を θ_{max}に制限したいとき
      1. あらかじめ $w_{max} = \cos\frac{\theta_{max}}{2}$ を計算しておく
      2. w 成分を $w \leq w_{max}$ の範囲に制限する

また関連した話題として、クォータニオンを PD 制御で補間するとき、角度や角速度を一定の範囲に制約したい場合があります。

このようなとき、クォータニオンの w 成分は回転角の余弦を表しているので、角度を制約するときには一旦余弦から角度を計算してから角度を制約し、再び余弦に戻す必要があります。

ただし、もし角度の範囲が -180° から180° の範囲にあれば、この角度の区間で余弦は-1から+1まで単調増加する関数になります。

ということで、もし制限する角度の範囲があらかじめ決まっている場合には、制限角の余弦をあらかじめ計算しておき、余弦によって直接クォータニオンの w 成分を制限することができます。このようにすることで、少し計算量を節約することができます。

PD制御の追従性が高い場合の計算誤差 (バネが強い場合)

- 素朴な実装では、前進差分法で値を更新する

```
v += dt * F; // 速度の更新
x += dt * v; // 位置の更新
```

- この方法は近似であり、実物とは誤差がある
  - バネ k_p が強く働くとき、計算誤差が大きくなりやすい
  - 目標値からどんどん遠ざかることもある (発散する)

⇒ 誤差を減らすために時間刻み dt を**より小さく調整する**必要がある

次に、PD制御の計算の安定性について説明したいと思います。

先の実装例のスライドで示した例では、素朴な実装として前進差分法によって速度と位置の値を更新していました。

前進差分法では、物体に加わる力つまり加速度に時間刻みをかけたものを加えて速度を更新し、速度に時間刻みをかけたものを加えて位置を更新します。

この更新方法は一種の近似的な方法で、実物と比べると誤差があります。

どのような誤差があるかと言えば、とくにバネが強く働くとき、物体に働く力も大きくなるので計算誤差が大きくなりやすくなります。

このようなときには、PD制御をおこなっているのに目標に近づかず、目標から遠ざかったり振動がどんどん発散したりします。

前進差分法によるこうした計算の誤差を小さくするためには、基本的には時間刻み dt を小さくする必要があります。

時間刻みの調整の実装例

```
void PD::stepUpdate(float dt, float xg) {
  float dtMax = 1.0f / 120;   // 時間刻みの上限値（例）
  while (dt > 0.0f) {
    float stepDt = std::fmin(dt, dtMax);
    update(stepDt, xg); // dtMax を超えない時間刻みに→
    dt -= stepDt;       // →よって分割更新する
  }
}

/* 問題点
  – 処理負荷が増える
  – 時間刻みの上限値を非常に小さくしないと発散する場合がある
    (発散しない上限値の決め方は調整パラメータに依存する)      */
```


このスライドでは時間刻みを小さくする実装の例について説明します。

仮に、時間刻みの最大値を 1/120 秒 とします。

もしこれより長い時間刻みが与えられたら、1/120秒ずつに小さく分割して処理することで誤差を小さくするという関数になります。

この実装方法の問題点は、まず何回かにわけて処理することで、処理負荷が増えるということです。

また、このスライドでは仮に 1/120秒を上限として時間刻みを分割していましたが、実際に発散しない上限値というのはバネや粘性抵抗などのパラメータに基づいて決める必要があります。ですので、パラメータの設定によっては非常に小さい時間刻みが必要になり、ますます処理負荷が増えることになります。

Stable PD

"Stable Proportional-Derivative Controllers"
Jie Tan, Karen Liu and Greg Turk
IEEE Computer Graphics and Applications (CG&A) 2011

時間刻み dt の調整を不要にする手法

- バネ k_p が強く働くときでも
  時間刻みの上限値についてあまり考えなくてもよい

- 後退差分法と一次のテイラー展開を利用
  - 結果が発散しない条件は $k_d/k_p > \mathrm{dt}$
  - Look-at IK でよくある設定値はこの条件を満たす

このような時間刻みによる安定性の問題を解決する方法が提案されているのでここで紹介したいと思います。

この Stable PD の手法を使うと、多くの場合には与えられた時間刻み dt をそのまま使うことができます。

バネの強さが大きく設定されていても、dt を調節する必要がなくなるということです。

手法としては、前進差分法の代わりに後退差分法と一次のテイラー展開による近似を利用します。

時間刻み dt を調節する必要がないと言いましたが、結果が発散しない条件がわかっていて kd/kp が dt よりも大きいという条件です。

ただ、通常 Look-at IK に利用するような設定値であればこの条件を満たしていることが多いので、Stable PD を使う場合には dt を小さく分割するようなケースはあまり発生しません。

Stable PD

"Stable Proportional-Derivative Controllers"
Jie Tan, Karen Liu and Greg Turk
IEEE Computer Graphics and Applications (CG&A) 2011

- 前進差分法による力 F

  $F = k_p(x_g - x^n) - k_d\dot{x}^n$ … x^n $\dot{x}^n$は現在の位置と速度

- 後退差分法による力 F

  $F = k_p(x_g - x^{n+1}) - k_d\dot{x}^{n+1}$ (1) x^{n+1} $\dot{x}^{n+1}$は未来の位置と速度

  – 未来の値をテイラー一次近似し、現在の値で置き換え
    - $x^{n+1} = x^n + dt\dot{x}^n$ (2)
    - $\dot{x}^{n+1} = \dot{x}^n + dt\ddot{x}^n = \dot{x}^n + dtF$ (3)

  – 式(1) 内の未来の値を上の近似 (2)(3) で置き換え
    - $F = k_p(x_g - x^n - dt\dot{x}^n) - k_d(\dot{x}^n + dtF)$ (4)

- 後退差分法による力 F (5) 現在の位置と速度を用いた近似式

  $F = (k_p(x_g - x^n - dt\dot{x}^n) - k_d\dot{x}^n) / (1 + dtk_d)$

実際に Stable PD でどのように運動方程式の力を計算するかを見ていきます。

まず、従来の前進差分法を用いた場合、目標を追従するために物体に加える力 F はスライドの方程式のようになります。

ここで、x と x. の右肩に n という記号がつくようになりましたが、この n は現在の位置と速度であるということを表しています。

(1) 次に Stable PD で後退差分法を利用する場合ですが、この場合には1ステップ未来の位置と速度を利用して力を計算します。

右肩の記号の n+1 が未来の状態であることを表しています。

ただし問題は、未来の状態はここで計算している力Fによって決まるということです。

したがって、この方程式は、左辺のF と、右辺の未来の位置と速度の両方が未知数になっています。

(2) こうした後退差分法の方程式を解く一般的な方法として、よく使われる方法として、未来の値を現在の値によって近似する方法を使います。

式(2)と(3) は、未来の値をテイラー一次近似を用いて現在の値によって表した式に

なります。

(3) この近似式(2)(3)を式(1)に代入して置き換えることで、式(4) のように力Fを現在の値だけで表すことができます。

式(4) にはまだ右辺に F が残っているので、これを整理します。

(4) そうすると、一番下の式(5) のように、現在の値だけを利用して力 F を表すことができます。

Stable PD の実装例

"Stable Proportional-Derivative Controllers"
Jie Tan, Karen Liu and Greg Turk
IEEE Computer Graphics and Applications (CG&A) 2011

```
void PD::stableUpdate(float dt,      // 時間刻み
                      float xg) {    // 目標位置
  // (1') mẍ の分子を計算
  float F = kp * (xg - x - dt * v) - kd * v;

  v += dt * F / (1.0f + kd * dt); // (2') 速度を更新
  x += dt * v;                    // (3) 位置を更新
}
/*
 前進差分法による PD::update() に補正項を加えた実装になる
 */
```


前のスライドの説明は少し複雑でしたが、最終的に出てくる近似式は簡単になっているので、プログラムとしての実装は難しくありません。

実装としては、前進差分法によるアップデート関数に対して、少し補正項を加えたものになります。

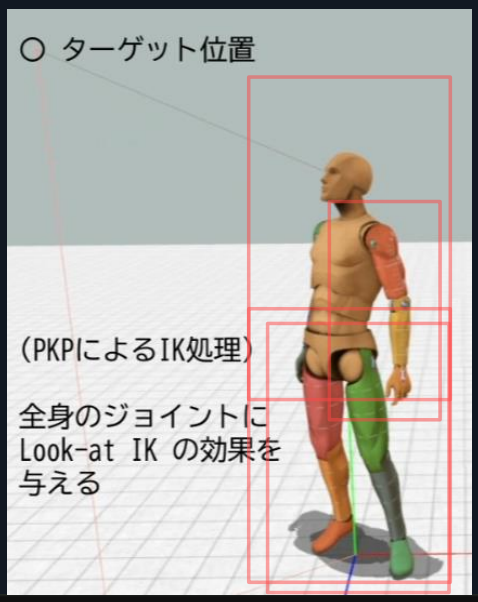
プログラム中では、(1 ') と (2') の行で黄色の文字で書いたところが前進差分法からの変更点になります。

また、前のスライドで説明したように、kp と kd の比率でこの関数の計算結果が不安定になるかどうかをチェックすることができます。

ですので、この関数にそのようなチェックを組み込むこともできます。

PD 制御を利用した Look-at IK 機能の実装

- ターゲット位置をなめらかに追従
- 頭・首から脊椎・骨盤まで
  一連のジョイントを動かして追従
- 視線方向変化による体重移動
- 胸郭の動きが腕に反動を与える
- 両脚の姿勢調整



以上のスライドで説明したような PD 制御の方法を、実際に Look-at IK として実装した機能について紹介したいと思います。

Look-at IK の基本的な機能として、与えられたターゲットの位置に頭を向けることができます。

このとき、現在の姿勢からターゲットを向く姿勢まで、なめらかに補間が行われます。

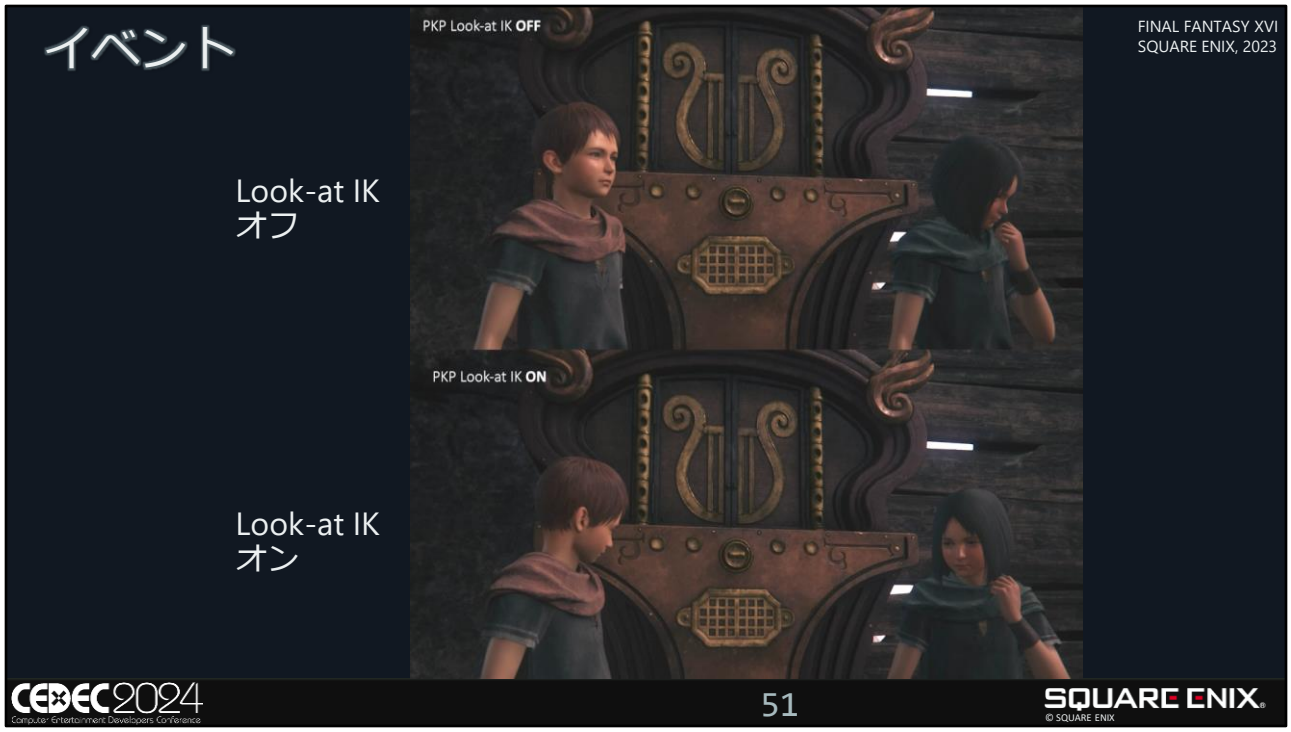
(1) また、姿勢を変えるときには、首のジョイントだけではなくて、頭から首、頚椎・脊椎・骨盤まで、関係する一連のジョイントすべてを動かして追従させます。このとき、首から上の追従は、胴体よりも先にターゲットの方を向きます。

(2) 頭の重量はそれなりに重いので、視線方向を変化させたときにはバランスを取るために体重移動の表現を加えます。動画では特に腰の位置が変化しているのがおわかりいただけると思います。

(3) また、視線を変えたときには胸郭・胴体が回転するので、この影響をうけて腕が少し動く表現も加えます。こうすることで、腕が胴体に固くくっついている印象を避けることができます。

(4) さらに、体重移動で腰の位置を変えたときには、そのままでは足先の位置まで変わってしまうので、足先が地面をすべらないように足全体の姿勢を調整します。

このような一連の表現を加えることで、視線方向の変化が全身のジョイントに影響を与えるような Look-at IK の機能を実装しています。



最後にもう一度、FF16 で Look-at IK の機能を利用している場面を見ていきます。ひとつめはゲーム中のイベントについての動画です。

下が Look-at IK をオンにしたときの様子、上が比較のために Look-at IK をオフにした様子を表しています。

ゲームプレイ

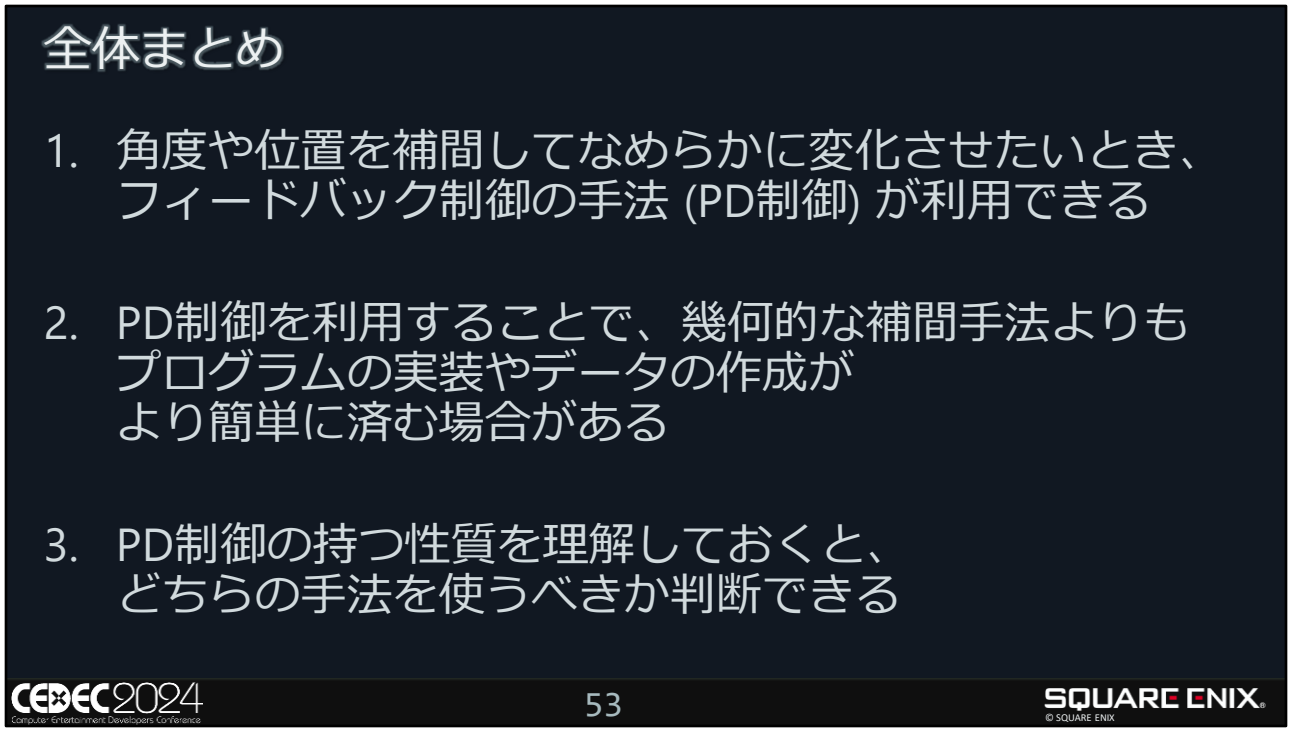
Look-at IK
オフ

Look-at IK
オン

FINAL FANTASY XVI
SQUARE ENIX, 2023

次のスライドは、ゲームプレイ中で Look-at IK を使用している場面です。

このとき、プレイヤーとその相手になるキャラクターは、互いの顔をみるように Look-at IK が働いています。



最後に発表全体についてまとめたいと思います。

この発表では、フィードバック制御の方法である PD 制御を利用して、位置や角度をなめらかに補間する方法について説明してきました。

また、PD 制御の特徴として、Look-at IK 機能のためのプログラム実装やデータの作成が簡単にできることを紹介しました。

さらに、幾何的な補間方法と PD 制御の方法について、どちらの方法がより向いているかを判断する手がかりとするために、それぞれの性質を比較しました。

Q&A

本発表に関するご質問・ご意見には
CEDECオンラインライブ・チャットをご利用ください

FINAL FANTASY XVI, SQUARE ENIX, 2023

（質疑応答）

質疑応答 (1/3)

(Q) Look-at IK に伴う全身の体重移動や腕の反動の表現については、フルボディIK などの別の技術を使用しているのか？

(A) 腰から上の Look-at IK の効果を処理したあとで、体重移動の効果・腕の反動の効果を順番に計算して重ねる処理を行っています。フルボディIK のような全身の同時計算ソルバーは使用していません。

（CEDECオンラインライブ・チャットで行われた質疑応答）

質疑応答 (2/3)

(Q) 制御の目標値が急激に変化する等によって結果が振動してしまう現象は、今回の手法によって完全に回避できたか？

(A) スライド中で紹介した Stable PD を使うことにより、アニメーションの振動や発散といった不具合の発生を防ぐことができました。

（CEDECオンラインライブ・チャットで行われた質疑応答）

質疑応答 (3/3)

(Q) インゲームでのPD制御の利用において、I制御がなくても十分との結論は数値的な評価によるものか？アーティストの主観評価によるものか？

(A) 最終的なアニメーション出力結果を見てI制御の必要性はなかったという判断です。
もし移動するターゲットを銃で正確に狙うような場面があればおそらく問題になったと思いますが、今回はそういったケースがありませんでした。

（CEDECオンラインライブ・チャットで行われた質疑応答）