Computational Robot Dynamics

Part 3: Improving Efficiency

Roy Featherstone

ISTITUTO ITALIANO
DI TECNOLOGIA
ADVANCED ROBOTICS

Contents

Recursive algorithms are already highly efficient; but there are a few things that can be done to implement them in the most efficient way. We shall look at these three:

- efficient spatial arithmetic,
- simplifying the model, and
- symbolic simplification.

6x6 Matrices Versus Compact Classes

In a matrix-oriented environment it makes sense to represent rigid-body inertias and Plücker coordinate transforms using 6x6 matrices. This approach is simple and easy, but computationally inefficient.

The cost of an $l \times m \times n$ matrix multiplication is $lmn\mathrm{m} + l(m-1)n\mathrm{a}$.

floating-point multiplication

floating-point addition or subtraction

So a 6x6x6 multiplication costs 216m+180a.

6x6 Matrices Versus Compact Classes

A solution to this problem is to store these quantities in classes (or data structures) equipped with a set of methods (functions) that implement efficient versions of each arithmetic operation.

6x6 matrix	class or data structure that represents it
$\begin{bmatrix} \boldsymbol{E} & \mathbf{0} \\ \boldsymbol{E}\tilde{\boldsymbol{r}}^{\mathrm{T}} & \boldsymbol{E} \end{bmatrix}$	$\texttt{plx(}\boldsymbol{E}\texttt{,}\boldsymbol{r}\texttt{)}$
$\begin{bmatrix} \boldsymbol{I} & \tilde{\boldsymbol{h}} \\ \tilde{\boldsymbol{h}}^{\mathrm{T}} & m\mathbf{1} \end{bmatrix}$	$\texttt{rbi(}m\texttt{,}\boldsymbol{h}\texttt{,}\mathrm{lt}(\boldsymbol{I})\texttt{)}$

6x6 Matrices Versus Compact Classes

A solution to this problem is to store these quantities in classes (or data structures) equipped with a set of methods (functions) that implement efficient vers[…] operation.

an object of type 'plx' having fields called 'E' and 'r' in which are stored the matrix $\boldsymbol{E}$ and the vector $\boldsymbol{r}$.

6x6 matrix

$$\texttt{plx}(\boldsymbol{E},\boldsymbol{r})$$

an object of type 'rbi', having fields called 'm', 'h' and 'I', in which are stored the scalar m, the vector $\boldsymbol{h}$ and the lower triangle of the symmetric 3x3 matrix $\boldsymbol{I}$.

$$\texttt{rbi}(m,\boldsymbol{h},\mathrm{lt}(\boldsymbol{I}))$$

6x6 Matrices Versus Compact Classes

These classes are more compact (i.e., they occupy less memory) than a 6x6 matrix.

Object	Size
6x6 matrix	36
plx(...)	12
rbi(...)	10

('size' is the number of floating-point numbers in the object)

6x6 Matrices Versus Compact Classes

However, their main advantage is the opportunity to implement efficient versions of common spatial arithmetic operations.

operation	computational cost 6x6 matrix	computational cost efficient
$\boldsymbol{X}\boldsymbol{v}$	$36\text{m} + 30\text{a}$	$24\text{m} + 18\text{a}$
$\boldsymbol{X}^*\boldsymbol{f}$	$36\text{m} + 30\text{a}$	$24\text{m} + 18\text{a}$
$\boldsymbol{X}_1\boldsymbol{X}_2$	$216\text{m} + 180\text{a}$	$33\text{m} + 24\text{a}$
$\boldsymbol{I}\boldsymbol{v}$	$36\text{m} + 30\text{a}$	$24\text{m} + 18\text{a}$
$\boldsymbol{X}^{\mathrm{T}}\boldsymbol{I}\boldsymbol{X}$	$432\text{m} + 360\text{a}$	$52\text{m} + 56\text{a}$

(this is not a complete list)

Some Examples

Operation: $\boldsymbol{X}\hat{\boldsymbol{v}}$

$$9\mathrm{m} + 6\mathrm{a}$$

$$\begin{bmatrix} \boldsymbol{E} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{E} \end{bmatrix} \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} \\ -\boldsymbol{r}\times & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E}\boldsymbol{\omega} \\ \boldsymbol{E}(\boldsymbol{v} - \boldsymbol{r} \times \boldsymbol{\omega}) \end{bmatrix}$$

$$9\mathrm{m} + 6\mathrm{a} \qquad 3\mathrm{a} \qquad 6\mathrm{m} + 3\mathrm{a}$$

$$= 24\mathrm{m} + 18\mathrm{a}$$

Some Examples

Operation: $\boldsymbol{I}\hat{\boldsymbol{v}}$

$$\begin{bmatrix}\bar{\boldsymbol{I}} & \boldsymbol{h}\times \\ -\boldsymbol{h}\times & m\mathbf{1}\end{bmatrix}\begin{bmatrix}\boldsymbol{\omega}\\ \boldsymbol{v}\end{bmatrix} = \begin{bmatrix}\overset{9\mathrm{m}+6\mathrm{a}}{\bar{\boldsymbol{I}}\boldsymbol{\omega}} \overset{3\mathrm{a}}{+} \boldsymbol{h} \overset{6\mathrm{m}+3\mathrm{a}}{\times} \boldsymbol{v} \\ \underset{3\mathrm{m}}{m\boldsymbol{v}} \underset{3\mathrm{a}}{-} \boldsymbol{h} \underset{6\mathrm{m}+3\mathrm{a}}{\times} \boldsymbol{\omega}\end{bmatrix}$$

$$= 24\mathrm{m} + 18\mathrm{a}$$

Exploiting Type Sensitivity

Many programming languages allow a function to perform different actions according to the types of its arguments.

For example, a function to apply a Plücker transform can be written in such a way that it uses the correct transformation rule for each type of object.

```
plx    X = ...
M6vec  v1, v2=...
F6vec  f1, f2=...
rbi    I1, I2=...
v1 = X.apply(v2);
f1 = X.apply(f2);
I1 = X.apply(I2);
```

$$\boldsymbol{v}_1 = \boldsymbol{X}\boldsymbol{v}_2$$

$$\boldsymbol{f}_1 = \boldsymbol{X}^*\boldsymbol{f}_2$$

$$\boldsymbol{I}_1 = \boldsymbol{X}^*\boldsymbol{I}_2\boldsymbol{X}^{-1}$$

'apply' and 'invapply'

The special form of the Plücker transform matrix means that it is never necessary to compute explicitly its inverse. Instead, one defines two functions, 'apply' and 'invapply':

X.apply(...)

apply the coordinate transform described by X to the argument

X.invapply(...)

apply the *inverse* of the coordinate transform described by X to the argument

'apply' and 'invapply'

X.apply(v):

$$\begin{bmatrix} \boldsymbol{E} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{E} \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\boldsymbol{r}\times & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} \overbrace{\boldsymbol{E}\boldsymbol{\omega}}^{9\text{m}+6\text{a}} \\ \underbrace{\boldsymbol{E}}_{9\text{m}+6\text{a}}(\boldsymbol{v} \underbrace{-}_{3\text{a}} \underbrace{\boldsymbol{r}\times\boldsymbol{\omega}}_{6\text{m}+3\text{a}}) \end{bmatrix}$$

X.invapply(v):

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \boldsymbol{r}\times & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{E}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{E}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E}^{\mathrm{T}}\boldsymbol{\omega} \\ \boldsymbol{E}^{\mathrm{T}}\boldsymbol{v} \underbrace{+}_{3\text{a}} \boldsymbol{r}\times\boldsymbol{E}^{\mathrm{T}}\boldsymbol{\omega} \end{bmatrix}$$

$\text{cost (both)} = 24\text{m} + 18\text{a}$

Simplifying the Model

Infinitely many different rigid-body systems have the same equation of motion.

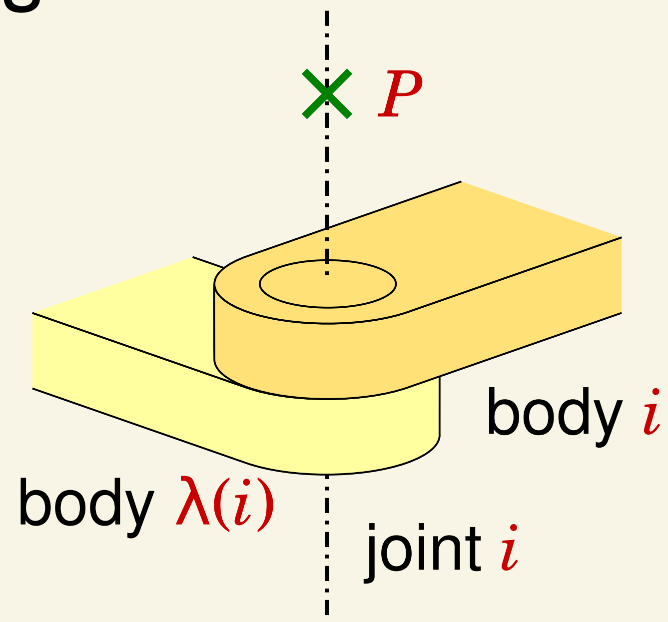
Therefore it is usually possible to replace a given system model with a simpler one having the same equation of motion.

In this context, 'simpler' means having fewer nonzero parameters, which can reduce the cost of calculating the equation of motion.

Example: Adding Point Masses

Suppose that joint i is revolute.

This means that any point on the joint axis is fixed both in body i and in body $\lambda(i)$.

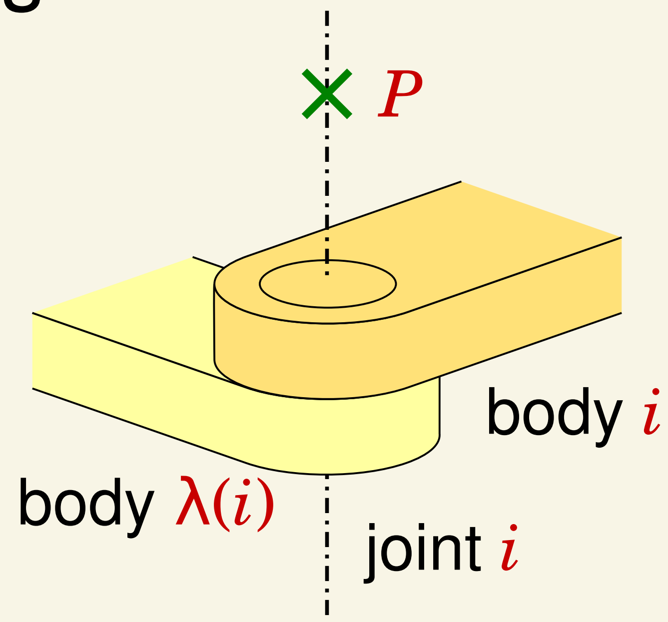


Example: Adding Point Masses

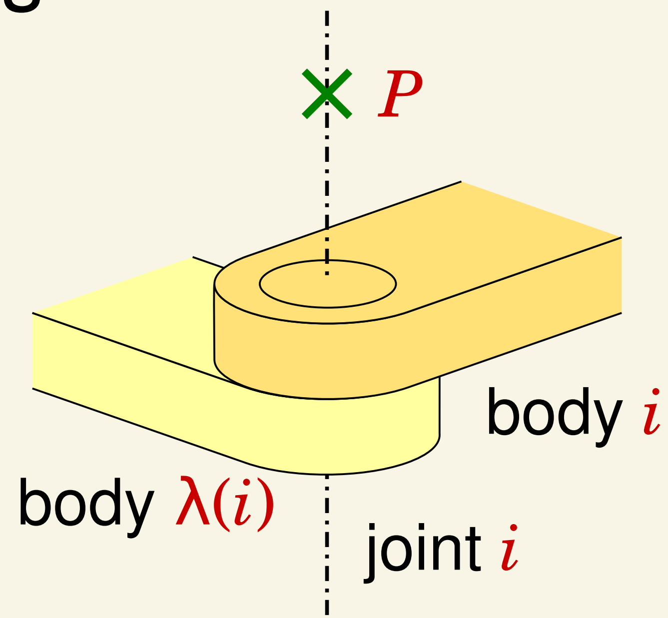
Suppose that joint i is revolute.

This means that any point on the joint axis is fixed both in body i and in body $\lambda(i)$.

Now suppose that we add a point mass m to body $\lambda(i)$ at P, and a corresponding point mass $-m$ to body i, also at P.



Example: Adding Point Masses



Suppose that joint i is revolute.

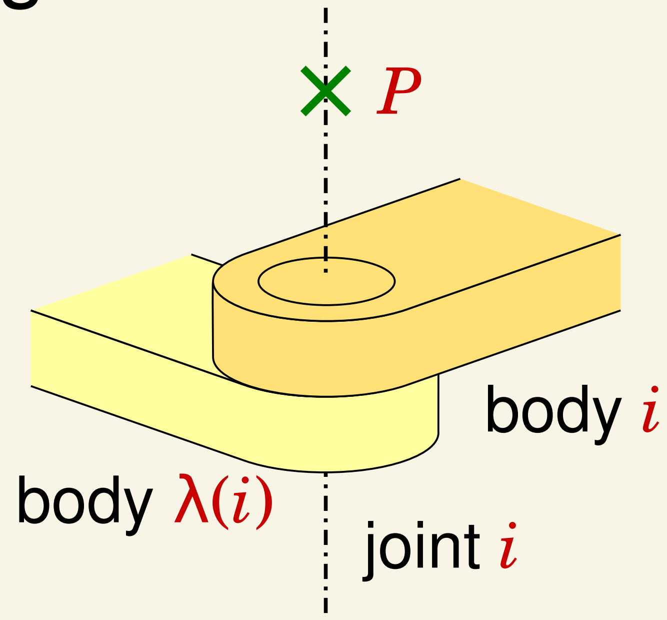
This means that any point on the joint axis is fixed both in body i and in body $\lambda(i)$.

Now suppose that we add a point mass m to body $\lambda(i)$ at P, and a corresponding point mass $-m$ to body i, also at P.

These point masses cancel, and therefore have no effect on the equation of motion; but both body i and body $\lambda(i)$ now have different inertias.

Example: Adding Point Masses

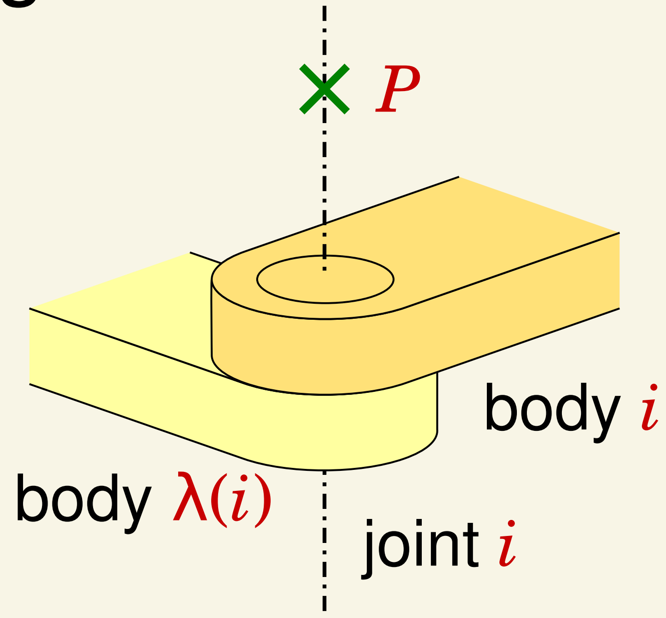
Suppose we set $m = m_i$ (the mass of body i). The effect is to zero the mass of body i.

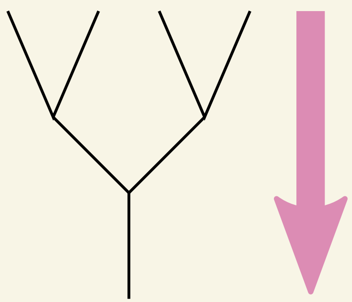


Example: Adding Point Masses

Suppose we set $m = m_i$ (the mass of body i). The effect is to zero the mass of body i.

If we do this at every revolute joint in the system, starting at the leaves and working towards the root, then most or all of the bodies will have zero mass.

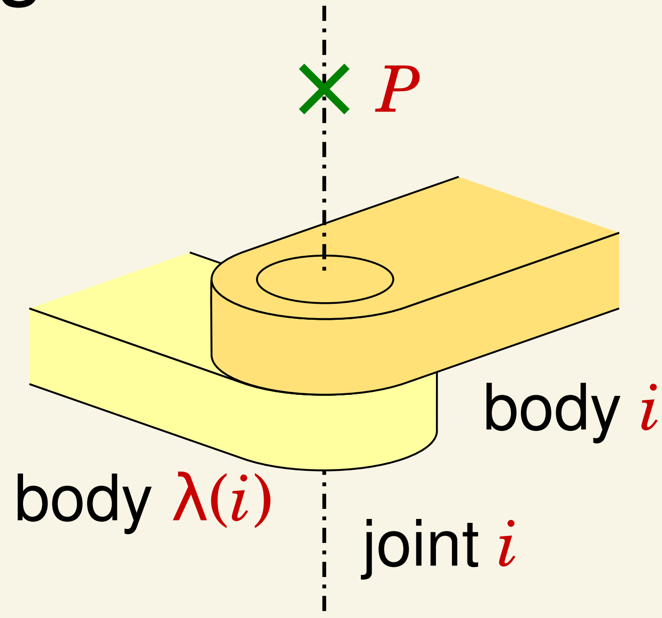


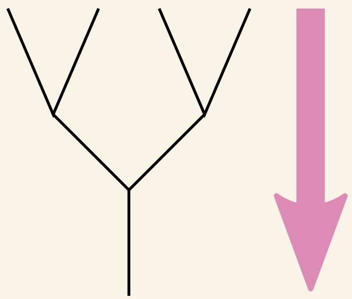


Example: Adding Point Masses

Suppose we set $m = m_i$ (the mass of body i). The effect is to zero the mass of body i.

If we do this at every revolute joint in the system, starting at the leaves and working towards the root, then most or all of the bodies will have zero mass.





If we know that a body has zero mass then we can simplify the operations that use its inertia.

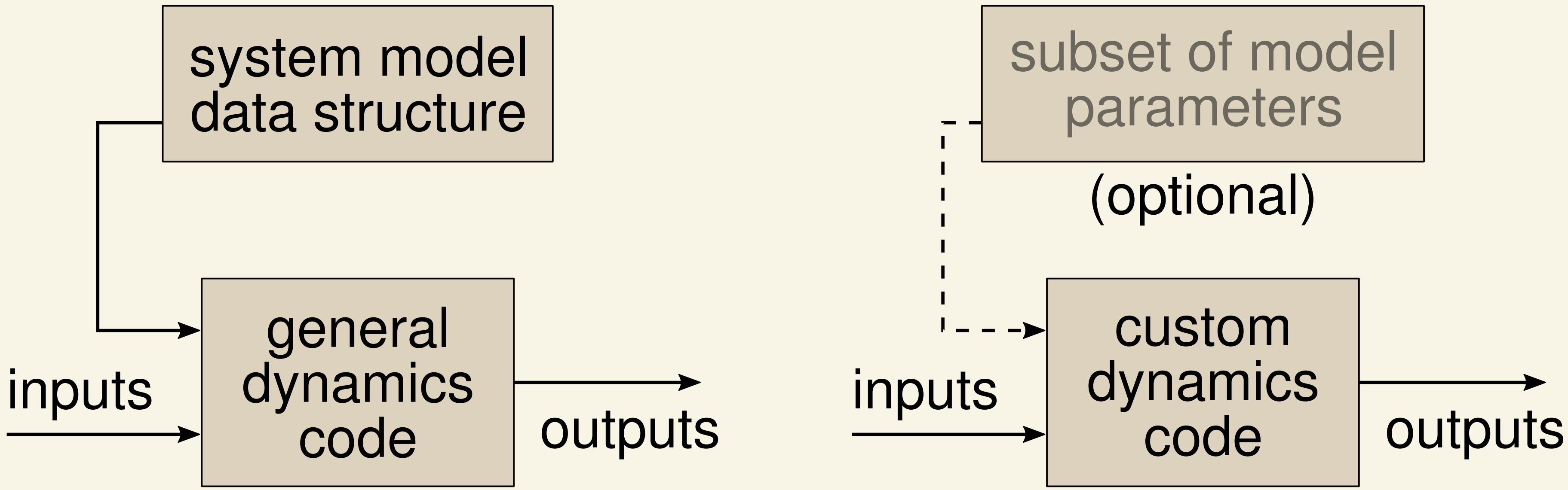
Other Uses

Techniques like these can improve the computational efficiency of dynamics calculations by around 10–20%, but they also have other uses. For example,

- allow the analyst to use simpler equations, or generalize a result; or
- show the designer how to change a design without altering its dynamics; or
- improve the effectiveness of symbolic simplification (which is the next topic . . .).

Symbolic Simplification

In robot dynamics, the term *symbolic simplification* refers to the *automatic* production of computer source code that implements *customized* versions of general dynamics calculations.



Symbolic Simplification

Symbolic simplification exploits the fact that many of the parameters in a practical system have special values, like zero or pi/2. This allows many calculations to be simplified.

Example: product of rotation matrix $\boldsymbol{E}$ with vector $\boldsymbol{v}$.

general case

cost: 9m+6a

$$\begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

special case: pi/2 rotation

cost: 0m+0a

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Symbolic Simplification

The symbolic simplification process works approximately like this: