The 4th Joint International Conference on Multibody System Dynamics
May 29 – June 2, 2016, Montréal, Canada

PyCraft: An Analytical and Computational Workbench for System Level Multibody Dynamics

Abhinandan Jain[1]

[1]*Mobility and Robotic Systems, Jet Propulsion Laboratory/California Institute of Technology, jain@jpl.nasa.gov*

ABSTRACT

In this paper, we describe the PyCraft computational workbench for studying and evaluating complex, system level dynamics properties of multibody systems. Examples of such dynamics properties are the system mass matrix, Jacobians and sensitivities of these quantities that are important for design and optimization of dynamics properties, as well as for controller development. In this paper we describe the C++/Python PyCraft workbench which builds upon rich mathematical operator methods from the Spatial Operator Algebra (SOA) that have been used for analysis and algorithm development for multibody dynamics. Mathematical operator expressions can be literally transcribed to the PyCraft command line to allow the easy evaluation of complex dynamics quantities. Several examples illustrating operator based analysis and corresponding PyCraft execution are included.

1 Introduction

The multibody research community has devoted much attention to the development of fast, accurate and stable methods for solving the equations of motion at the heart of dynamics simulations. In this paper, we address a complementary problem that has received much less attention - that of studying and evaluating system level dynamics properties of multibody systems. Examples of such properties are the system mass matrix, Jacobians and sensitivities of these quantities that are important for robotics, the design and optimization of dynamics properties, development of variational integrators and for controller development. Instead of brute force numerical methods, in this paper we describe an elegant and general framework for computing system dynamics properties that exploits the rich analytical structure of the underlying system dynamics. The framework defines a concise and expressive vocabulary of operator quantities that can be used to analytically describe a very broad class of system dynamics quantities, use them to carry out mathematical analysis, and to use a computational workbench for interactively evaluating these complex quantities based on the analytical expressions.

The family of operators mentioned above are from the *Spatial Operator Algebra (SOA)* framework for multibody dynamics [1, 2]. The SOA is based on a minimal coordinates dynamics representation, and shows that a family of *spatial operators* can be used to elegantly describe multibody quantities, reveal their underlying structural patterns [3], and carry out analysis and computations with them. For example, the mass matrix $\mathcal{M}$ for any tree topology system can be expressed as

$$\mathcal{M} = \mathrm{H}\phi\mathrm{M}\phi^*\mathrm{H}^* \tag{1}$$

where the block-diagonal H and M operators contain hinge articulation and body spatial inertia elements respectively, while the elements of ϕ define rigid body transformations between pairs of bodies. Beyond the elegance and compactness of such expressions, the spatial operators have important analytical properties that allow us to carry out further mathematical analysis. In particular, the ϕ operator has the form $\phi = (\mathrm{I} - \mathcal{E}_\phi)^{-1}$, where $\mathcal{E}_\phi$ is another operator whose non-zero block elements correspond to the adjacency matrix for the directed graph that describes the multibody system's topology. Exploiting such analytical structure, the SOA shows that the mass matrix can be

analytically inverted as illustrated by the following expressions:

$$\begin{aligned} \mathcal{M} &= \mathrm{H}\phi\mathrm{M}\phi^*\mathrm{H}^* \\ &= [\mathrm{I}+\mathrm{H}\phi\mathcal{K}]\mathcal{D}[\mathrm{I}+\mathrm{H}\phi\mathcal{K}]^* \\ [\mathrm{I}+\mathrm{H}\phi\mathcal{K}]^{-1} &= \mathrm{I}-\mathrm{H}\psi\mathcal{K} \\ \mathcal{M}^{-1} &= [\mathrm{I}-\mathrm{H}\psi\mathcal{K}]^*\mathcal{D}^{-1}[\mathrm{I}-\mathrm{H}\psi\mathcal{K}] \end{aligned} \tag{2}$$

The above expressions use just one additional operator $\mathcal{P}$ referred to as the articulated body inertia operator, and the derived operators $\mathcal{D} = \mathrm{H}\mathcal{P}\mathrm{H}^*$, $\mathcal{G} = \mathcal{P}\mathrm{H}\mathcal{D}^{-1}$, $\mathcal{K} = \mathcal{E}_\phi\mathcal{G}$, $\mathcal{E}_\psi = \mathcal{E}_\phi(\mathrm{I}-\mathcal{G}\mathrm{H})$ and $\psi = (\mathrm{I}-\mathcal{E}_\psi)^{-1}$. Eq. 2 illustrates both the expressiveness of the operator vocabulary, as well as the types of analysis that can be carried out with them. As is typical of such operator based structural properties, the Eq. 2 analytical expression for the mass matrix inverse is very general and holds for branched-topology systems of arbitrary size. Moreover, the bodies can be rigid as well as flexible. The constraint embedding technique [2] extends these tree-topology operator techniques to closed-loop graph systems as well [4].

While such analytical structure of system level dynamics properties can be studied via operators, another benefit of the operator expressions lies in the fact that these expressions directly lead to low-order recursive computational algorithms. The adjacency matrix structure of the $\mathcal{E}_\phi$ and $\mathcal{E}_\psi$ operators allows expressions such as ϕx, $\phi^* x$, ψx and $\psi^* x$ to be carried out via $\mathrm{O}(\mathcal{N})$ recursive algorithms without requiring the explicit computation of ϕ or ψ! The well known $\mathrm{O}(\mathcal{N})$ articulated body recursive algorithm for solving the equations of motion is a direct consequence of using this property with the Eq. 2 expression for the mass matrix inverse. Similarly, low-order algorithms for the operational space inertia, mass matrix sensitivity etc have been derived using spatial operators [2]. Figure 1 illustrates the use of SOA as a general purpose framework for supporting the analysis and algorithmic

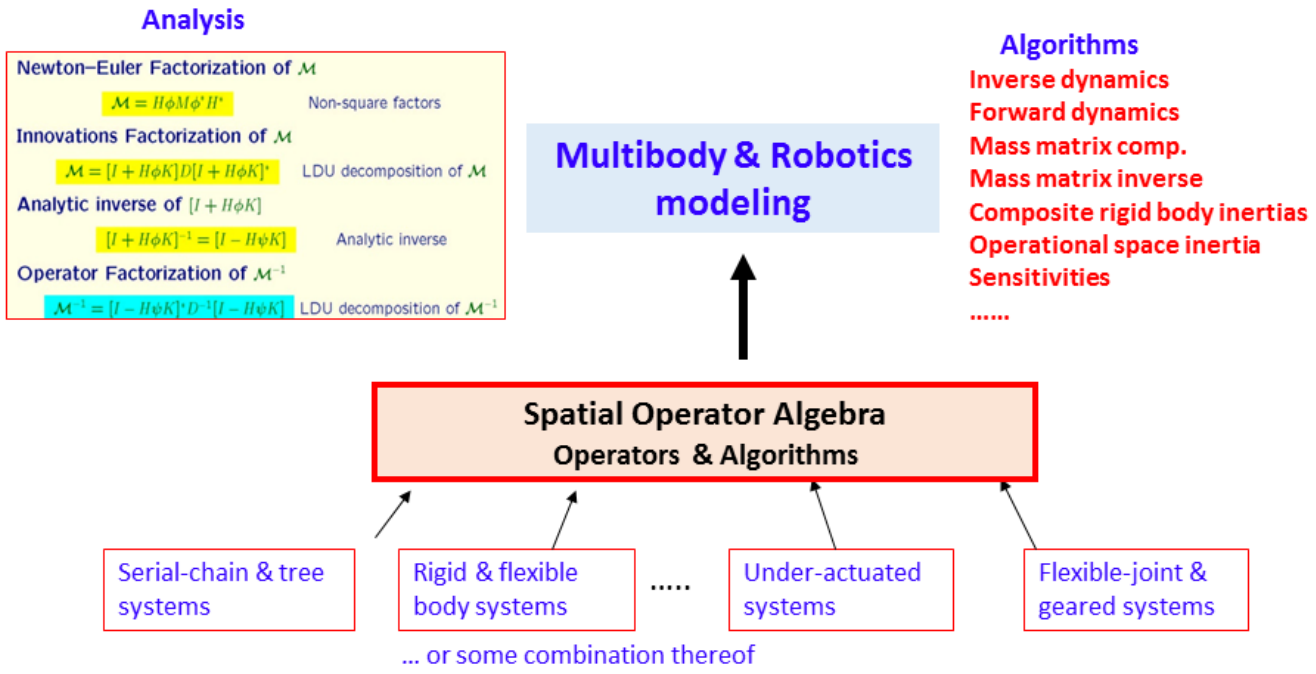

Fig. 1: The SOA operators provides a mathematical vocabulary for addressing analysis and algorithmic needs for a broad class of multibody systems.

needs for variety of classes of multibody systems.

We have developed an object-oriented toolbox called PyCraft in C++/Python for the computation of system level dynamics properties using the operator notation. PyCraft implements C++ classes for spatial operators that can be used for the computational evaluation of complex system dynamics properties described via mathematical operator expressions such as the above. A Python interface to the C++ classes allows the interactive, command-line evaluation of these operator expressions for a multibody model. Thus for instance, with 'g' representing

the C++/Python multibody model object, the mass matrix and its inverse in Eq. 1 and Eq. 2 can be numerically evaluated from the Python command line using the following Python statements:

```
>>> MassMatrix = H(g)*Phi(g)*M(g)*PhiStar(g)*HStar(g)
>>> InnovFactorInv = Id(g)-H(g)*Psi(g)*K(g)
>>> MassmatInv = InnovFactorInv.getTranspose()*Dinv(g)*InnovFactorInv
>>> assert(MassMat*MassmatInv).isIdentity()
```

Note the similarity between the mathematical operator expressions and the PyCraft command line expression syntax. Each of the `H(.)` etc represent classes for the corresponding SOA operator. The family of operator classes can be used for any desired system level dynamics computation.

The PyCraft implementation is general enough where '`g`' can in fact be any connected sub-tree of the multibody system. This can be useful for example for analyzing the dynamics of individual limbs in multi-limb robots. The overloaded '`*`' operations among the spatial operator objects take advantage of the structure-based low-order recursive algorithms that are applicable for efficient computation.

This paper provides an overview of the operator based analytical approach and the computational architecture of PyCraft for evaluating system dynamics properties and several examples of its use. This approach is especially powerful since the operator expressions remain unchanged across the entire class of tree-topology systems. We believe that PyCraft will provide an easy way to bridge the gap between theory and computation and thus support analytical and algorithm development and validation for multibody systems.

Section 2 reviews spatial operators and introduces some basic ones needed to define the mass matrix. Section 3 describes C++ and Python classes for these operators that allows the use of the operators interactively, and via scripts. Having established the basic concepts of the spatial operators and the PyCraft software architecture, Section 4 describes additional spatial operators and their use for more advanced computations. Section 5 describes the further application to the sensitivities of dynamics quantities.

2 Spatial Operators

For notational simplicity, we will focus our exposition of spatial operators on a serial chain system with n rigid bodies and $\mathcal{N}$ degrees of freedom. All the concepts however do extend to general tree systems. To introduce spatial operators, we begin with the following base-to-tip recursion that can be used to compute the $\mathcal{V}(k)$ spatial velocities for each of the bodies.

$$\begin{cases} \mathcal{V}(n+1) = 0 \\ \textbf{for } k = n \cdots 1 \\ \quad \mathcal{V}(k) = \phi^*(k+1,k)\mathcal{V}(k+1) + H^*(k)\,\dot{\theta}(k) \\ \textbf{end loop} \end{cases} \tag{3}$$

In the above, $\phi^*(k+1,k)$ denotes the rigid body transformation matrix between the $(k+1)^{th}$ and k^{th} bodies, and $H^*(k)$ the joint map matrix for the k^{th} body:

$$\phi(x,y) \triangleq \begin{pmatrix} I_3 & \widetilde{l}(x,y) \\ \mathbf{0}_3 & I_3 \end{pmatrix} \in \mathcal{R}^{6\times 6} \quad \text{and} \quad H^*(k) = \begin{bmatrix} h(k) \\ \mathbf{0} \end{bmatrix} \in \mathcal{R}^6 \tag{4}$$

The $\widetilde{\ }$ operation denotes the conversion of a 3-vector into its 3×3 skew-symmetric cross-product matrix. Now we introduce stacked vectors required to define system level relationships. We begin by defining the stacked vectors $\mathcal{V}$ and θ as

$$\mathcal{V} \triangleq \text{col}\left\{\mathcal{V}(k)\right\}_{k=1}^{n} = \begin{bmatrix} \mathcal{V}(1) \\ \mathcal{V}(2) \\ \vdots \\ \mathcal{V}(n) \end{bmatrix} \in \mathcal{R}^{6n}, \text{ and } \theta \triangleq \text{col}\left\{\theta(k)\right\}_{k=1}^{n} = \begin{bmatrix} \theta(1) \\ \theta(2) \\ \vdots \\ \theta(n) \end{bmatrix} \in \mathcal{R}^{\mathcal{N}} \tag{5}$$

The $\mathcal{V}$ stacked vector consists of the component body-level $\mathcal{V}(k)$ spatial velocity vectors assembled into a single large vector. Correspondingly, the θ stacked vector consists of the component body-level $\theta(k)$ generalized coordinates assembled into a single large vector.

Continuing on, define the block-diagonal H spatial operator as

$$H \triangleq \text{diag}\left\{H(k)\right\}_{k=1}^{n} = \begin{pmatrix} H(1) & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & H(2) & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & H(n) \end{pmatrix} \in \mathcal{R}^{\mathcal{N}\times 6n} \tag{6}$$

Now define the strictly block lower-triangular spatial operator $\mathcal{E}_\phi$ as:

$$\mathcal{E}_\phi \triangleq \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \phi(2,1) & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \phi(3,2) & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \phi(n,n-1) & \mathbf{0} \end{pmatrix} \in \mathcal{R}^{6n\times 6n} \tag{7}$$

The $\mathcal{E}_\phi$ operator has special structure, and is in fact an example of a *spatial kernel operator (SKO)* defined and discussed in reference [2]. Briefly, the sparse structure of an SKO matrix is that of the adjacency matrix for the multibody topology where the elements of the matrix are square matrices. For the $\mathcal{E}_\phi$ matrix, the non-zero entries are the 6×6 $\phi(i,j)$ rigid body transformation matrices for connected bodies. The sparse structure of $\mathcal{E}_\phi$ in Eq. 7 is that for a serial chain topology system. For more general branched tree topology systems, there can be additional non-zero entries corresponding to the topology's adjacency matrix. The SKO nature of this operator is the essential basis for the rich analytical structure for operators as well as the family of low-cost recursive computational algorithms that follow from them.

Combining Eq. 3, Eq. 6 and Eq. 7, we obtain the relationship

$$\mathcal{V} = \mathcal{E}_\phi^* \mathcal{V} + H^*\ \dot{\theta} \tag{8}$$

Since $\mathcal{E}_\phi$ is an SKO operator, it is always nilpotent, and its 1-resolvent $(I-\mathcal{E}_\phi)$ is always invertible. Such inverses of the 1-resolvent of SKO operators are referred to as *spatial propagation operators (SPO)* operators [2]. The SPO operator corresponding to $\mathcal{E}_\phi$ is denoted $\phi \in \mathcal{R}^{6n\times 6n}$ and is given by

$$\phi \triangleq (I-\mathcal{E}_\phi)^{-1} = I + \mathcal{E}_\phi + \mathcal{E}_\phi^2 + \ \cdots\ + \mathcal{E}_\phi^{n-1} \tag{9}$$

Now that we have an expression for the inverse of $(I-\mathcal{E}_\phi)$, we use it to obtain the following explicit expression for the the $\mathcal{V}$ spatial velocity stacked vector:

$$\mathcal{V} \ \overset{8}{=}\ (I-\mathcal{E}_\phi^*)^{-1} H^*\ \dot{\theta} \ \overset{9}{=}\ \phi^* H^*\ \dot{\theta} \tag{10}$$

This is a system level relationship using spatial operators corresponding to the component level one in Eq. 5. Note that from the Eq. 5 recursion it is clear that all of the body spatial velocities can be computed in a base to tips recursion sequence with $O(\mathcal{N})$ computational cost. However, the $\mathcal{V} = \phi^* H^*\ \dot{\theta}$ expression in Eq. 10 suggests the higher $O(\mathcal{N}^2)$ cost since it involves a matrix/vector product. The fact that this is not so, is due to the special property of SPO operators where products such as ϕx and $\phi^* x$, where x is a stacked vector, can always be carried out by $O(\mathcal{N})$ *tip-to-base (gather)* and *base-to-tip (scatter)* recursions respectively!

2.1 Jacobian matrix

The Jacobian matrix [5] provides a mapping from the generalized velocities to the spatial velocities of one or more nodes in the multibody system and plays an important role in robotics. The spatial velocity of node $\mathbb{O}_k^0$ on the k^{th} body is given by

$$\mathcal{V}\left(\mathbb{O}_k^0\right) = \phi^*(k, \mathbb{O}_k^0)\mathcal{V}(k) \tag{11}$$

In the stacked notation, we define the pickoff operator $\mathcal{B}$ to map the link spatial velocities into the node velocities as follows:

$$\mathcal{V}_{nd} \overset{11}{=} \mathcal{B}^*\mathcal{V} \quad \text{where} \quad \mathcal{B} \overset{\triangle}{=} \begin{bmatrix} \mathbf{0} \\ \vdots \\ \phi(k, \mathbb{O}_k^0) \\ \vdots \\ \mathbf{0} \end{bmatrix} \in \mathcal{R}^{6n \times 6} \tag{12}$$

Thus

$$\mathcal{V}_{nd} = \mathcal{J}\,\dot{\theta} \quad \text{where} \quad \mathcal{J} \overset{\triangle}{=} \mathcal{B}^*\phi^*H^* \in \mathcal{R}^{6n_{nd} \times \mathcal{N}} \tag{13}$$

The above describes the analytical expression for the $\mathcal{J}$ Jacobian in terms of spatial operators.

2.2 Mass matrix

With $m(k)$, $p(k)$ and $\mathscr{J}(k)$ denoting the mass, first moment and second moment of inertia respectively for the k^{th} body, the spatial inertia for the body is defined as

$$M(k) \overset{\triangle}{=} \begin{pmatrix} \mathscr{J}(k) & m(k)\widetilde{p}(k) \\ -m(k)\widetilde{p}(k) & m(k)I \end{pmatrix} \in \mathcal{R}^{6\times 6} \tag{14}$$

Once again, stacking up these quantities for all the bodies leads to the block-diagonal, symmetric and positive semi-definite $\mathcal{M}$ **spatial inertia** spatial operator defined as

$$\mathcal{M} \overset{\triangle}{=} \text{diag}\left\{M(k)\right\}_{k=1}^{n} \in \mathcal{R}^{6n\times 6n} \tag{15}$$

Using these spatial operators, it can be shown using SOA methods that the system mass matrix $\mathcal{M}$ has the following operator product form [2]:

$$\mathcal{M}(\theta) = H\phi M\phi^*H^* \in \mathcal{R}^{\mathcal{N}\times\mathcal{N}} \tag{16}$$

This factored form of the mass matrix is referred to as the **Newton–Euler Operator Factorization** of the mass matrix [2]. The mass matrix features in the system level equations of motion that has the following form:

$$\mathcal{M}\ddot{\theta} + \mathcal{C} = \mathcal{T} \quad \text{where} \quad \mathcal{C} = H\phi(\mathfrak{b} + M\phi^*\mathfrak{a}) \tag{17}$$

Here $\mathcal{C}$ denotes the overall vector of Coriolis, gyroscopic and gravitational terms, while $\mathfrak{b}$ and $\mathfrak{a}$ denote the stacked vectors of the link level velocity dependent terms.

3 PyCraft Software C++ Classes

Having described spatial operator expressions for some o the system level quantities, we now begin a discussion of the object oriented PyCraft software whose purpose is to allow the evaluation and computation of spatial operator expressions. PyCraft builds upon the DARTS software [6] for solving the equations of motion using minimal coordinates representation and algorithms for the multibody system. PyCraft makes use of the DARTS class instance `'g'` for the underlying multibody system.

The following describes the base classes for stacked vectors and spatial matrix operators.

`SpatialVector`**:** This is a base class for stacked vectors and consists of a map of body objects to vectors. The `SpatialVector V` is for the stacked vector of spatial velocities $\mathcal{V}$. `V[o]` returns the 6-dimensional spatial velocity for body `o`.

`SpatialMatrixOperator`**:** This is a base class for all spatial operators, and consists of matrix elements indexed by row and column elements consisting of multibody body or node objects. Thus for example, the `SpatialMatrixOperator phi` denotes the ϕ spatial operator, and `phi[o][p]` returns the 6×6 $\phi(o,p)$ matrix for the o and p bodies.

`SpatialSquareMatrixOperator`**:** This class is a specialization of the `SpatialMatrixOperator` class, where the row and column indices are the same. Thus the operator is square in the indices though the element matrices are not required to be square.

`SpatialDiagonalMatrixOperator`**:** This class is a specialization of the `SpatialSquareMatrixOperator` class, where only the diagonal elements are non-zero.

The constructors for each of these classes take a multibody subgraph g instance argument. These matrix vector classes have the normal arithmetic "+", "*" etc operators available that follow the expected rules for taking products of matrices and matrix/vectors with compatible indices. These operations are only permitted among operators sharing the same subgraph. Since many spatial operators are sparse, these operators only store non-zero elements, and any entry not explicitly defined is assumed to be zero by default.

Using these base classes, we next describe derived PyCraft classes for the basic kinematics and mass properties of the g multibody system:

`IdClass` **:** This class derived from `SpatialDiagonalMatrixOperator` corresponds to the identity operator.

`HStarClass` **:** This is the operator class derived from `SpatialDiagonalMatrixOperator` for the H^* spatial operator. Additionally the `HClass` class derived from `HStarClass` is for instances of the H operator.

`MClass` **:** This operator class corresponds to M and is derived from `SpatialDiagonalMatrixOperator`.

The SKO operator for g defines kinematic properties as well as the topology of the system. As discussed earlier, closely associated with an SKO operator is its SPO operator. The following PyCraft classes are used for creating instances of the family of SKO and SPO operators:

SKO operator classes: `SKOClass` is the base class for SKO operators and is derived from `SpatialSquareMatrixOperator`. The `EPhiClass` operator for $\mathcal{E}_\phi$ is derived from the `SKOClass` class. The `SKOStarClass` class for the transpose of an `SKOClass` operator is derived from the `SpatialSquareMatrixOperator` class and takes the `SKOClass` instance as a constructor argument.

SPO operator classes: `SPOClass` is the base class for SPO operators and is derived from `SpatialSquareMatrixOperator`. Its constructor takes an `SKOClass` operator instance argument. `SPOStarClass` is the class for the transpose of SPO operators and is derived from `SpatialSquareMatrixOperator`. Its constructor takes an additional `SKOClass` operator instance argument.

The special property of SPO operators, that products such as ϕx and $\phi^* x$ with vectors can be carried out via low-cost $O(\mathcal{N})$ gather and scatter recursions are exploited by implementing `operator *` methods for the `SPOClass` and `SPOStarClass` classes which implement these low-cost algorithms for products with vectors.

The additional `BStarClass` is defined for the $\mathcal{B}^*$ pickoff operator. The constructor for this class takes a list of node instances as an additional argument.

3.1 Python Interface

For more convenient use, PyCraft includes a Python interface for the C++ classes described above. As illustrated in Figure 2, the Python interface is auto-generated using the SWIG tool [7], and exactly mirrors the C++ class API. The Python interfaces allows the use of the operators and their methods from the Python command line as well as from within scripts. Using the Python interface, the following sequence of Python statements creates instances of

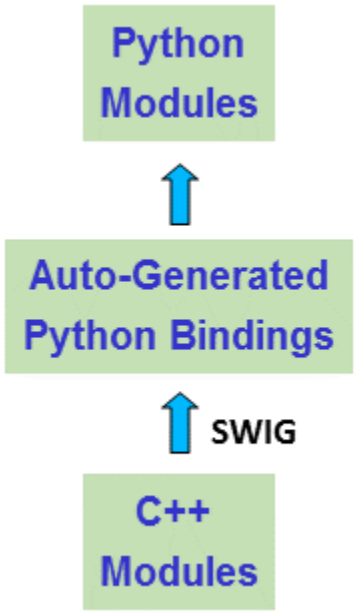

Fig. 2: PyCraft's Python bindings for the underlying C++ operator classes are auto-generated using the open source SWIG tool.

the basic operators that define the g multibody system's articulation, kinematic, topology and mass properties.

Listing 1: Basic operators for the multibody system

```
>>> H       = HClass(g)                  # instance of H operator (Eq. 6)
>>> ephi    = EPhiClass(g)               # instance of E_phi operator (Eq. 7)
>>> M       = MClass(g)                  # instance of M operator (Eq. 15)
```

As a next step, we define some additional operators, and use them to evaluate the system level Jacobian and mass matrix for the multibody system.

Listing 2: Evaluation of the Jacobian and the mass matrix $\mathcal{M}$

```
>>> Hst     = HStarClass(g)              # instance of H* operator (Eq. 6)
>>> phi     = SPOClass(ephi)             # instance of phi operator (Eq. 9)
>>> phistar = SPOStarClass(ephi)         # instance of phi* operator (Eq. 9)
>>> bstar   = BStarClass(g, nodes)       # instance of B operator (Eq. 12)
>>> Jac     = bstar * phistar * Hst      # evaluate the J Jacobian (Eq. 13)
>>> massmat = H * phi * M * phistar * Hst # evaluate the M mass matrix (Eq. 16)
>>> marray  = massmat.asArray()          # get M matrix content as an numpy array
```

The `SpatialMatrixOperator::asArray()` method used at the end returns a Python `numpy` array instance with the values for the mass matrix.

The *inverse dynamics* problem requires the evaluation of Eq. 17 to obtain $\mathcal{T}$ for specified $\ddot{\theta}$ generalized accelerations. Focusing on just the $\mathcal{M}\ddot{\theta}$ product for the moment, we can evaluate this product in PyCraft as follows:

Listing 3: Inverse dynamics computation

```
>>> T = H * phi * M * phistar * Hst * thetaddot # evaluate Mθ̈ for inverse dynamics (Eq. 17)
```

Since PyCraft implements SPO/vector products via $O(\mathcal{N})$ recursions, the above inverse dynamics computation ends up also being of $O(\mathcal{N})$ computational cost. In fact this procedure is precisely the well-known, optimal Newton-Euler inverse dynamics algorithm [8].

3.2 Mass matrix decomposition

Having introduced the basic spatial operators, we now look at examples of analysis that can be carried out using spatial operators, and the use of PyCraft for evaluating the resulting operator expressions.

Using SOA operator techniques, it can be shown that $\phi M \phi^*$ can be decomposed into the following sum [2]

$$\phi M \phi^* = \phi \mathcal{R} + \mathcal{R} \phi^* - \mathcal{R} \tag{18}$$

where $\mathcal{R} \triangleq \text{diag}\left\{\mathcal{R}(k)\right\}_{k=1}^{n} \in \mathcal{R}^{6n \times 6n}$ is a block-diagonal, symmetric, positive semi-definite operator with the $\mathcal{R}(k)$ block diagonal entries defined via the following tip-to-base gather recursion:

$$\begin{cases} \mathcal{R}(0) = \mathbf{0} \\ \textbf{for } k = 1 \cdots n \\ \quad \mathcal{R}(k) = \phi(k, k-1)\mathcal{R}(k-1)\phi^*(k, k-1) + M(k) \\ \textbf{end loop} \end{cases} \tag{19}$$

$\mathcal{R}(k)$ is the *composite body inertia* [9] for the k^{th} body and represents the spatial inertia of the k^{th}, and all its outboard bodies regarded as a single composite body.

The decomposition result in Eq. 18 is in fact a general result that applies to any operator product of the form $\mathbb{A}X\mathbb{B}^*$ where $\mathbb{A}$ and $\mathbb{B}$ are SPO operators and X is an arbitrary (but compatible) diagonal matrix. In this disjoint partitioning of $\phi M \phi^*$, the first term is lower triangular, the second term is upper triangular and the last term on the right hand side is block diagonal. Based on this decomposition and Eq. 16, the mass matrix can be decomposed into diagonal and triangular terms as follows:

$$\mathcal{M} = H\left(\phi \mathcal{R} + \mathcal{R} \phi^* - \mathcal{R}\right) H^* \tag{20}$$

At the operator level, it is easy to verify that $\mathcal{R}$ is in fact a solution to the following *forward Lyapunov* operator equation:

$$M = \mathcal{R} - \mathcal{E}_\phi \mathcal{R} \mathcal{E}_\phi^* \tag{21}$$

Due to this general property of SKO and SPO operators, the `SKOClass` class has the `SKOClass::LyapunovRecursion(SpatialDiagonalMatrixOperator)` method that returns the `SpatialDiagonalMatrixOperator` solution to the forward Lyapunov equation. In other words

Listing 4: Composite Rigid Body Inertia $\mathcal{R}$

```
>>> R = ephi.LyapunovRecursion(M)    # instance of ℛ operator (Eq. 19)
```

The procedure for computing the mass matrix based on the decomposition in Eq. 20 is the optimal $O(\mathcal{N}^2)$ composite rigid body inertia based algorithm for computing the mass matrix [9]. Our derivation using SOA techniques uses spatial operators to analyze and take advantage of the structure of the mass matrix to develop the low cost technique.

In PyCraft we can compute $\mathcal{M}$ via the brute force composition of operators shown in Listing Eq. 2. Alternatively we can use the Eq. 20 operator expression that uses the faster composite rigid body decomposition based algorithm as follows:

Listing 5: CRB decomposition of the mass matrix

```
>>> massmat = H * (phi * R + R * phistar - R) * Hst  # the mass matrix M (Eq. 16)
```

We can validate the correctness of this alternative method for computing the mass matrix by comparing it with the results from the procedure in Listing 2. Such cross-verification is an illustration of the use of PyCraft for the numerical validation of analytical results in a simple and easy manner. As seen earlier, the PyCraft statements very closely parallel the analytical operator expressions being evaluated. For all practical purposes, we are able to literally transcribe complex spatial operator expressions into a PyCraft environment in order to evaluate them.

4 Articulated Body Spatial Operators

Now define the block diagonal operator $\mathcal{P}$ as

$$\mathcal{P} \triangleq \text{diag}\left\{\mathcal{P}(k)\right\}_{k=1}^{n} \qquad \in \mathcal{R}^{6n\times 6n} \tag{22}$$

where the $\mathcal{P}(k)$ articulated body inertia elements are defined via the following *articulated body* tip-to-base gather recursion:

$$\left\{\begin{array}{rl} & \mathcal{P}^{+}(0)=\mathbf{0}, \quad \bar{\tau}(0)=\mathbf{0} \\ \textbf{for } k &= 1\cdots n \\ \psi(k,k-1) &= \phi(k,k-1)\bar{\tau}(k-1) \\ \mathcal{P}(k) &= \phi(k,k-1)\mathcal{P}^{+}(k-1)\phi^{*}(k,k-1)+M(k) \\ \mathcal{D}(k) &= H(k)\mathcal{P}(k)H^{*}(k) \\ \mathcal{G}(k) &= \mathcal{P}(k)H^{*}(k)\mathcal{D}^{-1}(k) \\ \mathcal{K}(k+1,k) &= \phi(k+1,k)\mathcal{G}(k) \\ \bar{\tau}(k) &= I-\mathcal{G}(k)H(k) \\ \mathcal{P}^{+}(k) &= \bar{\tau}(k)\mathcal{P}(k) \\ \textbf{end loop} & \end{array}\right. \tag{23}$$

Define the following additional spatial operators:

$$\begin{array}{lll} \mathcal{D} \triangleq \text{diag}\left\{\mathcal{D}(k)\right\}_{k=1}^{n} = H\mathcal{P}H^{*} & & \in \mathcal{R}^{\mathcal{N}\times\mathcal{N}} \\ \mathcal{G} \triangleq \text{diag}\left\{\mathcal{G}(k)\right\}_{k=1}^{n} = \mathcal{P}H^{*}\mathcal{D}^{-1} & & \in \mathcal{R}^{6n\times\mathcal{N}} \\ \mathcal{K} \triangleq \mathcal{E}_{\phi}\mathcal{G} & & \in \mathcal{R}^{6n\times\mathcal{N}} \\ \tau \triangleq \text{diag}\left\{\tau(k)\right\}_{k=1}^{n} = \mathcal{G}H & & \in \mathcal{R}^{6n\times 6n} \\ \bar{\tau} \triangleq \text{diag}\left\{\bar{\tau}(k)\right\}_{k=1}^{n} = I-\tau & & \in \mathcal{R}^{6n\times 6n} \\ \mathcal{P}^{+} \triangleq \text{diag}\left\{\mathcal{P}^{+}(k)\right\}_{k=1}^{n} = \bar{\tau}\mathcal{P}\bar{\tau}^{*} = \bar{\tau}\mathcal{P} = \mathcal{P}\bar{\tau}^{*} & & \in \mathcal{R}^{6n\times 6n} \\ \mathcal{E}_{\psi} \triangleq \mathcal{E}_{\phi}\bar{\tau} & & \in \mathcal{R}^{6n\times 6n} \end{array} \tag{24}$$

The $\mathcal{P}$ operator is in fact a solution to the following *Riccati equation*:

$$M = \mathcal{P} - \mathcal{E}_\psi \mathcal{P} \mathcal{E}_\psi^* = \mathcal{P} - \mathcal{E}_\phi (I - \mathcal{P} H^* \underbrace{(H \mathcal{P} H^*)^{-1}}_{\mathcal{D}} H) \mathcal{P} \mathcal{E}_\phi^* + M \tag{25}$$

$\mathcal{G}$

τ

$\bar{\tau}$

$\mathcal{P}^+$

The structure of $\mathcal{E}_\psi$ is identical to that of $\mathcal{E}_\phi$, and the pair only differ in the actual value of the non-zero elements. Indeed, $\mathcal{E}_\psi$ is an SKO operator in its own right. Its corresponding SPO operator is $\psi = (I - \mathcal{E}_\psi)^{-1}$. The structure of ψ is identical to that of ϕ and the pair only differ in the values of their elements.

The new articulated body operator class in PyCraft is `PClass` derived from `SpatialDiagonalMatrixOperator` for $\mathcal{P}$, with diagonal elements defined by the Riccati equation recursion. Thus

Listing 6: Articulated body operator $\mathcal{P}$

```
>>> P        = PClass(g)          # instance of P operator (Eq. 22)
```

The other articulated body operators can be evaluated via the following PyCraft statements:

Listing 7: Articulated body operators

```
>>> D        = H * P * Hst        # evaluate D (Eq. 24)
>>> Dinv     = D.inverse()        # evaluate D^-1 (Eq. 24)
>>> G        = P * Hst * Dinv     # evaluate G (Eq. 24)
>>> K        = ephi * G           # evaluate K (Eq. 24)
>>> tau      = G * H              # evaluate tau (Eq. 24)
>>> Id       =  IdClass(g)        # the identity operator
>>> taubar   = Id - tau           # evaluate taubar (Eq. 24)
```

Except for `K`, all of the operator instances above are block-diagonal. The $\mathcal{E}_\psi$ operator has its own class `EPsiClass` derived from the `SKOClass` class with elements from the `(ephi * taubar)` product.

Listing 8: The ψ family of SKO ans SPO operators

```
>>> epsi          =  EPsiClass(g)          # instance of E_psi operator (Eq. 24)
```

Since $\mathcal{E}_\psi$ is an SKO operator, we can define the corresponding SPO and other related operators for it as follows:

Listing 9: The ψ family of SKO and SPO operators

```
>>> psi           =  SPOClass(epsi)        # instance of psi = (I - E_psi)^-1 operator
>>> psistar       =  SPOStarClass(epsi)    # instance of psi^* operator
```

`psi` is structurally very similar to `phi` with differences being in the specific values of the non-zero elements.

4.1 Example operator identities

One of the advantages of spatial operators is their rich mathematical structure which allows us to derive new analytical relationships and simplifications of complex expressions. The following operator identities are illustrative examples of this [2]:

1.

$$\begin{aligned}[I-H\psi\mathcal{K}]H\phi &= H\psi \\ \phi\mathcal{K}[I-H\psi\mathcal{K}] &= \psi\mathcal{K} \\ [I+H\phi\mathcal{K}]H\psi &= H\phi \\ \psi\mathcal{K}[I+H\phi\mathcal{K}] &= \phi\mathcal{K}\end{aligned} \tag{26}$$

2.

$$H\psi M\psi^* H^* = \mathcal{D} \tag{27}$$

Notice the strong similarity between the expression in Eq. 27 and the expression for the mass matrix in Eq. 16. While we are able to mathematically establish these identities, the PyCraft classes allow us to actually compute and verify them as follows:

Listing 10: Operator identity

```
>>> assert ((Id - H * psi * K) * H * phi - H * psi).isZero() # verify Eq. 26 identity
>>> assert (D - H * psi * M * psistar * Hst).isZero()        # verify Eq. 27 identity
```

Once again we see that it is easy to transcribe and execute mathematical expressions within PyCraft.

4.2 Mass matrix inversion

One of the seminal results from the SOA spatial operator methodology has been that the mass matrix can be analytically factorized, and even inverted for arbitrary tree systems. This first step towards this consists of yet another factorization of the mass matrix shown below [2]:

$$\mathcal{M} = [I+H\phi\mathcal{K}]\mathcal{D}[I+H\phi\mathcal{K}]^* \tag{28}$$

This **Innovations factorization** has the advantage that its factors are square. We can validate this analytical expression by comparing the numerical values with the values obtained from either of the two earlier procedures for computing the mass matrix.

Listing 11: Innovations factorization of the mass matrix

```
>>> InFac   = Id + H * phi * K                # Innovations factor [I+HφK] (Eq. 28)
>>> massmat = InFac * D * InFac.getTranspose() # Innovations factorization of M (Eq. 28)
```

The next operator level result is the following analytical expression for the inverse of the $[I+H\phi\mathcal{K}]$ factor [2]:

$$[I+H\phi\mathcal{K}]^{-1} = I-H\psi\mathcal{K} \tag{29}$$

Once again, this relationship can be evaluated and verified easily within PyCraft as follows:

Listing 12: Innovations factor inverse

```
>>> InFacInv = Id - H * psi * K             # evaluate [I-HψK]
>>> assert (InFac * InFacInv).isIdentity()  # verify Eq. 29 identity
```

Putting together the above pair of results, immediately leads to the following closed form expression for the mass matrix inverse:

$$\mathcal{M}^{-1} = [I-H\psi\mathcal{K}]^*\mathcal{D}^{-1}[I-H\psi\mathcal{K}] \tag{30}$$

The PyCraft steps for evaluating this expression are simply:

Listing 13: Mass matrix inverse

```
>>> massmatInv = InFacInv.getTranspose() * Dinv * InFacInv # evaluate M^-1 (Eq. 30)
>>> assert (massmat * massmatInv).isIdentity()     # verify that this is inverse of M
```

4.3 Solving the equations of motion

Solving the equations of motion is one of the key problems in simulating system dynamics. The $O(\mathcal{N})$ articulated body forward dynamics algorithm is based on the expression for the mass matrix inverse derived in Eq. 30. Using this in Eq. 17 leads to the following operator expression for $\ddot{\theta}$:

$$\ddot{\theta} = \mathcal{M}^{-1}(\mathcal{T}-\mathcal{C}) = [I-H\psi\mathcal{K}]^* \mathcal{D}^{-1}[\mathcal{T}-H\underbrace{\psi(\mathcal{K}\mathcal{T}+\mathcal{P}\mathfrak{a}+\mathfrak{b})}_{\mathfrak{z}}]-\mathcal{K}^*\psi^*\mathfrak{a} \qquad \underbrace{\qquad\qquad\qquad}_{\epsilon} \qquad \underbrace{\qquad\qquad\qquad\qquad}_{\nu} \tag{31}$$

Due to the SPO nature of ψ, the matrix/vector products in the above expression can be carried out recursively at $O(\mathcal{N})$ cost, and forms the basis for the optimal $O(\mathcal{N})$ forward dynamics algorithms. Once again, the above algorithm can be evaluated within PyCraft via the following:

Listing 14: $O(\mathcal{N})$ AB forward dynamics

```
>>> z         = psi * (K * T + P * a + b)              # evaluate z  (Eq. 31)
>>> eps       = T - H * z                              # evaluate ϵ  (Eq. 31)
>>> nu        = Dinv * eps                             # evaluate ν  (Eq. 31)
>>> alpha     = psistar * (Hst * nu + a)               # evaluate α  (Eq. 31)
>>> thetaddot = nu - G.getTranspose() * ephistar * alpha # evaluate θ̈  (Eq. 31)
```

4.4 Operational Space Inertia

The operational space inertia is an important quantity for robotics control [10]. It is defined via its inverse which is given by

$$\underline{\Lambda} \triangleq \mathcal{J}\mathcal{M}^{-1}\mathcal{J}^* \quad \in \mathcal{R}^{6n_{nd}\times 6n_{nd}} \tag{32}$$

Quantities such as the operational space inertia also show up when solving closed-chain and contact dynamics equations of motion [2]. Recalling the $\mathcal{J} = \mathcal{B}^*\phi^*H^*$ Jacobian matrix expression from Eq. 13 leads to the following expression for $\underline{\Lambda}$:

$$\underline{\Lambda} \overset{32}{=} \mathcal{J}\mathcal{M}^{-1}\mathcal{J}^* \overset{13}{=} \mathcal{B}^*\phi^*H^*(I-H\psi\mathcal{K})^*\mathcal{D}^{-1}(I-H\psi\mathcal{K})H\phi\mathcal{B} \tag{33}$$

The use of the $(I-H\psi\mathcal{K})H\phi = H\psi$ identity from Eq. 26 results in the following simpler expression:

$$\underline{\Lambda} = \mathcal{B}^*\Omega\mathcal{B}, \quad \text{where} \quad \Omega \triangleq \psi^*H^*\mathcal{D}^{-1}H\psi \ \in \mathcal{R}^{6n\times 6n} \tag{34}$$

For serial chain systems, SOA spatial operators can be used to show that the operational space compliance matrix Ω can be decomposed into the following sum of component terms [2]:

$$\Omega = \psi^*\Upsilon + \Upsilon\psi - \Upsilon \tag{35}$$

where Υ is a block diagonal operator whose diagonal elements are defined by the following base-to-tip scatter recursion:

$$\Upsilon(k) = \psi^*(k+1,k)\Upsilon(k+1)\psi(k+1,k) + H^*(k)\mathcal{D}^{-1}(k)H(k) \tag{36}$$

It is easy to verify that the above recursion is equivalent to Υ being a solution to the following *backward Lyapunov equation*:

$$H^* \mathcal{D}^{-1} H = \Upsilon - \mathcal{E}_\psi^* \Upsilon \mathcal{E}_\psi \tag{37}$$

This backwards Lyapunov equation is the dual of the forward Lyapunov equation encountered earlier for SKO operators. The importance of the decomposition in Eq. 35 is that it provides a much lower cost algorithm for computing Ω and the operational space inertia compared with the brute force method for evaluating via Eq. 32 [2, 11]. This is one more example of the use of SOA spatial operators based analysis to understand the structure of complex dynamics quantities to generate simpler expressions as well as lower-cost computational algorithms.

Analogous to the `SKOClass::LyapunovRecursion()` method for the forward Lyapunov equation in PyCraft, the `SKOStarClass::LyapunovRecursion()` method returns the solution to the backward Lyapunov equation as follows:

Listing 15: Backward Lyapunov recursion for Υ

```
>>> Y = epsistar.LyapunovRecursion(Hst * Dinv * H)    # instance of Υ operator (Eq. 36)
```

Using this, it is easy to verify the expression for the operational space compliance matrix decomposition in PyCraft via:

Listing 16: Decomposition of the operational space compliance matrix

```
>>> Omega = psistar * Hst * massmatInv * H * psi          # evaluate Ω (Eq. 34)
>>> assert (Omega - (psist * Y + Y * psi - Y)).isZero()   # verify Eq. 35 identity
```

A further application of the operational space inertia related decomposition is in deriving an operator decomposition of the mass matrix inverse $\mathcal{M}^{-1}$ and using it to develop a lower cost computational algorithm for its evaluation. The reader is referred to reference [2] for further details.

5 Operator sensitivities

Now we turn to the topic of computing gradients of various dynamics quantities. We do not have to look too far to see a need for these, since even the Coriolis vector $\mathcal{C}$ in the equations of motion can be shown to be defined via *Christoffel symbols of the first kind* which are defined in terms of the gradients of the mass matrix elements as follows [2][1]:

$$\mathfrak{C}_i(j,k) \triangleq \frac{1}{2}\left[\frac{\partial \mathcal{M}(i,j)}{\partial \theta(k)} + \frac{\partial \mathcal{M}(i,k)}{\partial \theta(j)} - \frac{\partial \mathcal{M}(j,k)}{\partial \theta(i)}\right] \quad \text{for} \quad i,j,k = 1 \cdots n \tag{38}$$

For notational simplicity, the operator analysis in this section assumes that the multibody system has rotational, single degree of freedom hinges. The more general treatment is described in reference [2]. We begin with defining the following matrix related to the $H^*(k)$ joint map matrix:

$$\widetilde{H}^*(k) = \begin{pmatrix} \widetilde{h}(k) & 0 \\ 0 & \widetilde{h}(k) \end{pmatrix} \in \mathcal{R}^{6\times 6} \tag{39}$$

Using this component level, matrix, for the i^{th} body define the following block-diagonal operators $\widetilde{\mathcal{H}}^\omega_{\preceq i}$, $\widetilde{\mathcal{H}}^\omega_{\prec i}$ and $\widetilde{\mathcal{H}}^\omega_{=i}$ that are very closely related to the H^* joint map operator.

$$\begin{aligned} \widetilde{\mathcal{H}}^\omega_{\preceq i} &= \text{diag}\left\{\widetilde{H}^*(k)\mathbb{1}_{[k\preceq i]}\right\}_{k=1}^n \\ \widetilde{\mathcal{H}}^\omega_{\prec i} &= \text{diag}\left\{\widetilde{H}^*(k)\mathbb{1}_{[k\prec i]}\right\}_{k=1}^n \\ \widetilde{\mathcal{H}}^\omega_{=i} &= \text{diag}\left\{\widetilde{H}^*(k)\mathbb{1}_{[k=i]}\right\}_{k=1}^n \end{aligned} \tag{40}$$

[1] The notation, $[ij,k]$, is also often used for $\mathfrak{C}_k(i,j)$ in the literature for Christoffel symbols of the first kind.

$\widetilde{\mathcal{H}}^{\omega}_{=i}$ has only a single non-zero entry along the diagonal at the i^{th} slot, while $\widetilde{\mathcal{H}}^{\omega}_{\prec i}$ has non-zero elements from the first to the i^{th} (but not including the i^{th} position), while $\widetilde{\mathcal{H}}^{\omega}_{\preceq i}$ has non-zero elements from the first to the i^{th} position.

We can now develop operator expression for the partial derivatives of spatial operators with the goal of being able to analytically compute the gradient of the mass matrix [2]. Assuming that we are taking the gradient with respect to to the i^{th} coordinate, the following describe operator expressions for the gradients of basic spatial operators[2]:

$$[\mathcal{E}_\phi]_{\theta_i} = \widetilde{\mathcal{H}}^{\omega}_{\preceq i}\,\mathcal{E}_\phi - \mathcal{E}_\phi\,\widetilde{\mathcal{H}}^{\omega}_{\prec i} \tag{41a}$$

$$[\phi]_{\theta_i} = \phi\,\widetilde{\mathcal{H}}^{\omega}_{\prec i}\,\phi - \phi\,\widetilde{\mathcal{H}}^{\omega}_{\preceq i} \tag{41b}$$

$$[H^*]_{\theta_i} = \widetilde{\mathcal{H}}^{\omega}_{\prec i}\,H^* \tag{41c}$$

$$[M]_{\theta_i} = \widetilde{\mathcal{H}}^{\omega}_{\preceq i}\,M - M\,\widetilde{\mathcal{H}}^{\omega}_{\preceq i} \tag{41d}$$

Going further, the sensitivity of the mass matrix with respect to a generalized coordinate can be analytically shown to be [2]:

$$\mathcal{M}_{\theta_i} = H\phi\left[\widetilde{\mathcal{H}}^{\omega}_{=i}\,\phi M - M\phi^*\,\widetilde{\mathcal{H}}^{\omega}_{=i}\right]\phi^* H^* \tag{42}$$

This expression is structurally strikingly similar to the expression for the mass matrix itself. PyCraft implements the classes `H_iClass` , `Hs_iClass` and `Hd_iClass` derived from the `HStarClass` class for the $\widetilde{\mathcal{H}}^{\omega}_{\preceq i}$, $\widetilde{\mathcal{H}}^{\omega}_{\prec i}$ and $\widetilde{\mathcal{H}}^{\omega}_{=i}$ operators respectively. The constructors for these classes take the body object argument with respect to whose coordinates the sensitivity computations are desired. Thus assuming that we are taking derivatives with respect to the coordinates of the third body we create the following instances in PyCraft:

Listing 17: Sensitivity related basic classes

```
>>> Hi_3  =  H_iClass(g, bd3)   # instance of H^ω_≼i operator
>>> Hsi_3 = Hs_iClass(g, bd3)   # instance of H^ω_≺i operator
>>> Hdi_3 = Hd_iClass(g, bd3)   # instance of H^ω_=i operator
```

Using these we can evaluate basic operator sensitivities as follows:

Listing 18: Derivative of basic spatial operators

```
>>> phi_3  = Hdi_3 * phi * Hdi_3 - phi * Hdi_3   # evaluate [φ]_θi (Eq. 41a)
>>> M_3    = Hdi_3 * M - M * Hdi_3               # evaluate [M]_θi (Eq. 41d)
```

The gradient of the mass matrix can be evaluated in PyCraft as follows:

Listing 19: Derivative of the mass matrix

```
# evaluate M_θ3 (Eq. 42)
>>> massmat_3 = H * phi * ( Hdi_3 * phi * M - M * phistar * Hdi_3) * phistar * Hst
```

We can verify that the various analytical expressions for the gradients are indeed correct by comparing with numerically computed derivatives of the operators and other quantities using small perturbations of the coordinate value.

[2]The notation $[X]_{\theta_i}$ denotes $\frac{\partial X}{\partial \theta_i}$.

5.1 Articulated body sensitivities

We now take the story further by focusing on the gradients of the articulated body inertia operators discussed in Section 4. For this we introduce a new diagonal operator $\check{\lambda}_{\theta_i}$ as the solution to the following forward Lyapunov equation for $\mathcal{E}_\psi$:

$$\check{\lambda}_{\theta_i} - \mathcal{E}_\psi \check{\lambda}_{\theta_i} \mathcal{E}_\psi^* = \widetilde{\mathcal{H}}^\omega_{=i} \mathcal{P} - \mathcal{P} \widetilde{\mathcal{H}}^\omega_{=i} \tag{43}$$

As seen earlier, the solutions to the forward Lyapunov equation can be computed via a gather tip-to-base recursion analogous to the one in Eq. 19. We continue to assume that we are taking gradients with respect to to the third body's coordinate, and in this case the following PyCraft expression creates an instance of $\check{\lambda}_{\theta_3}$:

Listing 20: Forward Lyapunov solution for $\check{\lambda}_{\theta_3}$

```
>>> lambda_3 = epsi.LyapunovRecursion(Hdi_3 * P - P * Hdi_3)     # evaluate λ̌_θ3 (Eq. 43)
```

The new $\check{\lambda}_{\theta_3}$ spatial operator can be used to derive analytical expressions for the sensitivity of articulated body quantities [2] as shown below:

$$\begin{aligned}
\mathcal{D}_{\theta_i} &= \mathrm{H} \check{\lambda}_{\theta_i} \mathrm{H}^* && (44a)\\
[\mathcal{D}^{-1}]_{\theta_i} &= -\mathcal{D}^{-1} \mathrm{H} \check{\lambda}_{\theta_i} \mathrm{H}^* \mathcal{D}^{-1} && (44b)\\
[\mathcal{G}]_{\theta_i} &= \bar{\tau} \check{\lambda}_{\theta_i} \mathrm{H}^* \mathcal{D}^{-1} + \widetilde{\mathcal{H}}^\omega_{\prec i} \mathcal{G} && (44c)\\
[\tau]_{\theta_i} &= \bar{\tau} \check{\lambda}_{\theta_i} \mathrm{H}^* \mathcal{D}^{-1} \mathrm{H} + \widetilde{\mathcal{H}}^\omega_{\prec i} \tau - \tau \widetilde{\mathcal{H}}^\omega_{\prec i} && (44d)\\
\bar{\tau}_{\theta_i} &= -[\tau]_{\theta_i} && (44e)\\
[\mathcal{E}_\psi]_{\theta_i} &= \widetilde{\mathcal{H}}^\omega_{\preceq i} \mathcal{E}_\psi - \mathcal{E}_\psi \widetilde{\mathcal{H}}^\omega_{\prec i} - \mathcal{E}_\psi \check{\lambda}_{\theta_i} \mathrm{H}^* \mathcal{D}^{-1} \mathrm{H} && (44f)
\end{aligned}$$

Once again, these expressions can be directly transcribed into PyCraft for evaluation as illustrated below:

Listing 21: Evaluation of articulated body sensitivities

```
>>> D_3 = H * lambda_3 * Hst                              # evaluate D_θ3 (Eq. 43)
>>> G_3 = taubar  * lambda_3 * Hst * Dinv + Hs_3 * G      # evaluate G_θ3 (Eq. 43)
```

5.2 Mass matrix factor sensitivities

Building upon the articulated body sensitivities, the following are analytical expressions for the Innovations factors of the mass matrix and its inverse [2]:

$$[\mathrm{I} + \mathrm{H}\phi\mathcal{K}]_{\theta_i} = \mathrm{H}\phi \left[\widetilde{\mathcal{H}}^\omega_{=i} \tilde{\phi} \mathcal{P} + \bar{\tau} \check{\lambda}_{\theta_i} \right] \mathrm{H}^* \mathcal{D}^{-1} \tag{45}$$

$$[\mathrm{I} - \mathrm{H}\psi\mathcal{K}]_{\theta_i} = -\mathrm{H}\psi \left[\widetilde{\mathcal{H}}^\omega_{=i} \phi\mathcal{K} + \bar{\tau} \check{\lambda}_{\theta_i} \mathrm{H}^* \mathcal{D}^{-1} \right] (\mathrm{I} - \mathrm{H}\psi\mathcal{K}) \tag{46}$$

The PyCraft implementation of these sensitivity expressions is as follows:

Listing 22: Sensitivity of mass matrix factors

```
# evaluate [I + HφK]_θ3 (Eq. 45)
>>> InFac_3 = H * phi * ( Hd_3 * phitilde * P + taubar * lambda_3 ) * Hst * Dinv


# evaluate [I - HψK]_θ3 (Eq. 46)
>>> InFacInv_3 = -H * psi * ( Hd_3 * phi * K + taubar * lambda_3 * Hst * Dinv ) * InFacInv
```

The Innovations factor related sensitivities play an important role in the development of diagonalized dynamics models for the system [2, 12].

5.3 Fixman potential

As a final example, we look at the problem of computing the Fixman potential which arises in the context of molecular dynamics simulations. Fixman [13] showed that the introduction of hard holoniomic constraints in molecular dynamics models introduces statistical biases in the resulting ensemble averages of quantities computed using such models. Fixman proposed a solution for correcting such biases by adding the following compensating potential $\mathcal{U}_f$ (referred to as the Fixman potential) to eliminate the biases:

$$\mathcal{U}_f \triangleq \log\{\det\{\mathcal{M}\}\} \tag{47}$$

Including this potential in a molecular dynamics simulation actually requires the gradient of the potential in order to apply the resulting forces on the molecules. Despite its elegance, the use of the Fixman potential has remained impractical to implement for decades because of the lack of tractable techniques for evaluating the Fixman potential and its gradient. This problem has been addressed using spatial operator techniques, and these have been used to analyze and successfully derive the following simple expression for the Fixman potential gradient[14]:

$$\frac{\partial \log\{\det\{\mathcal{M}\}\}}{\partial \theta_i} = 2\,\mathrm{Trace}\left\{\mathcal{P}(i)\Upsilon(i)\widetilde{H}^*(i)\right\} \tag{48}$$

Evaluation of this expression in PyCraft is straightforward as follows:

Listing 23: Fixman potential

```
>>> grad_fixman_3 = 2 * Trace{P(3) * Y(3) * Hd_3(3)}    # evaluate ∂log{det{M}}/∂θ3 (Eq. 48)
```

6 Conclusions

This paper describes the PyCraft workbench for computing system level dynamics properties of multibody systems. PyCraft is based upon the SOA spatial operator methods that provide a rich mathematical vocabulary and analysis framework for describing a large variety of dynamics quantities. Examples of these range from system Jacobians, the mass matrix, the mass matrix inverse, the operational space inertia, operator sensitivities as well as the development of a range of recursive, low-cost computational algorithms. The effectiveness of the spatial operator approach lies in their ability of to address a large spectrum of dynamics computations in a concise and compact manner using just a handful of spatial operators. PyCraft is a C++/Python environment that allows the direct evaluation of dynamics quantities using statements that very closely mimic the mathematical operator expressions. One of the interesting aspects of PyCraft is that while it can be used for the overall multibody dynamics properties, it can also be applied to any connected subgraph of the multibody system (eg. limbs, legs etc.). The PyCraft workbench allows the easy evaluation and computation of complex dynamics quantities, and thus facilitates the development and validation of new analytical expressions.

While this paper has focused on multibody systems with rigid component bodies, the SOA methods have been shown to generalize to a much broader class of multibody systems including those with flexible bodies, flexible hinges and under-actuated systems. Remarkably, the operator expressions remain unchanged, even though the component elements of the operators change for the new types of systems. Thus we anticipate that future extensions of PyCraft to handle such broader classes of systems will leave its overall usage described here largely unchanged.

Another important area of extension is to closed-graph multibody systems. The approach with promise for this is one that builds upon the constraint embedding (CE) techniques. The CE techniques have been shown to

allow the transformation of closed-graph systems into tree-topology systems using aggregated compound bodies [2]. After such a transformation, the spatial operator techniques for tree systems readily extend to closed-graph systems as well. We believe that extension of the current PyCraft classes to handle compound bodies will thus allow its use to closed-chain systems.

Acknowledgments

The research described in this paper was performed at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under a contract with the National Aeronautics and Space Administration.[3]

References

[1] G. Rodriguez, A. Jain, and K. Kreutz-Delgado, "A spatial operator algebra for manipulator modeling and control," *International Journal of Robotics Research*, vol. 10, no. 4, p. 371, 1991.

[2] A. Jain, *Robot and Multibody Dynamics: Analysis and Algorithms*. Springer, 2011.

[3] A. Jain, "Structure Based Modeling and Computational Architecture for Robotic Systems," in *2013 IEEE International Conference on Robotics and Automation*, 2013.

[4] A. Jain, "Multibody graph transformations and analysis Part II: Closed-chain constraint embedding," *Nonlinear Dynamics*, vol. 67, pp. 2153–2170, aug 2012.

[5] J. J. Craig, *Introduction to Robotics*. Addison-Wesley, Pub. Co., Reading, MA, 1986.

[6] "Dynamics and Real-Time Simulation (DARTS) Lab." \url{http://dartslab.jpl.nasa.gov/}, 2016.

[7] "Simplified Wrapper and Interface Generator (SWIG)." \url{http://swig.org/}, 2016.

[8] J. Y. S. Luh, M. W. Walker, and R. P. Paul, "On-line Computational Scheme for Mechanical Manipulators," *ASME Journal of Dynamic Systems, Measurement, and Control*, vol. 102, pp. 69–76, jun 1980.

[9] M. W. Walker and D. E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," *ASME Journal of Dynamic Systems, Measurement, and Control*, vol. 104, pp. 205–211, sep 1982.

[10] O. Khatib, "The Operational Space Formulation in the Analysis, Design, and Control of Manipulators," in *3rd International Symposium Robotics Research*, (Paris), 1985.

[11] K. Kreutz-Delgado, A. Jain, and G. Rodriguez, "Recursive formulation of operational space control," *International Journal of Robotics Research*, vol. 11, no. 4, pp. 320–328, 1992.

[12] A. Jain and G. Rodriguez, "Diagonalized Lagrangian robot dynamics," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 4, pp. 571–584, 1995.

[13] M. Fixman, "Classical statistical mechanics of constraints: A Theorem and Application to Polymers," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 71, pp. 3050–3053, aug 1974.

[14] A. Jain, "Compensating Mass Matrix Potential for Constrained Molecular Dynamics," *Journal of Computational Physics*, vol. 136, no. 2, pp. 289–297, 1997.