Do not worry about your problems with mathematics. I can assure you that mine are far greater.

— Albert Einstein

Chapter 5

Snap-Together Mathematics

The mathematical techniques of the previous chapters permit controlling graphical objects by specifying the derivatives of functions of their parameters. In this chapter, we consider techniques for defining and evaluating these functions. The challenge stems from the dynamic nature of interactive systems: objects change in response to system actions as the system runs. This means that the functions that define controls must be created on the fly, in response to user actions. In order to effectively implement the mathematical calculations, we must evaluate the functions and their derivatives rapidly. This chapter presents *Snap-Together Mathematics,* a toolkit for dynamically defining functions and rapidly evaluating them and their derivatives.

With the differential approach, objects provide their attributes as output connectors for other objects to use, and as attachment points for controllers that will control the objects. These connectors compute functions of the objects' parameters and input dependencies, and must support the operations of attaching other object inputs and controllers. Snap-Together Mathematics provides a mechanism for realizing the connectors.

"Wiring" connector outputs to inputs builds new, more complicated functions from the elements being assembled. Building a new function may happen any time a new object, constraint or control is defined. It would be unacceptable if building a function required an extensive symbolic math computation or for the program to be recompiled and re-linked. Snap-Together Mathematics explicitly represents the expression graph of connected functional elements as C++ data structures. A connector is simply the output of a node in the expression graph. We will call the nodes *blocks.* To wire an input, it merely needs to be given a reference to some output. Snap-Together Mathematicshas efficient mechanisms for evaluating the values and derivatives of nodes by traversing the graphs.

For example, consider an expression graph that represents connecting the endpoints of two line segments together with an attachment constraint, as shown in schematically Figure 5.1. In this figure, the line segments have a state vector to store their parameters (x, y, θ, l), and provide the positions of their endpoints as connectors. The inputs to the attachment constraint are plugged into these output connectors. The output of the

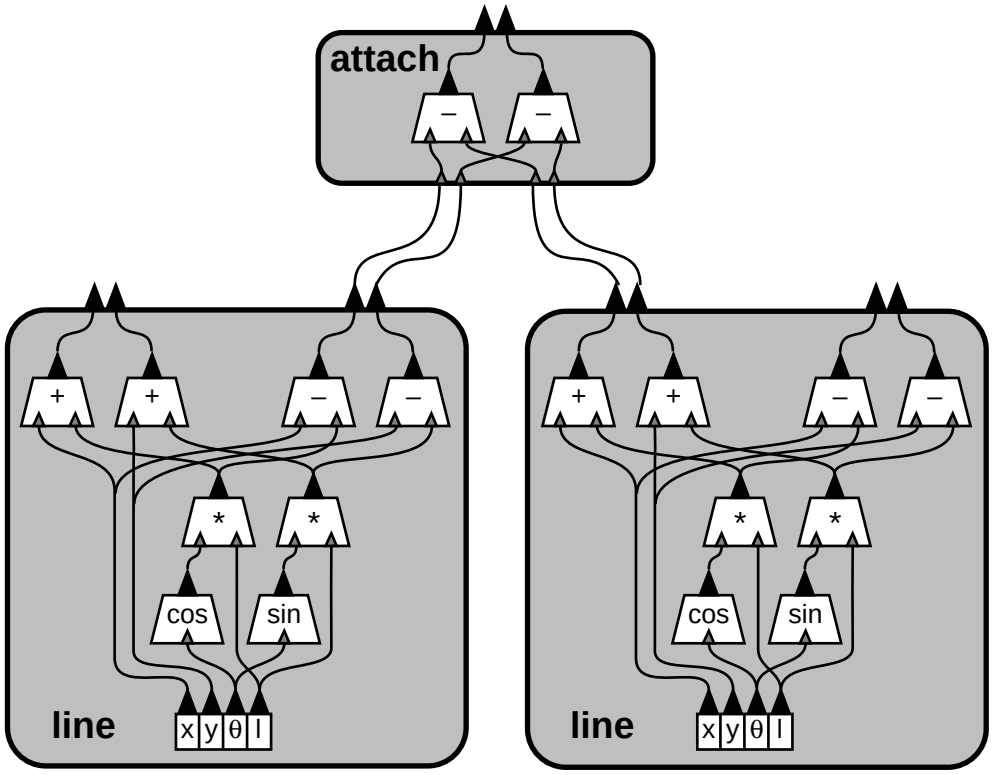

Figure 5.1: An expression graph representing two line segments connected by an attachment constraint. The outputs of the line segment objects, their attribute connector, serve as inputs to the attachment constraints. Function composition is also used inside the objects to build up the functions representing graphical object attributes.

constraint is a function of the variables of both line segments, built by composing the two attribute outputs with its own function. Snap-Together Mathematics allows this plugging to happen dynamically, for example if the user specified the constraint with a mouse click.

Snap-Together Mathematics provides a common protocol for block outputs, permitting any output to be wired to any input. This is important as it allows objects with inputs and those with outputs to be designed independently and dynamically snapped together at run time as needed.

Inside the graphical objects, blocks are wired together to compute the attribute functions. While these object functions could have been explicitly programmed because they would have been known ahead of time, constructing them by wiring together simpler blocks can simplify programming as it allows the programmer to avoid writing the code to evaluate the derivatives.

Several other systems, such as CONDOR [Kas92] and the SPAR Modeling Testbed [FW88], have explicitly represented expression graphs to the user. In contrast,

Snap-Together Mathematics provides the graph data structures as a general purpose tool for the programmer. The programmer could implement a graph viewer and permit the user to have direct access to the data structures. However, such an interface has not been implemented with Snap-Together Mathematics. The style of applications which it has been used to support intend to hide the mathematics from the user. Snap-Together Mathematics has been used for a number of purposes other than the differential approach including physical simulation and optimization-based motion planning.

The elements of Snap-Together Mathematics are not novel. Explicitly representing data flow graphs has been around for decades, and the techniques of automatic differentiation (AD), required to rapidly evaluate the derivatives, are becoming a common practice in the numerical analysis community. Snap-Together Mathematics addresses a very different need than previous AD systems have. Dynamic composition and evaluation of functions and their derivatives was introduced in a system by Witkin and Kass [WK88]. Snap-Together Mathematics is refines these basic ideas in an a simple, general purpose toolkit, allowing direct support for the abstractions of the differential approach. Snap-Together Mathematics was originally developed to support work in interactive physical simulation [WGW90], but has evolved into a more general purpose tool for encapsulating numerical computations.

5.1 Evaluating Functions

Evaluation is the most basic computation to be performed on expression graphs. In the interactive applications which we are considering, expression graphs will be evaluated many times per second, so performance is critical. The most efficient way to repeatedly evaluate an expression is to compile it into machine code. Unfortunately, compiling and linking code for each dynamically created expression is prohibitively expensive in the programming environments presently available.

Other approaches to evaluating the expression graph are interpretive: traverse the graph for each evaluation. Each node of the graph computes its output value, given the values of its inputs. A set of primitive function elements are predefined at compile time to do this. Evaluation of a node involves asking its predecessors for their output values then computing the "local" function of the node.

Performance can be enhanced using caching to exploit two types of redundancy: within an evaluation, common subexpressions need be evaluated only once (these subexpressions may be shared within one expression or between different expressions); between evaluations, certain old values might still be correct if some of the inputs did not change. Re-computation can be avoided by storing the results of a calculation and, for a later request, deciding whether this stored value is still correct. There are many possible ways to implement this cache validation; elaborate schemes might avoid some re-computation, but will require additional computation and storage to make the de-

termination. The more expensive the evaluations become, the more effort should be expended to avoid excess evaluations.

5.2 Evaluating Derivatives

We will need to evaluate the derivatives of expressions with respect to some subset of their inputs. Although the techniques extend to higher derivatives, for this discussion, we will consider computing first derivatives since this is what the differential approach's methods require. Derivatives are taken with respect to a set of variables that we call the *working set of variables.* The working set is a subset of the state vector. We denote the concatenation of this set of variables into a vector by $\mathbf{w}$. For a vector expression $\mathbf{f}$, the Jacobian, or first derivative, $\mathbf{J}$ is the matrix $\partial\mathbf{f}/\partial\mathbf{w}$. In this matrix, each row corresponds to an element of $\mathbf{f}$, while each column corresponds to a variable in $\mathbf{w}$.

There are three basic approaches to computing derivatives: approximate them numerically, derive a symbolic expression for the derivatives, or compose them using a process called *automatic differentiation.* The latter approach has been shown to be superior both in performance and precision of the results [Gri89].

To understand the process of automatic differentiation, consider how derivatives are computed manually. The chain rule allows us to decompose complicated functions into smaller pieces. For example, if our expression is $f = f(a, b, \ldots)$, then the chain rule yields

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w} + \cdots. \tag{5.1}$$

Differentiation involves recursive applications of the chain rule. If we are able to evaluate the derivative of each of the primitive functions with respect to their inputs then we can apply Equation 5.1 recursively to build the compound expressions. The recursion bottoms out at the constants, whose derivatives are 0, and at the variables, whose derivatives are 1 with respect to themselves and 0 with respect to others.

Symbolic differentiation applies the chain rule to an expression graph to transform it into a new expression graph that evaluates the derivative. The resulting expression must then be simplified to take advantage of the sparsity of the derivatives. Even then, the symbolic differentiation of a vector with respect to a vector yields a matrix of *expressions* which is unwieldy to manage.

Automatic differentiation also applies the chain rule to expressions; however, rather than symbolically composing more complicated expressions, the intermediate results are combined numerically. For any node in the graph, if the inputs to equation 5.1 are concatenated into a vector, the equation multiplies two matrices: the "local" Jacobian of the outputs with respect to the inputs, and the derivatives of the inputs with respect to the working set.

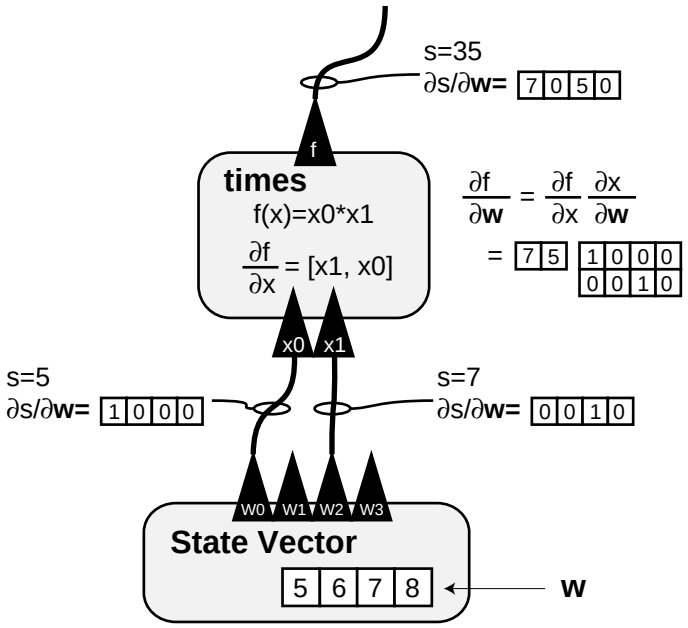

Figure 5.2: A simple example of derivative composition. Signals carry both values (s) and their derivatives ($\partial s/\partial \mathbf{w}$). The function block computes its internal Jacobian and composes the global Jacobian by multiplying the matrices.

We implement automatic differentiation by augmenting the expression graph with the ability to pass derivatives as well as values along edges. In addition to computing its output values, each node of the expression graph must also be able to compute the value of its local derivative, also a function of its inputs. The composition process builds the "global" Jacobian by multiplying this matrix with intermediate result matrices. By passing the entire intermediate result matrices along the edges of the graph, the derivative matrix can be built in one traversal of the expression graph. The same mechanisms for sharing intermediate results by caching as discussed in the previous section apply.

A recursive descent of the expression graph computes the derivative matrix. Each node in the graph is able to respond to requests for the derivative of its output with respect to the current working set. Constants and variables not in the working set return zero in response to this query. A state variable in the current working set returns a vector with one in the position corresponding to the variable, and zeros elsewhere. After determining that its cached value is not valid, a non-terminal node recursively asks its children for their derivatives, computes its local Jacobian, and multiplies these together to produce its derivative with respect to the current working set. Figure 5.2 demonstrates a simple example. Edges of the expression graph pass not only values, but also their derivatives.

This method of automatic differentiation assembles the Jacobian bottom-up, and is called the forward-mode. The alternative reverse-mode, or top-down, approach is presented by [Gri89] and implemented for an interactive system by [Sap93]. This algorithm reverses the order of the matrix multiplies, building the Jacobian matrix from the top down. It has the advantage that the intermediate result matrices are of small, fixed size. In the bottom-up approach, the size of intermediate matrices depends on the number of variables which contribute to that derivative. Because the intermediate results are fixed-sized, the top-down approach can achieve linear asymptotic complexity in places where the bottom-up approach has $O(n^2)$ complexity. However, this increased worst-case performance on dense problems comes at the expense of considerable bookkeeping, inability to fully exploit sparsity, inability to share intermediate results, much higher time constants, and difficulty in changing working sets.

It is important to recognize the generality of either of these derivative composition processes. Each node of the expression graph need only to be able to compute its *local Jacobian*, the derivative of its outputs with respect to its inputs. This matrix is a function only of the input values, not their derivatives. Given the local Jacobians, the composition process merely multiplies the matrices together to build the global derivatives.

5.3 Sparse Representations

The critical performance issue in building the Jacobian matrix, as well as most calculations that use this result, is exploiting sparsity. Bottom-up matrix passing schemes exploit sparsity by using sparse representations for the intermediate matrices. There are many possible ways to represent a sparse matrix, with many tradeoffs to consider in selecting a representation [DER86]. This decision is central to the design of an implementation. One particular representation with which we have had success is the half sparse matrix: a full vector of sparse vectors, as depicted in Figure 5.3. We call a system based on these data structures a *sparse vector* scheme.

In the sparse vector scheme we consider every output in the expression as an independent scalar, even if higher levels will interpret them as pieces of larger structures. A function block can have multiple scalar outputs. The gradient of each scalar is a sparse vector ($\partial x/\partial \mathbf{w}$).

Sparse vectors can be represented as a list of pairs $(index, value)$, taking space linear in the number of non-zero elements. If this list is sorted by index, we can perform the essential vector operations in time linear in the number of non-zero elements. For single vector operations, such as multiplying by a scalar or finding the magnitude, the algorithms simply run through the list. For multi-vector operations, such as addition, linear combination, or dot product, we exploit the sorted order of the lists and step through both in parallel, advancing which ever has the least index. These algorithms

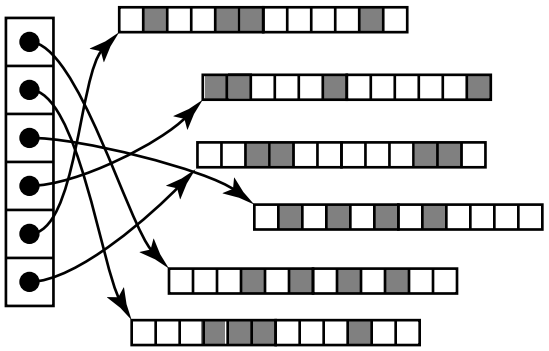

Figure 5.3: A *half-sparse* matrix representation store the matrix as a full vector of sparse vectors. Each entry in an array points to a sparse vector, depicted here by a sparsity pattern.

maintain the representation invariant so re-sorting is not needed.

Each derivative in the expression graph is represented as a sparse vector, the derivative of a scalar with respect to a set of variables. For each graph, one set of variables is denoted as the current working set: all derivatives are with respect to this set. For each variable, we must know an index to the corresponding column of the Jacobian.

Sparse vectors are collected into matrices which are half-sparse. While this is a non-standard representation, it does permit the operations required by the numerical methods we employ. In particular, half-sparse matrices can be multiplied by a vector or by its transpose rapidly. These methods are the essential computations in iterative linear system solvers like conjugate gradient [PFTV86]. The multiplication routines are among the most important in the entire implementation of the differential approach as they form the inner loop of $O(n^2)$ portion of the computation. However, the routines are very simple and can be coded efficiently. The algorithms are:

$\mathbf{v} = \mathbf{H}\mathbf{x}$
$multiply$ `HalfSparseMatrix` $\mathbf{H}$, `Vector` $\mathbf{x} \Rightarrow$ `Vector` $\mathbf{v}$
    $\mathbf{v}$ `= 0`
    `for` $i = 0 \ldots \mathbf{H}.rows$
        `for` $j = 0 \ldots \mathbf{H}.row[i].elements$
            $\mathbf{v}[i] += \mathbf{H}.row[i].value[j] * \mathbf{x}[\mathbf{H}.row(i).index[j]]$

$\mathbf{v} = \mathbf{H^T x}$
$multiplyTranspose$ `HalfSparseMatrix` **H**, `Vector` **x** $\Rightarrow$ `Vector` **v**
    v `= 0`
    `for` $i = 0 \ldots \mathbf{H}.rows$
        `for` $j = 0 \ldots \mathbf{H}.row[i].elements$
            $\mathbf{v}[\mathbf{H}.row[i].index[j]] += \mathbf{x}[\mathbf{i}] * \mathbf{H}.row[i].value[j]$

5.4 The Snap-Together Math Library

The machinery of the differential approach is encapsulated into a C++ toolkit called Snap-Together Math. The library implements the function composition and evaluation discussed in this chapter, as well as the differential solver.

5.4.1 The Protocol for Function Outputs

One of the essential elements to implement the differential approach is the standard protocol for connectors, the outputs of objects. Connectors correspond directly to Snap-Together Math block outputs. When a controller or object "plugs in" to a connector, it merely stores a reference to a Snap-Together Math output.

In order to be a Snap-Together Mathematics output, and object merely must provide two functions: one that computes its output value, and one that computes the derivatives of its values with respect to the current working set. In my implementation, the class of objects that have Snap-Together Mathematics outputs is the class `Port.` Each `Port` may provide a number of scalar outputs. The two methods that all `Port` subclasses must provide take an index that specifies which scalar value is being referred to, and return either the real number value of the output, or a pointer to a sparse vector (class `SpVec`) containing the derivative of the value with respect to the current working set. The signature for the minimum set of methods is:

```
class Port {
  public:
     virtual Real   o(const int);
     virtual SpVec* grad(const int);
}
```

In order to be a mathematical output, an object is required to provide only these two methods. The protocol supports many optional methods and fields such as the width

(the number of scalar values the object provides), strings that provide names for the signals, and nominal value ranges. To identify a particular scalar output, a pairing of a `Port` and an integer index is required. This is referred to as a `Signal.` Signals are constructed by pairing a pointer to a `Port` and an integer index.

Graphical objects could be subclasses of `Port` so that they could provide connectors for their attributes. However, this gives little structure when there are many outputs on a single objects. Instead, a graphical object will typically define other surrogate `Port` objects which provide smaller numbers of scalar outputs. For example, a polygon object might keep a list of vertices, each of which would be a `Port` object. This is important because these surrogate objects can be standardized. For example, all polygons could use the same vertex objects, so attaching to a vertex could be done independently of the polygon type.

Standardizing `Port` types adds further modularity. Snap-Together Mathematics permits connecting any scalar input and output. However, often such connections are most useful when they are grouped together and typed. For example, A 2D distance object has 4 scalar inputs, however these are most meaningful when they are thought of as two 2D point inputs. Graphical objects would provide specialized `Ports` which meet this standard.

Several standard types of `Ports` will be discussed in the following sections.

5.4.2 Functional Elements

The most basic element of Snap-Together Mathematics are objects which can be used in function composition. As explained in Section 5.4.1, these objects are subclasses of `Port` and must provide a few basic methods. Part of the beauty of the Snap-Together Mathematics scheme is the `Port` class is minimal enough that its functionality can be added to application classes. However, the Snap-Together Mathematics toolkit provides a variety of types of `Ports` for general mathematical elements that can be combined to build more complex structures.

The most fundamental `Port` subclass is `FBlock,` the class of function blocks. These objects compute mathematical functions of their inputs. The standard library includes various primitive functions, including almost all of the functionality of a scientific calculator. The class `FBlock` is implemented to have a fixed number of inputs for each subclass. Other special subclasses of `Port` can do things such as sum a variable number of inputs or take the magnitude of a vector of inputs. This permits creating controls on aggregate collections of objects, as in Garnet [VZMGS94].

Wiring functions together simply requires inserting an output `Signal` into the input field of a function block. The class `FBlock` stores its inputs as an array of `Signals`. Connecting the output of one function block to the input of another looks like

```
block1->ins[0] = Signal(block2,0);
```

where `block1` is a pointer to a function block (`FBlock*`), but `block2` can be a pointer to any subclass of `Port.` This code fragment connects the first output of the object pointed to by `block2` into the first input of the function block pointed to by `block1.` Notice that this wire is not explicitly represented, nor does the output port receive any indication that it is being attached to.

Defining a new function block requires specifying two methods. One that computes the function, and another that computes its internal Jacobian. It is important to note that this is really the only place where a programmer might have to take a derivative.

Writing the derivative routines is a simple matter of mechanically applying the rules of freshman calculus. However, this can be tedious as the functions get complicated, especially since we have a strong desire to have optimized code. Because the process is mechanical, it can be automated.

The *BlockMaker* tool automatically generates code for new function block types from mathematical expression. The tool, is written within the Mathematica symbolic algebra system [Wol88]. The tool takes as input an expression that describes the function block to be created, and a small amount of auxiliary information such as the number of inputs to the block and the name for the C++ class for the function block. The output of the tool is a C++ program file that contains the code for the block's methods, as well as a C++ header file describing the new class. The generated code is optimized using Mathematica's simplification tools as well as a common subexpression remover that I developed with Stephen Schwab. Because the tool runs within Mathematica, the full power of the symbolic algebra system is available to define the expressions used to create blocks.

New function block types are not often required. The Snap-Together Mathematics library includes many basic functions, such as those found on a scientific calculator, and more complicated functions can be created by composing these elements together. The main reason to create new blocks is efficiency. Composing a function out of other function blocks is more expensive than compiled code if the compiled code is optimized to exploit internal sparsity within the block and to share common subexpressions. For derivative evaluations, properly optimized code will perform the same evaluations as done by automatic differentiation. However, because it is explicitly compiled, there is less overhead. Automatic differentiation works better for more complex functions whose symbolic derivatives would be difficult to optimize.

5.4.3 Representing State Variables

The leaves of the expression graphs are constants and variables. The two are distinguished from one another in that the derivatives for the variables are not zero when taken with respect to itself, while the derivatives for the constants are always 0. Implementing a class to represent constant values is, therefore, simple; its methods just return constant values. The derivative of a variable must have a 1 in the column of the

gradient that corresponds to that variable.

The simple protocol for Snap-Together Mathematics does not address the issue of defining the working set of variables and the mapping of its members to columns of the Jacobian. This is indicative of the larger issue of managing collections of variables. On one hand, building systems in an object-oriented manner requires the state of the system to be distributed into the objects themselves. But, mathematical algorithms typically require this state in the form of contiguous vectors, which are gathered, ordered collections. This ordering also gives meaning to the columns of the Jacobian matrices.

There are many potential schemes for representing variables in a Snap-Together Math implementation ranging from having objects allocate space in a global state vector to modifying our numerical algorithms so that they operate on distributed, non-contiguous, vectors. Several of these were explored in earlier tools. The Snap-Together Mathematics library uses a combination of centralized and distributed representation. Objects each have their own state, however these variables are "gathered" into a centralized state vector for numerical computations. When an object's variable has been gathered, it knows where in the global vector to find it so it can still retrieve its value as well as index it for creating derivatives. In the context of Snap-Together Mathematics, derivatives can be taken only when variables are gathered, as this is the only time when variables correspond to matrix columns. When the numerical computations are complete, the values are scattered back into the corresponding smaller vectors.

The ability to scatter and gather variables has an important advantage over always keeping the variables centralized. It allows for the set of variables to be changed rapidly. This not only simplifies adding and deleting objects from the working set, but also makes it easy and efficient to operate on subsets of the variables. Techniques that make use of this latter feature are discussed in Section 4.4.

The scatter/gather scheme uses two main data structures: one to represent the smaller individual state vectors, and one to store the gathered global state vector. In Snap-Together Mathematics these classes are called `StObj` and `StVec` respectively. Each of these classes is a subclass of `Port,` although `StVec` objects rarely have connections made to them.

`StObjs` are objects that store a number of state variables. Each graphical object would have one that stores its configuration. `StObjs` are also used to store constants by marking their variables so that they are never gathered.

The gathering operation takes a list of `StObjs` and assigns designated variables in them to elements in an `StVec,` as depicted in Figure 5.4. Variables store their assigned location. If their value or derivative is requested when they are in an assigned state, they forward the request on to the `StVec.` If they are not assigned, they return the value stored internally and 0 for their derivative. A scatter operation returns each variable in the `StVec` to its corresponding `StObj,` updating the `StObj`'s internal value, and removing the assignment.

The `StVec` provides a contiguous vector for mathematical computations. Routines

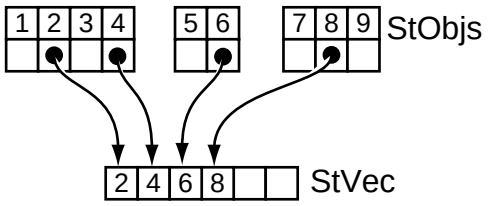

Figure 5.4: Selected variables from `StObj` objects are gathered into the `StVec` object. Variables in the working set point to slots in the `StVec` object.

such as ODE solvers look at the data stored here. The indices of the `StVec` define the columns of the Jacobian: requests for derivatives of the `StVec` return vectors with a 1 value in the corresponding column.

An important part of the scheme for storing state variables is the ability to gather[1] selectively. This provides the ability to switch a variable off, or turn it into a constant. Uses for disabling variables are described in Section 4.4.

During a gather operation, a function is provided that decides if a variable is to be gathered or not. Typically, the default function is used. This function makes its decisions based on a bit vector stored with each variable. The bit vector and function provide the following mechanisms for selection:

- Each variable belongs to a *caste* that identifies its type. Whether or not a given caste is to be gathered can be decided independently. This permits operations like "gather only the variables that affect lighting."

- Each variable has several type bits, which permit denoting the expected use of the variable. For example, if a variable is denoted as a constant, it will not be gathered.

- Each variable has a counter that freezes it whenever the count is non-zero. A problem with using a single bit for this purpose is that it is impossible to determine how many constraints are freezing a variable. If two constraints freeze the same variable, deleting one of the constraints would re-enable the variable. A counter re-enables the variable only when all freezes have been removed.

Selective gathering also provides a mechanism for merges, or equating two variables. During a gather, two variables can be made to share one space in the `StVec`. This will effectively merge them, constraining them to have the same value. A similar technique can be use to constrain a variable to have the same value as any other `Signal`. Such features are not used much in the Snap-Together Mathematics implementation, because they make the optimization of the next paragraph impossible.

[1] The corresponding scatter operations always scatter what was last gathered, so there is no selection involved in them.

An important special case of a function block is one that has each of its inputs connected to the same `StObj`. This is a very common case, used often for graphical objects where the function blocks each compute some attribute. Because all of the inputs are connected to variables, the Jacobian of the inputs of the block is the identity matrix, possibly with some columns missing. By exploiting this, the matrix multiply to compute the block's Jacobian can be considerably faster. This leads to substantial speedups in many applications, as these direct-connected blocks are very common, and very often constitute the majority of the "wide" input blocks, which are the most expensive to compute.

Partitioning, as introduced in Section 4.3.3, reorders the elements of the `StVec` so that variables in a common partition are next to one another, facilitating solving each subset independently. Partitioning is done only when a gather operation is performed, not each time the linear system is solved. This is done because re-ordering the state vector would confuse the process of differential equation solving.

Because the partitioning algorithm does not actually look at the values of $\mathbf{J}$, they do not need to be computed when finding the initial matrix to partition. In fact, rather than using the real values of the matrix, it can be better to use binary values which represent that the matrix element might be non-zero, rather than that it is non-zero for the current value of the state vector. This is easily achieved by having each `Port` provide a method which works like the `grad(int)` method, but does a logical or instead of a linear combination of its input vectors. For objects that do not provide this optional method, the gradient may be used instead.

5.4.4 Caching in Snap-Together Math

As hinted at in Section 5.4.1, caching is an important tool for enhancing performance in Snap-Together Mathematics evaluations. Whenever a value or derivative is computed by a function block, or other `Port` type that implements caching, it is stored inside the block in case the result is used again. Each time a value or derivative is requested, the block first decides if its cached value is valid; if it is, it avoids re-computation and simply returns the cached value.

The Snap-Together Mathematics toolkit employs a simple cache validation scheme. A single global timestamp is used. Whenever any of the inputs (state variables) are changed, this timestamp is incremented. A block must recompute if its internal timestamp is older than the global timestamp. This scheme is very simple but has the obvious problem that it invalidates much more than needs to be. Changing a single value invalidates all caches in the entire system.

In the context of the differential approach, invalidating all the caches simultaneously is not as catastrophic as it might seem since the variables are all updated simultaneously with each ODE solver call for evaluation of the differential optimization. Schemes that do substantially better require some combination of substantial amounts

of graph traversal, explicitly representing links bidirectionally, doing numerical and sparse data structure comparisons, or exploiting knowledge about the particular problem. For most applications, such complication is not warranted as, at best, it serves only to reduce the constants on the linear complexity portions of the differential approach.

5.4.5 The Differential Solver

The Snap-Together Mathematics toolkit encapsulates the abstractions of the differential approach in a set of classes that implement the techniques of the previous chapters. An object is used to represent the differential optimization problem, storing information about the controls and the variables they effect. This object is called a constraint engine or `ConstEngine.`

The `ConstEngine` class has fields that contain an `StVec` and a list of `StObj` that are to be gathered for computations. A `ConstEngine` object also stores information to define the objective function for the differential optimization, such as a list of `Signals` that make up $\mathbf{g}$.

The controls for the differential optimization problem are stored in a `ConstEngine` by a list of `Controller` objects. `Controllers` are objects that specify desired derivatives for connectors, as discussed in the next chapter. The Snap-Together Mathematics class for a controller specifies a `Signal` to control, a controller type, and several parameters.

Solving the differential optimization problem, e.g. using the techniques of Chapter 3 and Chapter 4, is executed by a method of the `ConstEngine` class. This is the only part of the system in which constraint solving needs to be done. This method is implemented in a manner that interfaces with the ODE solver implementations of the underlying mathematics toolkit. The `ConstEngine` class also interfaces with non-linear iterative solvers.

Snap-Together Mathematics is built on top of an object-oriented mathematics toolkit that I wrote. The toolkit includes an object-oriented framework for defining ODE problems and solvers. The class `Integrand` represents an ODE problem by defining a single method that defines a function to compute $\dot{\mathbf{q}}$ given $\mathbf{q}$ and t. An `ODESolver` object stores an integrand and an initial condition and offers a method to step time forward. The `ConstEngine` class is a subtype of `Integrand`.

When used as an `Integrand` to solve a differential optimization problem, a `ConstEngine` must first load the $\mathbf{q}$ vector provided by the ODE solver into its `StVec.` Loading the state allows the solver to try different values for the state in the process of taking an ODE step. The `ConstEngine` keeps all of the intermediate results of the solving process, such as the Lagrange multipliers, as internal fields.

5.4.6 The Whisper/Snap-Together Math Interface

While Snap-Together Mathematics is a C++ toolkit, an interface is provided to it as an extension to the Whisper interpreter described in Appendix A. The extension adds several new data types to Whisper, as well as many new primitive functions. `WhSTM` provides a convenient way to access to the functionality of the Snap-Together Mathematics, and can be used on its own to develop simple applications completely in Whisper. The Whisper/Snap-Together Mathematics interface, called `WhSTM,` shows how the functionality of Snap-Together Mathematicscan be provided in a more convenient form, and will be used in later portions of the thesis.

`WhSTM` provides primitive Whisper data types for most the Snap-Together Mathematics classes `Port`, `FBlock`, `StObj`, `Signal`, and `Const`. Other classes, generated by primitives written in C++ are generally treated by the more generic class that is appropriate. For example, a summation block, which is not an `FBlock` because it allows variable numbers of inputs, simply appears as a `Port` in Whisper. No facility for defining new subtypes of Snap-Together Mathematics classes is provided in Whisper. `WhSTM` defines 88 primitives, including creation functions for 30 different types of function blocks.

`WhSTM` supports automatic promotion of types as needed. Any function requiring a `Port` can take anything that is a subtype, including `FBlock` and `StObj` types, even though Whisper has no subtyping mechanism. Real numbers are also promoted to `Port` where necessary; a constant valued port object is automatically created.

The constructor for `Signal` takes a `Port` and an index. It permits specifying the index either as an integer or as a string name if the block being connected to supplied the optional names for its outputs. If the index is omitted, it is assumed to be 0. This allows a `Port` to be promoted to a `Signal` when needed.

Function block creation routines all optionally take initial values for the signal input. The convenience of this, coupled with the automatic promotions, is demonstrated in this simple example

```
(set b (plus-block (times-block (signal point-port 2) 5)
                   (signal point-port 'y)))
```

that computes the sum of 5 times the z value of a 3 output port representing a Cartesian coordinate and its y value. The ease with which functions can be built in Whisper will be used in the Bramble toolkit, described in Chapter 7. A more extensive example of using `WhSTM` to build function graphs is described in Section A.2.

The Whisper interface does not provide constraint engines or ODE solvers as basic types. However, `WhSTM` does have default instances of these objects, so that commands like

```
(add-const (controller sig '= 2.0))
```

implies the use of the “built-in” constraint engine, and

```
(rk4-step .1)
```

uses the built-in 4th order Runge-Kutta solver instance to step the default constraint engine forward a time step. Other packages further extending Whisper can alter the defaults. This will be used extensively in Bramble.