

nwcommands: Scalable and extensible network analysis in Stata

Thomas U. Grund
RWTH Aachen University

Abstract. An early, informally distributed prototype of `nwcommands` (2014, GitHub only, never formally published) demonstrated that network analysis could follow Stata’s own syntax conventions, but an unconditional dense adjacency matrix, used throughout the prototype’s 92 commands (by August 2015), left it impractical much beyond a few hundred nodes. Run today against this article’s own benchmark networks, that prototype’s unweighted betweenness centrality command takes 8.7 seconds at 500 nodes, 69 seconds at 1,000, and nearly 9 minutes at 2,000 – a size well within what a single organization’s employee network or a school’s friendship network would routinely produce. `nwcommands`, introduced here in its first officially published release, is a complete architectural rewrite. A sparse, size-adaptive index becomes the primary representation in place of unconditional dense storage, and persistent network objects replace ad hoc per-network global macros: a network’s directedness, valuedness, two-mode status, and temporal status are declared once and read by every subsequent command, rather than re-specified or silently lost between commands. The Stata-facing interface is separated from the computational backend, so that betweenness centrality – one of two operations found to need it so far – runs through purpose-built native code, with an always-available Mata implementation otherwise and identical syntax either way. Sparse-native structural operations are demonstrated on networks of 100,000 nodes and 1,000,000 edges, and betweenness centrality’s own runtime comes within reach of a dedicated graph package (R’s `igraph`) rather than remaining orders of magnitude behind it. We describe this architecture, demonstrate it with a worked example, and report the benchmark evidence – against both the 2014 prototype and R – that motivated its design. Social scientists, epidemiologists, and other researchers already working in Stata who need to analyze relational data of realistic size are the intended audience.

Keywords: `st0001`, `nwset`, `nwsummarize`, `nw2project`, networks, sparse matrices, social networks, graph theory

1 Introduction

Network data occupy an unusual position in general-purpose statistical software. Most statistical analyses assume data organized around independent observational units: rows represent individuals, organizations, events, or other cases, while columns represent their characteristics. Networks instead describe relations between units. Their natural representation is therefore relational: observations concern pairs of nodes, and many quantities of interest depend on the structure of the network as a whole.

There are two standard ways to write a network’s relational structure down, and both recur throughout this article. An *edge list* records one row per tie: a pair of node identifiers and, if the tie carries a value, a third column for it. An *adjacency matrix*

instead records every possible pair at once, as an $n \times n$ table with a 1 (or a tie’s value) where two nodes are connected and a 0 where they are not. The two are interchangeable descriptions of the same structure. A four-node network in which Ana follows Bo, Bo follows Cara, Cara follows Ana, and Cara also follows Dan – the same network Section 2.4’s Figure 1 draws from real `nwset/nwplot` output – is equally

Edge list		Adjacency matrix				
ego	alter		Ana	Bo	Cara	Dan
Ana	Bo	Ana	0	1	0	0
Bo	Cara	Bo	0	0	1	0
Cara	Ana	Cara	1	0	0	1
Cara	Dan	Dan	0	0	0	0

an edge list of four rows, or a 4×4 adjacency matrix with one 1 per tie, not mirrored across the diagonal since “follows” is not automatically reciprocal (how `nwcommands` declares and remembers directedness is Section 3.1’s subject). `nwcommands` accepts either as input and can produce either as output (Section 2.4); which one is more practical to store depends on how sparse the real network is, since an edge list costs space proportional to the number of ties actually present while an adjacency matrix costs space proportional to n^2 regardless – the distinction Section 5 returns to as the central design problem `nwcommands` solves.

Social network analysis studies exactly this kind of relational structure – how actors are connected, and how the resulting pattern of ties shapes outcomes for individuals and for the system as a whole (Wasserman and Faust, 1994; Freeman, 2004; Borgatti et al., 2018; Newman, 2018) – and is this article’s primary motivating field and running example, but the underlying representation is not specific to social units: the same nodes-and-edges structure, and the same `nwcommands` commands, apply equally to genomic interaction networks, semantic networks, or any other relational data with a natural graph structure. Section 4’s worked example introduces the specific descriptive quantities used throughout the article – degree, centrality, density – as they arise, rather than in the abstract here.

This creates a practical difficulty for users of Stata. Stata provides an extensive environment for data management, statistical modeling, graphics, and reproducible research, but network analysis has historically required researchers either to leave that environment or to translate repeatedly between conventional Stata datasets and specialized network representations. An early prototype under the `nwcommands` name was developed to reduce this separation, providing a collection of network-analysis commands using Stata syntax (Grund, 2014). That prototype was never formally published – it was distributed via GitHub and presented at conferences, with no accompanying journal article – so this article is the first official release of `nwcommands` and the first peer-reviewed description of it in any form, not a sequel to one.

The prototype’s internal representation of a network motivates much of the redesign this article describes. From its first public version (2014), it represented a network as a dense adjacency matrix – built in Mata for efficient linear algebra, but anchored to the surrounding Stata session through a loose collection of per-network global macros (a counter and, for each network, a name, a label list, a directedness flag, each its

own separate global). This design was flexible and Mata-native from the start, but carried two compounding costs as the prototype’s 92 commands (by August 2015) grew around it: the matrix itself cost $O(n^2)$ storage regardless of how sparse the actual network was, and a network’s own properties – directed or not, weighted or not – lived as loose, ad hoc strings rather than as structured fields a command could depend on. Run today, against the identical benchmark networks used in Section 6, the prototype’s betweenness centrality command – an ordinary structural measure, not an outlier – takes 8.7 seconds at 500 nodes, 69 seconds at 1,000, and nearly 9 minutes at 2,000, a size well within what a single organization’s employee network, a school’s friendship network, or a modest citation network would routinely produce.

One command stood apart from the rest of this architecture. `nwergm`, an experimental statistical-modeling addition rather than a core structural command, estimated exponential-family random graph models by shelling out to a separate R installation entirely – writing the network to a `.dta` file, invoking R’s own `ergm` package as an external process, and reading the results back in, rather than estimating anything itself. This was not how the rest of the prototype worked; it was a one-off, and a fragile one, dependent on R being installed, reachable, and compatible with whatever `.dta` format Stata itself happened to be writing. Run today, it no longer works at all, failing on exactly that kind of accumulated incompatibility: a Stata/R data-format mismatch that has built up in the decade since.

`nwcommands`, as officially released here, addresses both problems, though differently in scope. The dense matrix is replaced with a sparse, size-adaptive index across the whole package (Section 5), and the scattered globals with a single, structured network object (Section 3); separately, and unrelated to that general rewrite, `nwergm` is rebuilt as a native Stata/Mata MCMC implementation with no external dependency at all (Section 5.4). Table 1 summarizes the prototype and the official release, and the known limitation of each.

Table 1: The unpublished 2014–2015 prototype versus `nwcommands`’ internal representation, as officially released here.

	Representation	Known limitation
2014–2015 prototype (unpublished)	Dense Mata matrix, anchored by per-network global macros	$O(n^2)$ storage; network properties as loose, unstructured strings
<code>nwcommands</code> (this release)	Sparse, size-adaptive index inside a structured network object, with selective native kernels	Some algorithms remain dense by necessity (Section 5.2)

This was not an oversight discovered later but a limitation the prototype’s own documentation anticipated from the start. Its August 2015 book-length manual – current at 92 commands across 268 files at the time (Grund, 2015) – already described the global-macro-plus- dense-Mata-matrix design as storing “network quasi-objects,” and stated explicitly that although “the internal design and structure might change” in later

versions, “the external design and the interfaces for the way one deals with networks through the `nwcommands` will remain the same.” `nwcommands`, as released here, completes a transition its own earliest documentation had already promised rather than starts one from scratch: the persistent, structured network object described next is that decade-old promise, kept.

That object is the central idea of this article: a network should be an object with persistent properties, rather than merely a matrix supplied to individual commands – not any particular command that creates one. Concretely, `nwcommands` rests on three architectural principles, each with a section of its own below, plus one further principle governing not how the system is built but how it is meant to behave for users:

1. **Networks as first-class, persistent objects** (Section 3): a network’s directedness, valuedness, two-mode status, and temporal status are declared once and read by every subsequent command, and several named networks can coexist alongside ordinary Stata variables in the same session.
2. **A representation independent of the interface** (Section 5): sparse storage with a size-guarded dense fallback, replacing the unconditional $O(n^2)$ matrix of earlier versions, without the commands built on top of it needing to know which representation backs a given network.
3. **Computation independent of the interface** (Section 5): stable `nw*` syntax that can be served by more than one underlying implementation – Mata by default, and, where evidence justifies it, compiled native code – without the user-facing command changing.
4. **A Stata-native command philosophy** (Section 2.3): beyond vocabulary that mirrors Stata’s own (`netname` for `varname`, `nwdrop` for `drop`), commands are built to behave the way an experienced Stata user already expects a Stata command to behave – so that using `nwcommands` draws on existing intuitions rather than requiring new ones.

Commands consequently inspect the network on which they operate and determine which operations are meaningful for its type, without the user re-specifying directedness, valuedness, two-mode status, or temporal status at each step – and, since many empirical networks are sparse relative to the number of possible ties, this determination is made without ever building an $O(n^2)$ matrix in the common case.

The result is intended to be more than an expanded collection of network commands – though it is that too: as of this writing, the package comprises more than 130 documented commands (up from 92 in the prototype’s own August 2015 status snapshot (Grund, 2015)), organized into the families described in Section 4. `nwcommands` provides an infrastructure on which specialized network methods can be built while preserving a common Stata workflow. For example, exponential-family random graph modeling (`nwergm`) operates directly on the same network-object representation used for descriptive analysis and visualization, however that object happened to be declared. Such specialized methods are treated separately; the focus of this article is the underlying network-analysis framework.

The remainder of the article proceeds as follows. Section 2 introduces the network data model and shows how networks are declared and stored. Section 3 describes network types and persistent metadata, including directed, weighted, two-mode, and temporally indexed networks. Section 4 illustrates the common command interface using a worked example. Section 5 describes the computational architecture and the interaction between Mata, sparse representations, and the optional native backend. Section 6 reports the benchmark evidence for these design choices. Section 7 discusses compatibility with the 2014 prototype. Section 8 discusses the framework’s design relative to alternatives and concludes.

2 Networks as Stata data

2.1 Installation

nwcommands requires Stata 12 or later and is installed like any other user-written Stata package, from within Stata:

```
. net from "https://raw.githubusercontent.com/thomas-u-grund/nwcommands/master"  
. net install nwcommands  
. nwininstall, all
```

The first line points Stata at the package’s source; the second installs a small, permanent bootstrap package – just **nwininstall** itself and a handful of landing help topics, so **help nwcommands** works even before anything else is on disk. **nwininstall, all** then downloads and installs everything else – the full command set, all help files, and the extension and dialog-box packages – in one call, internally still split into several numbered chunks because Stata’s own **.pkg** format caps how many files a single package may list, a limit this package’s size exceeds; a reader never needs to know the chunk names or how many there are. **master** is the package’s current, tested release line – active development happens on a separate **develop** branch and is merged into **master** only once tested – so it, not a branch still in motion, is what this article’s installation instructions, command counts, and benchmark results refer to throughout. An installation that tracks the repository directly, so that it always reflects the latest commit rather than a **net install**-time snapshot, can instead add a cloned copy of the repository to Stata’s own **adopath**.

2.2 Quick start

Before the architectural detail that fills most of this article, here is the complete round trip for a small network: declare it from raw edge data, describe it, and compute a structural measure – three commands, no setup beyond ordinary Stata data entry.

```
. clear  
. input str8 ego str8 alter  
      ego      alter  
1. "Ana" "Bo"  
2. "Bo" "Cara"  
3. "Cara" "Ana"
```

```

4. "Cara" "Dan"
5. end
. nwset ego alter, edgelist directed name(mini)
. nwsummarize mini

```

```

Network name: mini
Network id: 1
Directed: true
Valued: false
Two-mode: false
Nodes: 4
Selfloop: false
Arcs: 4
Minimum value: 0
Maximum value: 1
Density: .3333333333333333
Temporal: false
. nwdegree mini

```

```

Network name: mini

```

Degree distribution			
_indegree	Freq.	Percent	Cum.
1	4	100.00	100.00
Total	4	100.00	

_outdegree	Freq.	Percent	Cum.
0	1	25.00	25.00
1	2	50.00	75.00
2	1	25.00	100.00
Total	4	100.00	

```

Indegree centralization:: 0
Outdegree centralization:: .444

```

`nwset` parses `ego/alter` as an edge list and, with `directed` declared once at this first call, remembers it for the rest of the session: `nwsummarize` reports `Directed: true` without the option being repeated, and `nwdegree` accordingly reports separate in-/out-degree distributions rather than a single undifferentiated degree. Nothing here required exporting to another package, converting between formats, or telling the same command about the network's properties twice. The remainder of this section explains why that is true in general, not just for this one example.

2.3 A Stata-native command philosophy

Before describing the network object model in detail, it is worth making explicit a design choice that runs through the whole package and that much of what follows simply instantiates: `nwcommands` is built so that network commands behave the way an experienced Stata user already expects a Stata command to behave, rather than introducing a parallel, network-specific idiom to learn from scratch. That correspondence goes beyond vocabulary to command semantics, to how destructive operations are guarded, and to how state persists across a session.

Familiar vocabulary

Table 2 lists the basic concepts this mirroring covers; a **netname** plays exactly the role a **varname** plays in an ordinary Stata command, a **netlist** is abbreviated with the same wildcard notation as a **varlist** (Section 2.4), and so on.

Table 2: Basic concepts and their Stata counterparts.

Stata	nwcommands	Description
varname	netname	Name of an existing network
newvarname	newnetname	Name of a new network
varlist	netlist	List of networks
exp	netexp	Network expression

This carries through to entire commands, not just terminology. Table 3 lists a representative subset of **nwcommands** programs that are close, deliberate analogues of an ordinary Stata command applied to networks instead of variables. A user who already knows that **drop** removes variables and **keep** retains them does not have to learn a new mental model to guess, correctly, what **nwdrop** and **nwkeep** do to networks.

Table 3: A representative subset of commands with a direct Stata analogue.

Stata	nwcommands	Description
clear	nwclear	Clear all network data
drop	nwdrop	Drop networks from memory
keep	nwkeep	Keep networks in memory
use	nwuse	Use network data
webuse	nwebuse	Use network data from the internet
save	nwsave	Save network data
generate	nwgenerate	Generate a new network
egen	nwgen	Advanced network generation
replace	nwreplace	Modify an existing network
recode	nwrecode	Recode networks
summarize	nwsummarize	Summarize networks
tabulate	nwtabulate	Tabulate networks
correlate	nwcorrelate	Correlate networks
syntax	nw_syntax	Parse network syntax (Section 5.5)

Familiar command semantics

Naming is only the most visible part of the correspondence. Stata itself distinguishes **generate**, which creates a new variable and refuses to overwrite an existing one, from **replace**, which modifies an existing variable’s values – a general principle that a command should never silently discard existing state, requiring the user’s intent to be stated wherever an operation is potentially destructive. **nwgenerate** and **nwreplace** implement exactly this distinction for whole networks rather than variables, and the same

principle recurs throughout the package: more than thirty commands require an explicit **replace** before producing a network or file under a name that already exists, and **nwload**'s own size guard (Section 6.3) requires an explicit **force** before materializing an unusually large network as Stata variables. The following is genuine, captured behavior, not a paraphrase: two small random networks are declared first, named **friendship** and **advice**, and **nwgenerate** is then asked to create a third network under the name **friendship** – already taken by the first line:

```
. nrandom 6, prob(.5) undirected name(friendship)
. nrandom 6, prob(.5) undirected name(advice)
. capture noisily nwgenerate friendship = 2*advice
Network friendship already exists. Change netname or specify option replace.
. nwgenerate friendship = 2*advice, replace
. nwreplace friendship[2,1] = 0
```

The **nwgenerate** call is rejected outright, by name, because **friendship** already exists; only once **replace** is added does it succeed, exactly as **generate/replace** would for an ordinary Stata variable. **nwreplace** then modifies a single tie in place – the network-level analogue of **replace**'s own dyadic counterpart. **nwset** follows the identical guard: an explicit **name(foo)** colliding with an existing network errors with the same message unless **replace** (or its legacy alias, **overwrite**) is given, and the check runs once, centrally, before a new network is ever registered, regardless of which of **nwset**'s six declaration forms is used. The right-hand side of **nwgenerate/nwreplace** also accepts a full network expression – ordinary Stata/Mata operators and functions applied element by element to the underlying adjacency matrix – and **nwgen** dispatches across a library of named functions the same way **egen** does, from network generators to per-node summary statistics.

Stata-like lists, defaults, and state

The same philosophy governs how a session accumulates and refers back to its own state. Network lists accept the same wildcard notation (*****, **?**, ranges, **_all**) as an ordinary Stata **varlist** (Section 2.4); a *current network* lets a **netname** be omitted where unambiguous, the network-level analogue of Stata's own implicit **in-memory** dataset (Section 2.4); and, as Section 2.5 describes below, results are written into ordinary Stata variables rather than trapped inside a separate network environment. None of this is forced by the underlying architecture – it is chosen because it is what a Stata user already expects, so that this article's central claim, that network analysis becomes part of the same workflow as ordinary Stata data management rather than a separate system bolted alongside it, is felt at the keyboard, not just argued in the abstract.

2.4 Declaring and naming networks

The principal entry point to **nwcommands** is **nwset**. Suppose the current Stata dataset contains an edge list – a pair of variables recording, one row per tie, who is connected to whom – with variables **ego** and **alter**. A network is declared using

```
nwset ego alter [value], edgelist [[, options]]
```

(reproduced from `nwset.sthlp`). The following four lines are enough to turn raw edgelist data into both a declared network object and a picture (Figure 1):

```
. clear
. input str8 ego str8 alter
      ego      alter
1. "Ana" "Bo"
2. "Bo" "Cara"
3. "Cara" "Ana"
4. "Cara" "Dan"
5. end

. nwset ego alter, edgelist directed name(mini)
. set scheme stsj
. nwplot mini, label(_nwnode) labelopt(mlabsize(medium) mlabposition(3) mlabgap(*2)) ///
>      scatteropt(mfcolor(gs4) mlcolor(black) msize(*1.6)) lineopt(lcolor(gs8) lwidth(medium)) ///
>      xscale(off) yscale(off) plotregion(margin(zero)) graphregion(margin(small))
Calculating node coordinates...
Generating splines...
Plotting network...
```

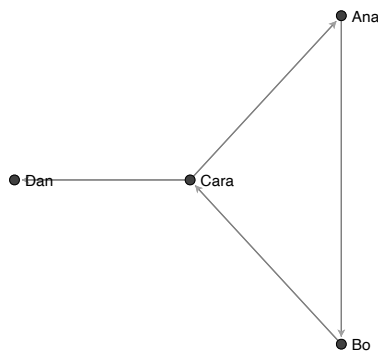


Figure 1: The 4-node network declared from the edgelist above.

The important distinction is that `nwset` does not perform a one-off conversion required by the next command. It creates a persistent network object – concretely, a Mata class instance living in Mata’s own workspace, not a set of Stata dataset characteristics – that `nwplot`, and every other `nw` command, can subsequently address by name (`mini` above). Section 3 describes what such an object stores; Section 5.5 returns to its concrete Mata representation for programmers extending the package.

`nwset` has six declaration forms in total, sharing one option set: the edgelist form just shown; an adjacency matrix from a variable list (each variable one column of the matrix); a network from an existing Mata matrix (`nwset, mat(matamatrix)`); a two-mode network from an edgelist or a wide affiliation matrix (`twomode/bipartite`, see Section 3.2); and a temporal network from an edgelist (`time()/interval()/eventtime()`, see Section 3.3). The Mata-matrix form is a single line even for a directed network with

labeled nodes –

```
. nwset, mat((0,1,0,0\0,0,1,0\0,0,0,1\1,0,0,0)) directed name(toy) labs(A, B, C, D)
```

(Section 4 shows this exact call with its real captured output, as part of a worked example on why the `directed` declaration matters downstream). A bare `nwset` (no arguments) instead displays the networks currently in memory; `nwset, clear` drops networks only, and `nwset, nwcLEAR` drops networks and any Stata variables they were loaded into.

More than one network can exist in memory simultaneously. Every network has a name (1–32 characters, must start with a letter or underscore), and commands that operate on a single network accept an optional `netname` argument that defaults to the *current network* (queried or set with `nwcurrent`) when omitted – the network-level analogue of Stata’s own implicit in-memory dataset. Commands that can meaningfully operate on several networks at once accept a `netlist` instead, using Stata’s ordinary wildcard notation: `*` for zero-or-more characters, `?` for exactly one, `-` for a range, and `_all` for every network currently in memory. For example, given networks `flomarriage` and `flobusiness` (introduced properly in Section 4):

```
. nwwebuse florentine, nwcLEAR
. nwdegree flobusiness
```

Network name: flobusiness			
Degree distribution			
_degree	Freq.	Percent	Cum.
0	5	31.25	31.25
1	3	18.75	50.00
2	2	12.50	62.50
3	2	12.50	75.00
4	3	18.75	93.75
5	1	6.25	100.00
Total	16	100.00	
Degree centralization:: .238			

```
. nwcomponents *marriage
```

Network name: flomarriage			
Components: 2			
_component	Freq.	Percent	Cum.
1	15	93.75	93.75
2	1	6.25	100.00
Total	16	100.00	

```
. nwsummarize flo*
```

```
Network name: flobusiness
Network id: 1
Directed: false
```

```

Valued: false
Two-mode: false
Nodes: 16
Selfloop: false
Edges: 15
Minimum value: 0
Maximum value: 1
Density: .125
Temporal: false

```

```

Network name: flomarriage
Network id: 2
Directed: false
Valued: false
Two-mode: false
Nodes: 16
Selfloop: false
Edges: 20
Minimum value: 0
Maximum value: 1
Density: .1666666666666667
Temporal: false

```

Issuing `nwset` with no arguments reports the networks currently available in memory by name:

```

. nwwebuse florentine, nwcLEAR
. nwset
(2 networks)

```

```

flobusiness
flomarriage

```

This is particularly important for applications involving several types of relations, network comparisons, projections of two-mode networks, or repeated observations of network structure; the identifier scheme permits up to 9,999 named networks in memory simultaneously, subject to available memory. This design is intended to preserve a central advantage of Stata: analyses can be organized as sequences of commands operating on a well-defined data state, so that network analysis becomes part of the same workflow as ordinary data management and statistical analysis rather than requiring repeated export to and import from a separate network environment.

2.5 Node identities, labels, and Stata variables

Every node in a declared network has an integer `nodeid` (1 through N , assigned automatically and renumbered contiguously if nodes are later dropped) and, separately, a `nodelab` – a user-facing string label that can be changed without altering the underlying identifier. The two are looked up in each direction by `_nwnodeid` and `_nwnodelab`.

Critically, the network object does not, by default, occupy any Stata variables at all: building and analyzing a network never has to spend Stata’s own variable budget unless the user asks for it. But most non-trivial analyses do want to combine network structure with ordinary, per-node research data – wealth, treatment status, survey responses – and this is where the architecture has to do real work to keep the two representations from silently drifting apart.

How node attributes stay matched to the network. The join key is a single string variable, `_nwnode`, holding each node’s label. Whenever an `nw` command reads or writes a per-node Stata variable – an attribute such as `wealth` supplied by the user, or a result such as `_between` the command itself produces – it matches dataset observations to network nodes by `_nwnode`, not by row position, and creates `_nwnode` automatically if it does not yet exist. This reconciliation is handled by `nw_datsync`, invoked transparently by every command that touches node-level Stata variables, so that the workflow already demonstrated in Section 4 –

```
[ordinary Stata data preparation, e.g. wealth already in the dataset]
nwset ego alter
[network calculations, results written to Stata variables by _nwnode]
regress wealth _between
```

– is safe even though the dataset’s observation order and the network’s own internal node order are two independent things that need never coincide: a name-based match, not a positional one, is what keeps them aligned. The check itself is cheap and skipped whenever the dataset is already sorted consistently with the network, so there is ordinarily no reason to intervene; `nw_datsync` exists mainly so that a user working with an unusually large network can switch this off deliberately (`off`), at the cost of then being responsible for row alignment directly.

A network’s own tie values (its adjacency matrix, as opposed to per-node attributes) can additionally be materialized as Stata variables on request, via `nwload` or the `xvars` option many commands accept – one variable per node, in the same `_nwnode`-matched layout. This is a separate, optional mechanism from ordinary attribute syncing, with its own explicit synchronization command, `nwsync`: by default, changing a network through an `nw` command automatically re-syncs its loaded Stata-variable copy, and `nwsync, fromstata` can push edits made directly to those Stata variables back into the underlying network – a capability that exists for interoperating with ordinary Stata data commands that know nothing about networks, though `nwreplace` (Section 2.3) remains the more direct way to edit a network’s own tie values. Network measures become variables in subsequent analyses, and ordinary Stata variables can supply attributes used by network commands – the network representation supplies relational structure; Stata remains responsible for the research data, and `_nwnode` is the thread that keeps the two honestly connected.

3 Network types and persistent metadata

One of the principal architectural changes in `nwcommands` is that characteristics of a network belong to the network itself. Concretely, every `NWdef` object carries boolean fields – `isdirect`, `isvalued`, `is2mode`, `isselfloop`, `istemporal` – set once when the network is declared and read by every command that subsequently operates on it, rather than re-derived or re-specified with every command call.

3.1 Directed and weighted networks

A network’s directedness – declared explicitly with the `directed` or `undirected` option – retains that property for its lifetime once set. (Edge-list input can instead be left undeclared, in which case `nwset` checks whether the resulting tie structure is exactly symmetric and marks the network undirected only in that case; an explicit declaration is unambiguous and is what this article’s own examples use throughout.) Commands that distinguish incoming and outgoing relations – for example, in- and out-degree, or the directed forms of centrality and reachability – determine from this metadata whether the distinction applies to the network they were called on, rather than requiring the user to specify directedness again at every step.

Similarly, a network’s `isvalued` property records whether edge weights are meaningful. `nwcommands` separates the existence and storage of edge weights from the semantics a particular algorithm imposes on them: the same stored weight can be treated as a strength (larger is closer) by one command and as a cost or distance (larger is farther) by another – for example, in geodesic-distance and betweenness calculations, which support both an unweighted (hop-count) mode and an explicit weighted mode.

3.2 Two-mode networks

Two-mode (bipartite) networks – connecting two distinct kinds of actors, such as people and the events they attend – are declared as a distinct network type (`is2mode`), rather than treated as an ordinary network that happens to await projection to one mode. `nwcommands` never projects a two-mode network automatically merely because a requested command expects a one-mode graph: commands with a native two-mode formulation redirect to it automatically. Calling `nwdegree` or `nwclustering` on a two-mode network, for example, prints a note (“...is a two-mode network – using `nw2degree` instead”) and dispatches to `nw2degree`/`nw2clustering` rather than computing the one-mode version against the wrong input; `nwergm` checks and rejects two-mode input explicitly. `nw2project` is the explicit projection command for cases with no native two-mode formulation at all: a person-by-event network is not silently converted into either a person-by-person co-attendance network or an event-by-event overlap network, and `nw2project` requires the analyst to say which projection, and which weighting of it, is intended.

The current scope of this guard across the rest of the structural command family is discussed together with the architecture’s other limitations in Section 6.3.

3.3 Temporal information

The architecture also allows timing information associated with ties or network observations to be retained, using one of three mutually exclusive declarations at `nwset` time: a per-edge `time()` snapshot timestamp, a `start/end interval()` over which a tie is present, or an `eventtime()` declaration for timestamped relational events, which are kept as a distinct event list rather than ever being folded into an ordinary persistent-tie graph. A declared temporal network can be sliced back to an ordinary static network at a given time point with `nwattime`, whose result is then usable with any ordinary `nw*` command.

The purpose of declaring temporal metadata at this point in the architecture – ahead of the full analytical command set that will eventually use it – is to avoid an architectural dead end in which timing information would later have to be bolted onto a representation designed exclusively for static graphs. Section 6.3 discusses the current scope of what is built on top of this declaration so far.

4 A common network-analysis interface

Rather than catalogue the package’s command families, this section follows a single network object through data management, structural analysis, comparison, visualization, and ordinary Stata modeling, to make the case that these are one continuous workflow rather than separate systems joined by data export. The example uses the Florentine-families marriage and business networks of Padgett and Ansell (1993) (16 families each, distributed with the package and fetchable with `nwebuse`), also used as the package’s own top-of-file example.

```
. nwebuse florentine, nwcLEAR
. describe
Contains data from https://raw.githubusercontent.com/thomas-u-grund/nwcommands/master/data/florentine.nwd
> ta
Observations:      16
Variables:         5      16 May 2019 17:02
```

Variable name	Storage type	Display format	Value label	Variable label
_nwnode	str12	%12s		
_nwinclude	float	%9.0g		
wealth	float	%9.0g		
priorates	float	%9.0g		
seat	float	%9.0g		

```
Sorted by:
Note: Dataset has changed since last saved.
. list in 1/5
```

	_nwnode	_nwinc-e	wealth	priora-s	seat
1.	acciaiuoli	1	10	53	1
2.	albizzi	1	36	65	1
3.	barbadori	1	55	0	0
4.	bischeri	1	44	12	1
5.	castellani	1	20	22	1

`nwebuse` loads both the network structure and three ordinary node attributes – `wealth`, `priorates`, `seat` – as plain Stata variables in the same dataset, exactly the point made in Section 2.5: the network object supplies relational structure, while Stata’s own dataset remains responsible for the research data sitting alongside it. Summarizing the marriage network reports the metadata described in Section 3 – directedness, valuedness, two-mode status, temporal status, node and edge counts, density – read directly off the declared network object, without the user repeating any of it:

```
. nwsummarize flomarriage
```

```

Network name: flomarriage
Network id: 2
Directed: false
Valued: false
Two-mode: false
Nodes: 16
Selfloop: false
Edges: 20
Minimum value: 0
Maximum value: 1
Density: .16666666666666667
Temporal: false

```

The same object is then used for a structural measure, whose result feeds directly into an ordinary Stata regression – the general pattern sketched in Section 2.5 (`nwset` / network calculations / `regress`) made concrete:

```
nwbetween [netlist] [[, generate(newvarlist) replace nosym standardize
silent weighted alpha(real) ]]
```

```
. nwbetween flomarriage, generate(_between)
```

Network name: flomarriage						
Betweenness centrality						
Variable	Obs	Mean	Std. dev.	Min	Max	
_between	16	9.75	12.30056	0	47.5	
. regress wealth _between						
Source	SS	df	MS	Number of obs	=	16
Model	2515.25701	1	2515.25701	F(1, 14)	=	1.97
Residual	17880.6805	14	1277.19146	Prob > F	=	0.1823
				R-squared	=	0.1233
				Adj R-squared	=	0.0607
Total	20395.9375	15	1359.72917	Root MSE	=	35.738
wealth	Coefficient	Std. err.	t	P> t	[95% conf. interval]	
_between	1.052739	.7501664	1.40	0.182	-.5562077	2.661686
_cons	32.29829	11.54646	2.80	0.014	7.533589	57.063

`nwbetween` writes its result into an ordinary Stata variable, explicitly named via `generate()` rather than left to a command default – the same explicit-naming discipline Section 2.3 already described for `nwgenerate`/`nwreplace` – and `regress` consumes it exactly as it would any other Stata variable, alongside `wealth`, which came from the same `nwebuse` call – the network-measure result and the rest of the Stata workflow are not two separate systems. (At 16 families the regression itself is illustrative of that mechanical pattern, not a claim about Renaissance Florence; nothing about the workflow depends on the coefficient reaching conventional significance, and it does not here.) Two declared networks can also be compared directly:

```
. nwcorrelate flomarriage flobusiness, permutations(100)
(100 missing values generated)
```

```
Network1 name: flomarriage
Network2 name: flobusiness
```

```
Correlation: .372
P-value: 0
```

Section 3.1 described directedness as metadata read directly off the network rather than re-specified at every command call; a minimal directed example makes this concrete. Declaring a 4-node directed cycle and calling `nwdegree` on it produces `_indegree`/`_outdegree` rather than the single `_degree` variable the marriage network above produced, with no option needed to request the distinction:

```
. nwset, mat((0,1,0,0\0,0,1,0\0,0,0,1\1,0,0,0)) directed name(toy) labs(A, B, C, D)
. nwdegree toy
```

```
Network name: toy
```

```
Degree distribution
```

<code>_indegree</code>	Freq.	Percent	Cum.
1	4	100.00	100.00
Total	4	100.00	
<code>_outdegree</code>	Freq.	Percent	Cum.
1	4	100.00	100.00
Total	4	100.00	

```
Indegree centralization:: 0
Outdegree centralization:: 0
```

Finally, the same marriage network can be drawn with `nwplot` (Figure 2), and since its `edgecolor()` option itself takes a *network*, the network expressions from Section 2.3 do real work here: a single elementwise multiplication marks which marriage ties overlap with a business tie, and that overlap network colors the plot's edges directly from the same declared objects, not a separately exported copy of them:

```
. set scheme stsj
. nwgen overlap1 = flobusiness * flomarriage
. nwplot flomarriage, label(_nwnode) edgecolor(overlap1) scheme(sj) edgefactor(1.6) ///
>     labelopt(mlabsize(medium) mlabposition(12) mlabgap(*2)) ///
>     scatteropt(mfcolor(gs4) mlcolor(black) msize(*1.6)) ///
>     xscale(off) yscale(off) plotregion(margin(zero)) graphregion(margin(small))
Calculating node coordinates...
Plotting network...
```

Solid edges in Figure 2 mark marriages reinforced by a business tie, dashed edges mark those without one. `nwplot`'s own graphical range – layout algorithms, attribute-driven color and size mapping, coordinate export – is beyond this article's scope; its role here is architectural, drawing from and coloring by the same objects every other command in this section used.

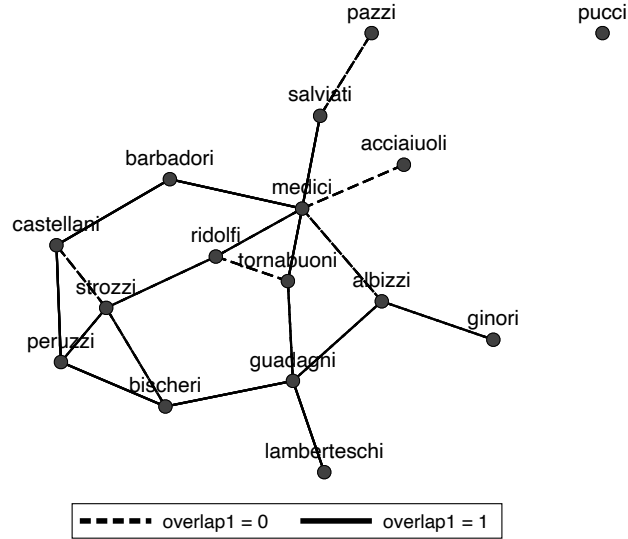


Figure 2: The Florentine marriage network, drawn with `nwplot`; edge style marks overlap with the business network.

The same network object also feeds statistical modeling, not only descriptive measures or visualization: `nwergm` fits an exponential-family random graph model directly against the same `flomarriage` object used throughout this section, no separate data-preparation step or export to another package’s format required:

```
. nwergm flomarriage, edges gwesp(.5) nodematch(seat)
Exponential-family random graph model
Network:          = flomarriage
Nodes:            = 16
Ties:             = 20
Directed:         = No
Estimation:       = MCMLE
MCMC sample size: = 3000
```

flomarriage	Coefficient	Std. err.	z	P> z	[95% conf. interval]	
edges	-1.967704	.1656122	-11.88	0.000	-2.292298	-1.64311
nodematch_seat	.4867564	.2000735	2.43	0.015	.0946195	.8788932
gwesp_.5	.0936174	.1307498	0.72	0.474	-.1626475	.3498822

MCMLE converged after 1 iteration(s).

The three specified terms mix structural, higher-order-dependence, and nodal-covariate statistics; `nwergm`’s own estimation machinery – MCMC sampling, MCMLE optimization, and validation against R’s `ergm` – is the subject of a separate article (Section 8). What matters here is architectural, not statistical: the same persistent network object (Section 3) serves every one of these commands without re-declaration.

Beyond this illustrative sequence, `nwcommands` organizes its more than 130 commands into families for import/export, network generation, descriptive information, network manipulation, structural analysis (centrality, cohesion and components, community detection, positions and roles, paths and reachability), statistical network models, visualization, and a small programming layer, exactly as organized in Stata’s own `help nw_topical`; Appendix A summarizes these families with representative commands from each.

5 Computational architecture

5.1 The scaling problem

A dense adjacency matrix for n nodes requires storage proportional to n^2 , regardless of how many possible ties are actually present. For a network of 50,000 nodes, that is 2.5 billion matrix cells; a social network of that size with an average degree of 10 contains only roughly 250,000 undirected edges. This mismatch between $O(n^2)$ dense storage and the $O(n)$ – $O(n \log n)$ tie count typical of empirical social networks is what made the package’s earlier, dense-matrix-only representation impractical beyond a few thousand nodes – the earlier design allocated an $n \times n$ Mata matrix unconditionally on network creation, whether or not the network was sparse.

5.2 Sparse representation

Every network object now additionally maintains a sparse index – a compressed row-storage (CSR) structure for out-neighbors, and, for directed networks, an analogous structure for in-neighbors. This index is built lazily on first use and supplies $O(1)$ or $O(\text{degree})$ accessors (neighbor lookup, degree, edge weight, tie existence) that algorithm code uses in place of dense matrix indexing; Section 5.5 gives the exact programmer-facing accessor names. A genuinely sparse-native construction path also exists for building a network directly from an edgelist, used throughout the package’s import and manipulation commands, that never allocates an $n \times n$ matrix at all.

The original dense matrix is retained as a compatibility fallback, not removed: it is materialized lazily, only on first real demand, and this lazy materialization is itself size-guarded (networks above 20,000 nodes raise an explicit error rather than risk an out-of-memory dense allocation). Most computationally significant commands – components, clustering, betweenness, degree, the geodesic-distance family (both unweighted breadth-first search and weighted Dijkstra), neighbor queries, and eigenvector centrality (via sparse power iteration rather than dense eigendecomposition) – have been migrated to use the sparse accessors exclusively. A small number of algorithms remain dense by necessity rather than oversight: spectral analysis (`nwspectral`) requires the full eigen-spectrum, for which Mata offers no sparse alternative to its dense `symeigensystem()`, and a small bounded-subgraph routine used internally by cohesive-subgroup diameter checks. Section 6 reports the storage and runtime consequence of this migration at scale.

5.3 Separation of interface and computation

Figure 3 summarizes the resulting layering. Every `nw` command’s Stata syntax is independent of which implementation actually runs. A command such as `nwbetween` calls a dispatcher (`NativeGraphAvailable()`) at run time; if a compiled native plugin is present for the host platform *and* applicable to the requested computation, it is used, and the result is otherwise obtained from the always-available Mata implementation. Compiled kernels are built and distributed for macOS (a universal x86_64/arm64 bundle), Windows, and Linux; a platform without a matching compiled plugin – or a computation outside what the compiled kernel covers, such as `nwbetween`’s weighted mode – falls through to the Mata implementation automatically, never with an error. This selection is per-command, not a global switch, and is transparent to the user: syntax, options, and stored results are identical either way.

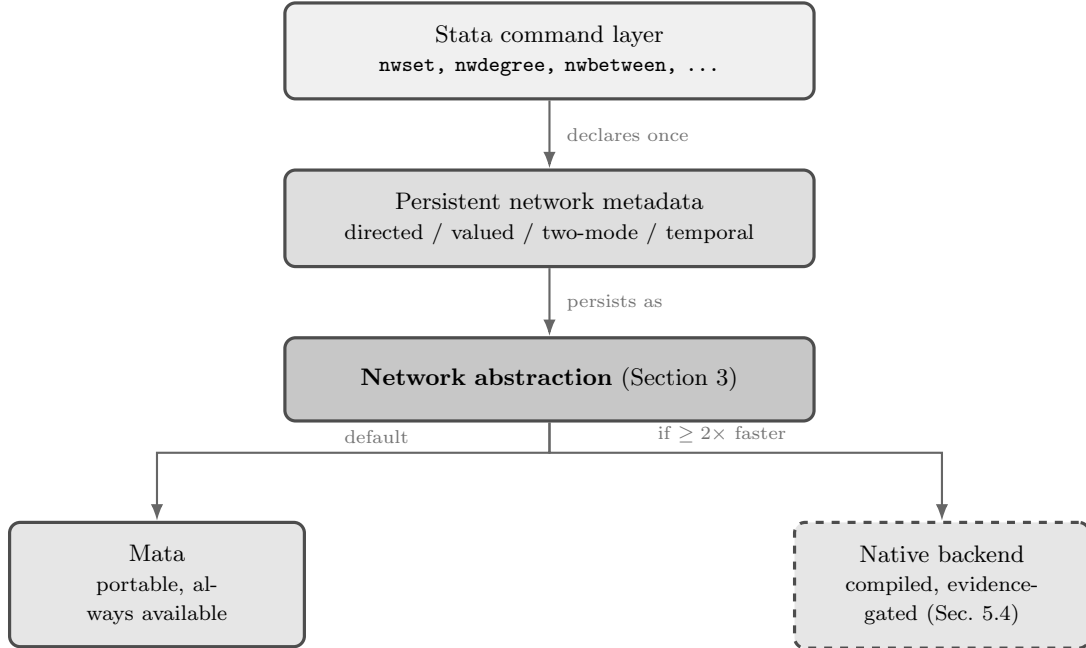


Figure 3: Layering of the Stata command surface, persistent network metadata, and the Mata/native computational backends; darker shading marks the layer a command author never has to reach below (Section 5.5).

5.4 Evidence-based native acceleration

A native implementation does not replace a working Mata implementation simply because native code is available. We use a conservative, quantified threshold: a native implementation is adopted only when it is at least twice as fast as the sparse Mata implementation over the relevant benchmark range and passes the corresponding certification

tests confirming identical results. As of this writing, this bar has been cleared for exactly two computations: unweighted betweenness centrality (Brandes’ algorithm (Brandes, 2001), in `nwbetween`’s default mode) and the MCMC sampler used by `nwergm` for exponential-family random graph estimation. Both are implemented as small, purpose-built C kernels compiled through the Stata Plugin Interface, not as bindings to an external graph library.

We also evaluated linking `nwcommands` against established native graph libraries – `igraph` (Csárdi and Nepusz, 2006) among them, the most familiar option to a network-analysis audience – rather than writing bespoke kernels. For the operations currently accelerated, benchmarking favored a small, purpose-built kernel over a general-purpose library regardless of that choice: betweenness centrality specifically showed the interpreter-overhead signature a native port addresses, while component-finding and k -core peeling, profiled at the same sizes, did not. A small, purpose-built kernel keeps the dependency footprint minimal and leaves the Mata implementation as a portable fallback for every unaccelerated operation – a decision driven by that evidence rather than by any specific objection to the libraries considered. Among the remaining structural measures benchmarked in Section 6, eigenvector centrality (`nwevcent`) is the clearest evidence-based candidate for the next native implementation (Section 6.2).

5.5 Extensibility

A third-party Stata programmer can write a new `nw`-family command without knowing how the sparse index, the dense fallback, or the native dispatcher work internally. Concretely, a single global anchor (a Mata variable named `nw`, of class `NWs`) holds a container object (class `NWsdef`) that stores an array of pointers to individual network objects (class `NWdef`), indexed by name – the object graph Section 2.4 described at a higher level. This class hierarchy dates to 2015, when StataCorp invited the author to visit for a month and helped set up its foundational structure in what is now `unw_core.do`; everything described in Sections 3 through 6 is built on that foundation. The package’s programming layer exposes this through a single entry point, `nw_syntax`, which – given a network name – leaves behind local macros for the network’s Mata pointer (named `netobj`), node count (`nodes`), and type flags (directed, valued, two-mode, self-loops), so that a new command’s own code never has to name `NWs`, `NWsdef`, or `NWdef` directly – only the local macros `nw_syntax` itself populates:

```
nw_syntax `netname'
tempname __nw_result
mata: `__nw_result' = `netobj'->calculate_something()
mata: st_store(1::`nodes'), "`generate'", `__nw_result')
mata: mata drop `__nw_result'
```

The sparse accessors described in Section 5.2 (`neighbors()`, `degree()`, `has_edge()`, `edge_weight()`, `edgelist()`) are the preferred entry points into a network object for new algorithm code, in preference to the dense matrix accessor. This pattern is documented, together with a set of concrete Mata programming pitfalls specific to this architecture, in the package’s own programming help file, and a written style guide governs naming, network-type handling, and error-message conventions for new commands.

The infrastructure itself – the sparse index, the class hierarchy, the native dispatcher – was the hard part to build, and is what Sections 3 through 6 describe. Once it is in place, adding one more descriptive measure is comparatively small: typically a single Mata method on `NWdef` written against the sparse accessors above, plus the same thin `nw_syntax` wrapper every existing command already uses. The package’s own growth is evidence of this: from 92 commands in the original prototype’s own August 2015 status snapshot (Grund, 2015) to more than 130 today (Section 1), with several genuinely new capabilities the prototype never had at all added after this architecture was in place – k -core decomposition (`nwkcore`) and eigenvector centrality via sparse power iteration (`nwevcent`) among them, both used in Section 6’s benchmarks. The architectural cost is paid once, centrally; each additional algorithm pays comparatively little of it again.

6 Performance and scalability

The architecture described in Section 5 makes two empirical claims, and this section answers them in turn: does sparse storage permit practically larger networks than the earlier dense representation, and does selective native acceleration remove the computational bottlenecks that remain once storage is no longer the limiting factor?

6.1 Does sparse storage permit practically larger networks?

At 100,000 nodes and 1,000,000 edges, building the network and answering degree and neighbor queries against it via the sparse index completes in about one second, and a full connected-components pass in about seven seconds; the lazy-dense flag confirms that the $n \times n$ dense matrix – which would require approximately 74.5 GB at this size – is never constructed. Figure 4 extends this comparison to seven sizes from 100 to 100,000 nodes at the same fixed average degree (10) used throughout Section 6. The sparse-index series is measured, not estimated: each point builds a network via the genuinely sparse-native construction path (`set_edge_from_triplets()`, Section 5.2) and reads the resulting object’s own index arrays directly, rather than assuming a bytes-per-edge constant. The dense series remains analytic ($n^2 \times 8$ bytes) for the obvious reason that this matrix is deliberately never built – there is nothing to measure – but the formula itself is exact, not an approximation, since a dense Mata real matrix costs precisely 8 bytes per cell. At 100,000 nodes the gap is already a factor of roughly 4,762; at 500 nodes, the smallest size shared with Tables 4 and 5 below, it is a factor of roughly 24 – sparse storage’s advantage widens with n rather than being a large-network special case.

6.2 Does selective native acceleration remove the bottleneck?

Tables 4 and 5 report measured wall-clock time at two sizes, 500 and 5,000 nodes, on sparse random undirected networks with fixed average degree 10 (Section 5.1’s storage argument only holds in this regime; a fixed *density* instead would make every operation below scale quadratically by construction, which is not how the networks this article is about actually grow), generated once per size with a fixed random seed and read by both languages so Stata and R time exactly the same edges rather than two inde-

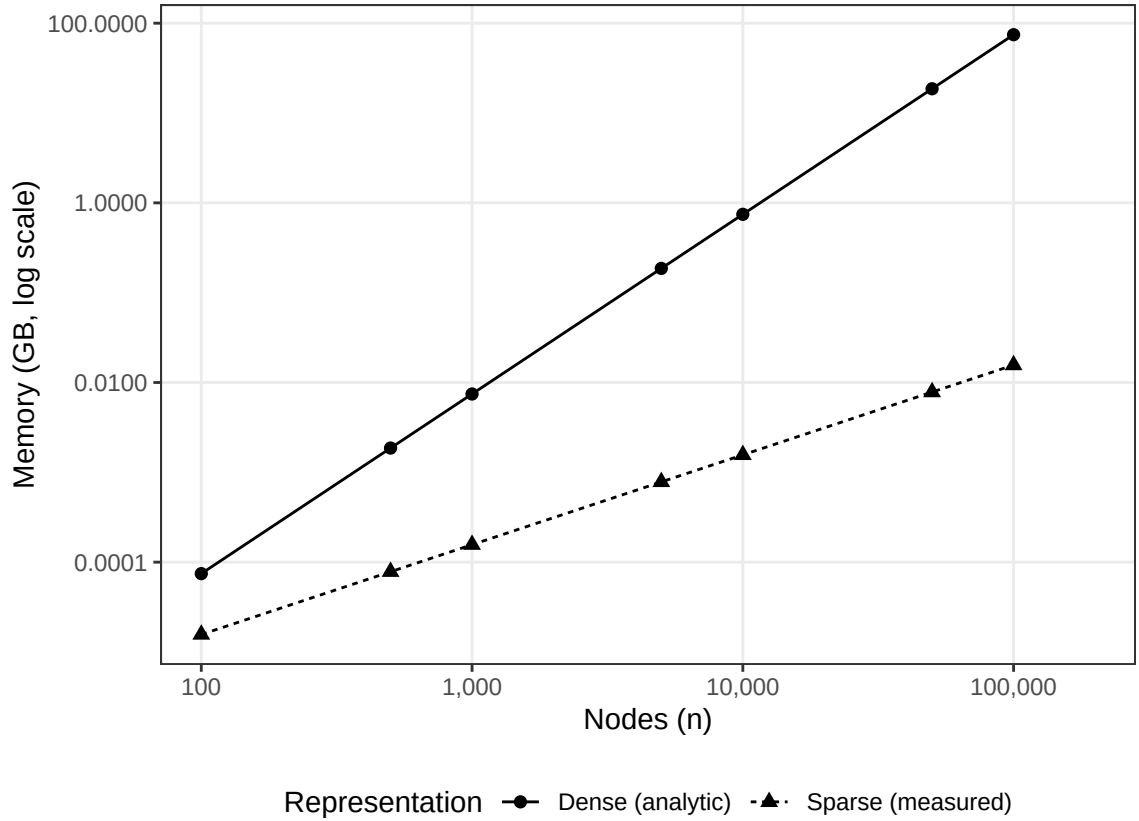


Figure 4: Measured sparse-index memory versus analytic dense-matrix memory, seven sizes at fixed average degree 10 (log–log scale).

pendent draws, covering six operations – degree, component-finding, k -core, clustering, eigenvector centrality, and betweenness – using the harness in `paper/bench/`, which also generates the edgelist. Every Stata figure is the *median* of 5 repeated calls on an already-built network – construction cost is deliberately excluded, since these tables are about per-call cost, not one-time setup – and every native call is preceded by an untimed warm-up call so the one-time cost of loading the compiled plugin into a fresh Stata session never lands inside a measurement. An earlier version of this comparison skipped both of these precautions: a single untrimmed draw at $n = 100$ landed on the plugin-load cost and reported it as a 0.391s betweenness time, which it was not.

All figures in this section were measured on a single machine (Apple Silicon, 64 GB RAM, macOS 26), using Stata 18 BE, R 4.6.0, and igraph 2.3.2; absolute times will differ on other hardware, though the comparisons between implementations on the same machine should not.

R’s non-betweenness figures need one further explanation, since a naive single-call timing of most of them reports a flat 0.000s at every size tested here. That is not a sign anything is wrong with R or with igraph; it means R’s own wall-clock timer has roughly 1 millisecond resolution on the system these were run on, and a single igraph call to degree, component-finding, k -core, or clustering genuinely finishes within a fraction of that – confirmed directly by timing 1,000 consecutive calls in a single block and dividing, which gives a stable, non-zero per-call estimate that no single-call reading can resolve. Every R figure below therefore times a calibrated batch of consecutive calls (as many as needed to clear the timer’s resolution) and divides; betweenness, and eigenvector centrality at the larger size, are already slow enough to clear that resolution on one call, so batching does not change those numbers.

Table 4: Runtime at 500 nodes, 2,500 edges (seconds; Stata figures are the median of 5 repetitions).

Operation	R (igraph)	Stata
Degree (degree/nwdegree)	0.0001	0.006
Components (components/nwcomponents)	0.0001	0.004
k -core (coreness/nwkcore)	0.0001	0.003
Clustering (transitivity/nwclustering)	0.0001	0.035
Eigenvector centrality (eigen_centrality/nwevcent)	0.0006	1.277
Betweenness (betweenness/nwbetween)	0.0100	0.017

Every Stata figure above is what an ordinary call to the command actually costs – nothing here depends on knowing, or caring, whether a given command happens to be backed by a compiled kernel or by plain Mata; that choice is invisible at the interface by design (Section 5.4). It matters for only one row in this table. **nwbetween** can also be forced, for comparison, to skip its compiled kernel and run the same sparse Brandes’ algorithm in plain Mata instead: doing so takes 3.862s at 500 nodes, more than 200 times the 0.017s above. A separate, one-off check goes back further still: running the actual pre-sparse, dense-Mata-matrix **nwbetween** from the last tagged release before the sparse rewrite (v1.5.1, November 2015, in a separate git worktree) against this identical 500-node edgelist takes a median of 8.705s over 5 repeated calls (individual timings 8.636–8.848s) – to check a specific claim against evidence rather than assert it, that the pre-sparse implementation could not practically handle networks of even this size. It can; 8.7 seconds is not “impossible.” But it is roughly 500 times slower than the compiled kernel reaches today. The same v1.5.1 **nwset/nwdegree/nwcomponents** calls on this network took 0.094s/0.010s/0.006s, essentially the numbers Stata’s column above already shows for degree and components: those two were already fast even in the dense, pre-sparse architecture at this scale, consistent with their showing no profiling signal worth compiling today either. Betweenness, which does real repeated traversal work rather than a single pass over the matrix, was the one operation dense storage made slow – exactly the profile Section 5.4’s evidence-gating exists to find. **nwkcore** did not exist at all before the sparse rewrite (it was added together with it); it is not merely faster now, it is new.

^aA measurement of this identical cell taken earlier in the package’s development – same

Table 5: Runtime at 5,000 nodes, 25,000 edges (seconds; Stata figures are the median of 5 repetitions).

Operation	R (<i>igraph</i>)	Stata
Degree (<i>degree/nwdegree</i>)	0.0002	0.081
Components (<i>components/nwcomponents</i>)	0.0002	0.090
k -core (<i>coreness/nwkcore</i>) ^a	0.0002	0.017
Clustering (<i>transitivity/nwclustering</i>)	0.0007	0.388
Eigenvector centrality (<i>eigen_centrality/nwevcent</i>)	0.0063	13.251
Betweenness (<i>betweenness/nwbetween</i>)	1.1910	1.290

methodology, same generated network – found 1.7 seconds here, roughly 100 times slower than the 0.017s reported above (see the discussion below).

None of degree, component-finding, k -core, clustering, or eigenvector centrality is compiled (Section 5.4): all five run entirely in Mata on the Stata side and entirely in *igraph*’s C core on the R side, at both sizes. Degree, component-finding, and k -core all stay comfortably fast in absolute terms even at 5,000 nodes (0.02–0.09s); clustering is a step slower (0.39s) but still well inside what a user would notice. Eigenvector centrality is a different story entirely, and is the one exception worth naming plainly: 13.3 seconds at 5,000 nodes, next to R’s 0.0063s, is a gap of roughly $2,100\times$ – far larger than anything else in this comparison, including the plain-Mata betweenness numbers discussed above relative to their own compiled kernel. That is exactly the interpreter-overhead-compounding-inside-an-iterative-loop signature Section 5.4’s $\geq 2\times$ threshold is designed to catch: *nwevcent* iterates a power method to convergence rather than making a single pass over the network, the same structural reason Brandes’ betweenness needed a compiled kernel. Two kernels are currently compiled (Section 5.4): unweighted betweenness and the ERGM MCMC sampler. On this evidence, eigenvector centrality is the clearest current candidate for the next one.

k -core’s own history is worth being explicit about, since Table 5 reports a very different number from an earlier measurement of the identical cell. That earlier measurement – 5,000 nodes, the same methodology, the same generated network – found 1.7 seconds; the number above is 0.017s, effectively tied with degree and component-finding. Both measurements are real; what changed in between is that the underlying sparse *nwkcore* implementation was itself improved as part of the package’s broader, ongoing development (visible in its own commit history), not through anything this article’s evidence-gating policy did. Whether that specific improvement was a direct response to the earlier measurement or an incidental byproduct of that broader effort is not something a benchmark run from outside the package can establish – but the outcome is worth keeping regardless of cause: not every gap this kind of measurement finds needs a compiled kernel to close. Sometimes a better Mata implementation is enough, and *nwkcore* turned out to be exactly that case.

Betweenness is the operation Section 5.4 already compiles, and the numbers discussed above show why: going from the dense, pre-sparse implementation (8.7s at 500 nodes) to sparse storage in plain Mata (3.9s at 500, 28.3s at 1,000, measured separately) roughly halves or more the cost at a given size but does not fix the underlying growth

rate – consistent with per-operation Mata interpreter overhead compounding inside Brandes’ tightly nested loops, exactly the condition the $\geq 2\times$ threshold from Section 5.4 is designed to catch. Once the compiled kernel is engaged, the remaining gap to R’s igraph shrinks a great deal but does not disappear, and it is worth being precise about exactly how it shrinks: Stata is 70% slower at 500 nodes (0.017s vs. 0.0100s, a 7ms difference) and 8% slower at 5,000 nodes (1.290s vs. 1.191s, a 99ms difference). The *absolute* gap grows with n , from 7ms to 99ms – it does not vanish – but it grows more slowly than the runtime itself does, so as a *percentage* of the total it shrinks from 70% to 8%. Both figures are correct at once; which one is more relevant depends on whether an analysis’ cost is dominated by one betweenness call or by many. To rule out a single noisy median as the explanation, betweenness was independently re-timed 15 times at each size in both languages; the pattern held in both cases (n=500: Stata 0.015–0.018s, R 0.009–0.011s; n=5,000: Stata 1.29–1.33s, R 1.17–1.28s). Chained together, the four betweenness figures discussed in this section – 8.705s (pre-sparse dense), 3.862s (sparse, plain Mata), 0.017s (sparse, compiled), and R’s own 0.0100s – are the concrete evidence behind the claim implicit in this article’s title (Figure 5): for the one operation currently compiled, **nwcommands** has gone from a genuine bottleneck at hundreds of nodes to within reach of a dedicated graph package (Csárdi and Nepusz, 2006) on the metric that matters, even if not yet fully at parity with it. Each stage isolates one architectural intervention: sparse storage alone roughly halves the cost without changing the growth rate; the compiled kernel is what actually closes most of the remaining gap.

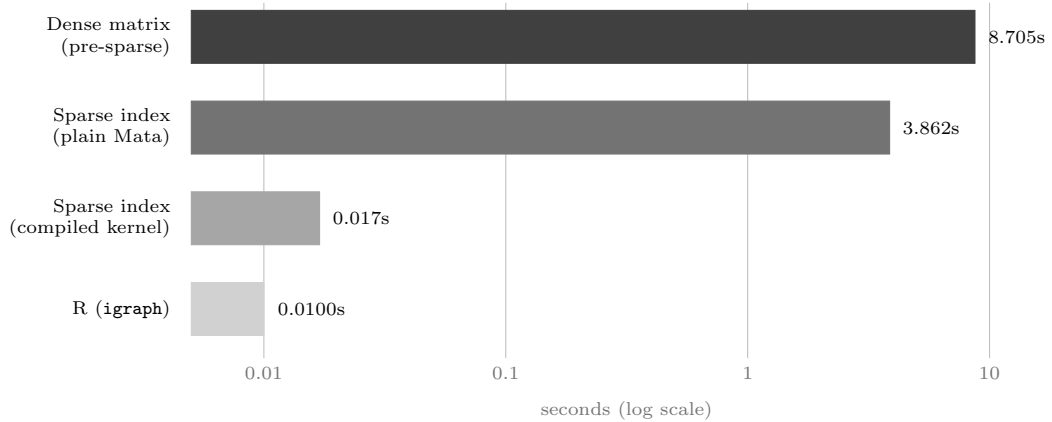


Figure 5: Unweighted betweenness centrality at 500 nodes, chained across this article’s three architectural interventions plus R’s **igraph** for reference (log scale; same figures as discussed in the surrounding text).

All of the above uses one network shape: undirected, unweighted, uniform random attachment at fixed average degree. Two further checks, at the same two sizes and using the same median-of-5 methodology, test whether this generalizes. A scale-free network (Barabási–Albert preferential attachment, which concentrates ties on a few high-degree hubs rather than spreading them uniformly – closer to what many empirical social

networks actually look like) shows the same pattern as the uniform-random case: Stata 0.015s/1.246s against R’s 0.010s/1.132s (50% slower narrowing to 10%). A directed network (same average out-degree, so twice the tie count of the undirected cases at a given n) shows a similar shape but a larger persistent gap: Stata 0.021s/1.758s against R’s 0.010s/1.138s (110% slower narrowing to only 55%, not single digits). Table 6 reports both alongside the uniform-random undirected figures already discussed, for comparison. The native-acceleration evidence is not an artifact of testing exclusively uniform-random undirected networks – compiled Stata remains substantially faster than plain Mata regardless of shape – but the specific degree to which it approaches R’s igraph does depend on network shape, and directed networks are currently the weaker case; whether that reflects the directed C kernel itself or the doubled edge count at matched n is not established here and would need its own profiling pass to separate.

Table 6: Betweenness runtime generalization check: same sizes and methodology as Table 4/5, two additional network shapes (seconds, median of 5 repetitions).

Network shape	500 nodes		5,000 nodes	
	R	Stata	R	Stata
Undirected, uniform random (Table 4/5)	0.0100	0.017	1.1910	1.290
Undirected, scale-free	0.0100	0.015	1.1320	1.246
Directed, uniform random	0.0100	0.021	1.1380	1.758

At 20,000 nodes (beyond either table above, by design for this comparison) the step that marshals the network’s tie list to the native plugin itself becomes the bottleneck rather than the native kernel; closing that gap is a smaller, separate implementation task and does not bear on the architecture this article describes.

6.3 Practical limits and current limitations

How large a network is usable in practice depends on which command is doing the work, since (Sections 5.1–5.2) different parts of `nwcommands` store and access a network differently. Sparse structural commands (`nwcomponents`, `nwdegree`, `nwclustering`, `nwneighbor`, unweighted `nwgeodesic`, and related commands) are proven at 100,000 nodes / 1,000,000 edges and practical considerably beyond that. Commands still requiring the full dense adjacency matrix (most weighted-distance calculations, `nwspectral`’s Laplacian analysis, and older algorithms not yet migrated to the sparse representation) are guarded at 20,000 nodes with an explicit error, but since both memory and running time for these commands scale with n^2 , comfortable use is realistically closer to a few thousand nodes. Loading a network as Stata variables (`nwload`, or the `xvars` option some commands accept) is bounded by Stata’s own variable-count ceiling rather than anything `nwcommands` imposes, and is in any case avoidable in the first place: as Section 2.5 described, analyzing a network’s own structure never touches Stata’s variable budget unless the user explicitly asks for it. The same backend architecture underlies `nwergm`; detailed validation and benchmarking of ERGM estimation, including its own comparison against R’s `ergm` package (Hunter et al., 2008), are reported separately. None of these figures are a promise for every network shape; a command’s own help file

is the authority on anything specific to it.

The architecture also has known limitations worth stating together rather than leaving scattered across earlier sections. Exactly two operations are currently compiled (Section 5.4): unweighted betweenness centrality and the ERGM MCMC sampler; every other operation, including eigenvector centrality’s iterative power method (Section 6.2, the current worst-case gap against R’s `igraph`), runs in Mata only. Temporal and two-mode declarations cannot currently be combined at `nwset` time, and only exact-timestamp event slicing (`nwatime`) is supported so far, not windowed or aggregated slicing (Section 3.3). The explicit-projection-or-redirect guard around two-mode input (Section 3.2) is applied to `nwdegree`, `nwclustering`, and `nwergm`, but not yet to the rest of the structural command family: `nwcomponents`, `nwbetween`, and `nwkcore`, checked directly, currently run on a two-mode network’s combined node set without erroring, redirecting, or warning, rather than requiring an explicit `nw2project` call the way the audited commands do. And at around 20,000 nodes (Section 6.2), the step that marshals a network’s tie list to the compiled betweenness kernel itself becomes the bottleneck rather than the kernel’s own computation, a separate, smaller implementation task not yet undertaken.

6.4 Correctness certification

Every command’s correctness is tracked in a living certification record rather than asserted informally: a feature progresses through four recorded stages – implemented, tested (an automated regression test in `cscripts/`, run in both interpreted and compiled mode), certified (verified against a small network with an analytically known answer, against edge cases – empty networks, isolates, disconnected components, directed and undirected variants, weighted ties, self-loops – and, where an implementation replaced a prior one, against byte-identical output on the same input), and documented (a current `.sthlp` reflecting any options the certification pass touched). The suite currently comprises 156 automated regression tests covering the command set, shipped in full as `cscripts/` within the package’s public repository alongside the `.ado` files they test – so any claim of correctness made in this article is backed by a test a reader can run and inspect directly, not merely asserted. Rather than re-running the entire suite after every change – a strategy that stopped scaling once a single sweep meant 180-190 sequential Stata invocations – each change is certified at a tier matched to its actual dependency radius, from a single command’s own test up to the full suite for changes touching widely shared core functions; an unexpected failure surfaced at a lower tier triggers escalation to the next one rather than being dismissed as unrelated.

7 Compatibility and migration from the 2014 prototype

An architectural redesign is of limited value if it invalidates existing analyses. Wherever possible, `nwcommands` therefore preserves the prototype’s established syntax while replacing the computational machinery beneath it. Superseded files are archived, not deleted, under an explicit, documented policy, and are never left on Stata’s `adopath`.

Where old and new syntax coexist, the package’s stated policy is to preserve the old syntax as a working alias while documenting the new, canonical form as primary

Table 7: What changed, feature by feature, from the 2014–2015 prototype architecture summarized in Table 1.

	2014–2015 prototype	nwcommands (this release)
Storage	Dense $n \times n$ Mata matrix, allocated unconditionally	Sparse CSR/CSC index, lazy size-guarded dense fallback
Network state	Loose global macros per network (<code>nwlab_s_N</code> , <code>nwdirected_N</code> , ...)	Persistent metadata on a Mata class object (<code>NWdef</code>)
Two-mode	Separate nw2* commands only	Declared network type (<code>is2mode</code>) and nw2* commands for two-mode-specific math
Temporal	Not represented	Metadata architecture: <code>time()</code> , <code>interval()</code> , <code>eventtime()</code>
Computation	Mata only	Mata (default) + evidence-gated native backend (2 kernels)
Large networks	Impractical above a few thousand nodes	Sparse-native construction, tested to 100k nodes/1M edges
Statistical models	nwergm (ERGM), shelling out to an external R installation	nwergm (ERGM), native Stata/Mata MCMC, no R dependency
Command count	92 (August 2015) (Grund, 2015)	130+ (Section 1)

– with one explicit exception: statistically questionable legacy treatment of weights or ties, once identified during the package’s internal audit, was corrected rather than silently preserved. Correctness is ranked above backward compatibility where the two conflict. There is no separate migration guide beyond the package’s own change log and style documentation; most users will find that scripts written against the package’s descriptive and structural commands continue to run unchanged.

8 Conclusions

There are two common ways to add network analysis to general-purpose statistical software. One is to provide individual commands that each accept an adjacency matrix or edgelist as input, computed and discarded independently of one another. The other is to introduce a network abstraction – an object with its own persistent identity and properties – that can be shared by descriptive, graphical, and statistical procedures alike. **nwcommands** takes the second approach.

That distinction matters because Stata’s traditional strengths and network analysis’s traditional weaknesses are complementary rather than competing. Stata is well established for rectangular data management, panel and multilevel data, survival and event-history analysis, causal inference, survey data, and reproducible research; dedicated network packages are well established for relational structure, but typically outside that same environment. **nwcommands** is not interesting because it duplicates what **igraph** (Csárdi and Nepusz, 2006), **statnet/ergm** (Handcock et al., 2008; Hunter et al., 2008), **RSiena** (Snijders et al., 2010), or **UCINET** (Borgatti et al., 2002) already do well – it does not attempt to match any of them feature for feature – but because it allows

relational structure to coexist, in one session and one object model, with all the other kinds of data researchers already analyze in Stata. Network data are increasingly combined with conventional individual-, dyad-, organization-, spatial-, and event-level data, and a researcher who otherwise conducts their empirical workflow in Stata should not have to leave it simply because part of that workflow involves a network.

The existence of `nwergm`, built on the same network-object architecture described here, is evidence that this extends beyond descriptive measures: statistical models whose outcome is itself a network – not only individual- or dyad-level outcomes computed from one – can sit on the same abstraction. Detailed treatment of that model is left to a separate article; its architectural dependence on the framework described here is the reason this article exists as a prerequisite for it.

Two limitations bound what this article claims, both stated explicitly in Section 6.3 rather than left implicit: the two-mode input guard is applied to three commands, not the whole structural command family, and exactly two operations are currently compiled, with eigenvector centrality the clearest evidence-based candidate for a third (Sections 5.4, 6.2). Neither is a reason to wait for a future release before adopting the architecture – every operation runs correctly in Mata regardless, and the compiled path is an accelerant, not a correctness requirement – but neither should be understated either.

Taken together, `nwcommands` turns a prototype collection of network-analysis commands into a common infrastructure for relational data in Stata. Sparse storage addresses the computational limitations of the dense representation documented in Section 1; persistent metadata allows commands to recognize directed, weighted, two-mode, and temporally indexed networks without the user re-specifying them; separation of the Stata interface from the computational backend permits specific, evidence-gated operations to use compiled native code without changing the user-facing workflow; and a command philosophy modeled deliberately on Stata’s own means that none of this has to be learned as a separate idiom. More broadly, it allows network structure to be incorporated into the same data-management, modeling, graphical, and reproducible-research environment already used for other forms of empirical analysis in Stata – not a faster version of an existing tool, but the first officially released, peer-reviewed version of `nwcommands`, and the first usable on the networks empirical research actually produces.

Availability

`nwcommands` is available at <https://github.com/thomas-u-grund/nwcommands>, including the tutorials and an issue tracker for reporting problems; Section 2.1 gives the installation commands and minimum Stata version. The software is free to install and use, including for commercial research; redistribution, modification, and commercial licensing require the author’s permission (see the repository’s own `LICENSE` file). Point-and-click dialog boxes – Stata’s standard `.dlg` mechanism – are available for 112 of the package’s 130+ commands, reachable from a Network Analysis menu that `nwinstall` adds to Stata’s menu bar; a dialog exposes the same options documented in the command’s own help file, so the two interfaces stay interchangeable rather than diverging.

Acknowledgments

The author thanks StataCorp for hosting a month-long visit in 2015, during which the persistent network-object architecture described in Section 5.5 was set up.

9 References

- Borgatti, S. P., M. G. Everett, and L. C. Freeman. 2002. Ucinet for Windows: Software for Social Network Analysis.
- Borgatti, S. P., M. G. Everett, and J. C. Johnson. 2018. *Analyzing Social Networks*. 2nd ed. Thousand Oaks, CA: SAGE.
- Brandes, U. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25(2): 163–177.
- Csárdi, G., and T. Nepusz. 2006. The igraph software package for complex network research. *InterJournal, Complex Systems* 1695. URL <https://igraph.org>.
- Freeman, L. C. 2004. *The Development of Social Network Analysis: A Study in the Sociology of Science*. Vancouver, BC: Empirical Press.
- Grund, T. U. 2014. nwcommands: Software Tools for the Statistical Modeling of Network Data in Stata. Computer software. URL <https://github.com/thomas-u-grund/nwcommands>.
- . 2015. nwcommands: Network Analysis in Stata. Unpublished book-length manual, version dated 18 August 2015.
- Handcock, M. S., D. R. Hunter, C. T. Butts, S. M. Goodreau, and M. Morris. 2008. statnet: Software Tools for the Representation, Visualization, Analysis and Simulation of Network Data. *Journal of Statistical Software* 24(1): 1–11.
- Hunter, D. R., M. S. Handcock, C. T. Butts, S. M. Goodreau, and M. Morris. 2008. ergm: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks. *Journal of Statistical Software* 24(3): 1–29.
- Newman, M. E. J. 2018. *Networks*. 2nd ed. Oxford: Oxford University Press.
- Padgett, J. F., and C. K. Ansell. 1993. Robust Action and the Rise of the Medici, 1400–1434. *American Journal of Sociology* 98(6): 1259–1319.
- Snijders, T. A. B., G. G. van de Bunt, and C. E. G. Steglich. 2010. Introduction to Stochastic Actor-Based Models for Network Dynamics. *Social Networks* 32(1): 44–60.
- Wasserman, S., and K. Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press.

About the author

Thomas U. Grund is at RWTH Aachen University.

A Command families

Rather than reproduce the package’s own help index as a directory listing, Table 8 summarizes the 14 command families `help nw_topical` organizes commands into, with representative commands from each – most already used somewhere in this article. The complete, current list (`help nw_topical` within Stata, or the package repository’s own documentation) is authoritative and not duplicated here: at the time of writing it indexes 125 commands, with a handful of more recently added ones – `nwclustering` (used throughout Section 6) among them – not yet folded into that index.

Table 8: Command families and representative commands (Section 4’s worked example already used several of these).

Family	Commands	Representative
Import/export	12	<code>nwset</code> , <code>nwfromedge</code> , <code>nwebuse</code>
Generators	21	<code>nwrandom</code> , <code>nwgen</code>
Information	7	<code>nwsummarize</code> , <code>nwcurrent</code>
Manipulation	18	<code>nwgenerate</code> , <code>nwreplace</code> , <code>nwdrop</code>
Centrality	7	<code>nwbetween</code> , <code>nwdegree</code> , <code>nwevcent</code>
Cohesion and components	9	<code>nwcomponents</code> , <code>nwkcore</code>
Community detection	3	<code>nwcommunity</code>
Positions and roles	11	<code>nwburt</code> , <code>nwbrokerage</code>
Paths and reachability	6	<code>nwgeodesic</code> , <code>nwneighbor</code>
Statistical models	5	<code>nwergm</code> , <code>nwcorrelate</code>
Other analysis utilities	7	<code>nwgen</code> , <code>nwgenvar</code>
Utilities	14	<code>nwload</code> , <code>nwsync</code>
Visualization	4	<code>nwplot</code> , <code>nwmovie</code>
Programming	1+	<code>nw_syntax</code> (Section 5.5)