# The Selection Gradients Project: Adaptive Geometry of Antagonism and Cooperation

I'm working on a paper about selection gradients and interactions - I'm not ready to upload it here, but I think I'll do the simulation here.

I'm using Sage to evaluate and analyze these models.

- Adaptive Geometry of Resource Competition: introducing the MacArthur-Levins resource competition model, and Sage code for it
  - Adaptive Geometry of Resource Competition: One Species, One Resource: analysis of the simple case
  - Adaptive Geometry of Resource Competition: Two Species, Two Resources: not so simple
  - Notes

This project also uses Sage code that's stored in an external Git repository:

- SageDynamics

## General results about selection gradients

For some reason a bunch of this is on Adaptive Geometry of Resource Competition, including review of adaptive dynamics, adaptive dynamics of constrained phenotypes, and the adaptive dynamics of $a$ and $k$, including the direct and indirect effects.

Here is the part about motion of the $(a_{ij}, a_{ji})$ pair.

When there are more than one or two species interacting, the only good way to visualize the motion of $a$ and $k$ terms is pairwise, so we'll be looking at $(a_{ij}, a_{ji})$ pairs moving on the classic plane of four quadrants. This is also how I've been presenting the result, as motion from the other three quadrants toward the mutualistic quarter. Here then is the motion of those pairs.

Using the expressions for motion of the matrix row $\mathbf{A}$, we introduce a little new notation: let

$D_{ik} = \gamma \hat{X}_i \partial_1 a(\mathbf{u}_i, \mathbf{u}_k) \partial_1 \mathbf{A}_i(\mathbf{u}_i)^T S(\mathbf{A}_i)$

be the $k$th entry of $D(\mathbf{u}_i)$, i.e. the direct effect of change in $\mathbf{u}_i$ on $a_{ik}$; and let

$I_{ik} = \gamma \hat{X}_k \partial_2 a(\mathbf{u}_i, \mathbf{u}_k) \partial_1 \mathbf{A}_k(\mathbf{u}_k)^T S(\mathbf{A}_k)$

be the $k$th entry of $I(\mathbf{u}_i)$, the indirect effect of change in $\mathbf{u}_k$ on $a_{ik}$.

NOTE the $D$ and $I$ notation in the other section is not acceptable: $\partial_2 \mathbf{A}_i(\mathbf{u}_k)$ is incoherent. Fix throughout. Possibly using notation like $\partial_2 a(\mathbf{u}_i, \mathbf{u}_k)$.

Anyway, the change in $a_{ij}$ is

$$\begin{aligned} \dot{a}_{ij} &= D_{ij} + I_{ij} \\ &= \partial_1 a(\mathbf{u}_i, \mathbf{u}_j) \dot{\mathbf{u}}_i + \partial_2 a(\mathbf{u}_i, \mathbf{u}_j) \dot{\mathbf{u}}_j \end{aligned}$$

and the change in the pair $(a_{ij}, a_{ji})$ is

$$\begin{aligned} \frac{d}{dt} \begin{pmatrix} a_{ij} \\ a_{ji} \end{pmatrix} = D_{(i,j)} + I_{(i,j)} &= \begin{pmatrix} D_{ij} \\ D_{ji} \end{pmatrix} + \begin{pmatrix} I_{ij} \\ I_{ji} \end{pmatrix} \\ &= \begin{pmatrix} \partial_1 a(\mathbf{u}_i, \mathbf{u}_j) \dot{\mathbf{u}}_i \\ \partial_1 a(\mathbf{u}_j, \mathbf{u}_i) \dot{\mathbf{u}}_j \end{pmatrix} + \begin{pmatrix} \partial_2 a(\mathbf{u}_i, \mathbf{u}_j) \dot{\mathbf{u}}_j \\ \partial_2 a(\mathbf{u}_j, \mathbf{u}_i) \dot{\mathbf{u}}_i \end{pmatrix} \end{aligned}$$

Q: how does this $D_{(i,j)}$ relate to $S$? Is there a meaningful $S_{(i,j)}$?

We also define

$S_{(i,j)} = \begin{pmatrix} \hat{X}_j \\ \hat{X}_i \end{pmatrix}$ (???),

which is the two entries of $S(\mathbf{A}_i)$ and $S(\mathbf{A}_j)$, respectively, corresponding to the entries of $D_{(i,j)}$, since the fuller direct effect vectors $D_i$, $D_j$ are considered to be $S_i$ and $S_j$ filtered through the biases of available variation; so $S_{(i,j)}$ may be considered an unconstrained version of $D_{(i,j)}$.

Consequently we think the $D$ entries are biased toward positive, while the $I$ entries are not. Note that the $D$ vector is *not* necessarily positive; it's only true that each entry $D_{ij}$ is an element of a vector that's in a half-space defined by positive dot product with the positive vector $S = \hat{X}$. There may be non-positive entries of the $D$ vector, required by tradeoffs between the elements of $\mathbf{u}_i$ (or by the form of $a()$??).

We are plotting

- the vectors $S_{(i,j)}$ (red), $D_{(i,j)}$ (green), $I_{(i,j)}$ (purple), and $d/dt(a_{ij}, a_{ji})$ (blue) at representative points
- the trajectory of $(a_{ij}, a_{ji})$

for all pairs $(i, j)$, or at least the half of them sorted with $i \leq j$.

Q: how to display the change in $k_i$? $(k_i, a_{ii})$ or something?

## Code

Here is Sage code that works with the SageDynamics code to analyze the dynamics of Lotka-Volterra coefficients.

```
from sage.all import *

from dynamicalsystems import *

from sage.symbolic.relation import solve
#from sage.symbolic.function_factory import function

class GeneralizedLotkaVolterraModel(PopulationDynamicsSystem):
    """The GLV model is dX_i/dt = (r_i + sum_j a_i_j X_j) X_i"""
    def __init__(self, x_indices = [], X = indexer('X'), r = indexer('r'),
      a = indexer_2d('a'),
      bindings = Bindings()):
        self._indexers = {'r': r, 'a': a, 'X': X}
        super(GeneralizedLotkaVolterraModel,self).__init__(
            [ ], x_indices, X, bindings=bindings )
    def flow(self):
        return self.make_flow(**self._indexers)
    def make_flow(self, X, r, a):
        return dict( (X[i],
          X[i]*(r[i] + sum( a[i][j]*X[j] for j in self._population_indices )))
          for i in self._population_indices)
    def interior_equilibrium_bindings( self, phenotype_bindings=Bindings() ):
        eq = self.bind( phenotype_bindings ).interior_equilibria()[0]
        eb = Bindings( { vhat:eq[vhat] for vhat in self.equilibrium_vars() } )
        #print 'equilibrium bindings:', eb
        return eb

class LotkaVolterraException(Exception):
    def __init__(self, message):
        self.value = message
    def __str__(self):
        return repr(self.value)

def get_coeff( expr, vars, power ):
    for c, p in expr.coefficients( vars ):
        if p == power:
            return c

class DerivedLotkaVolterraModel( GeneralizedLotkaVolterraModel ):
    """This is a LotkaVolterraModel made by decomposing the dynamics of
    some other population dynamics model into linear and quadratic terms.
    It has the attributes of a GeneralizedLotkaVolterraModel, plus some
    additional information about the original model."""
    def __init__(self, model, r_name='r', a_name='a', r_indexer=None, a_indexer=None ):
        self._original_model = model
        self._a_name = a_name
        self._r_name = r_name
        if a_indexer is None: a_indexer = indexer_2d(a_name)
        if r_indexer is None: r_indexer = indexer_2d(r_name)
        self._a_indexer = a_indexer
        self._r_indexer = r_indexer
        self.calculate_lv()
```

```
        super(DerivedLotkaVolterraModel,self).__init__(
            model._population_indices,
            X = model._population_indexer,
            r = r_indexer,
            a = a_indexer
        )
    def calculate_lv(self):
        model = self._original_model
        x_indexer = model._population_indexer
        aij_dict = {}
        vars = {v:1 for v in model._vars}
        for i in model._population_indices:
            xi = x_indexer[i]
            lvi = model._flow[xi].expand()
            print 'Inferring LV coefficients from', xi, 'equation:', lvi
            xic = get_coeff( lvi, xi, 1 )
            if xic is None: xic = 0
            ri = xic
            aij = get_coeff( lvi, xi, 2 )
            if aij is None: aij = 0
            aij_dict[self._a_indexer[i][i]] = aij
            lvi -= aij * xi**2
            for j in model._population_indices:
                if j != i:
                    xj = x_indexer[j]
                    ri = get_coeff( ri, xj, 0 )
                    if ri is None: ri = 0
                    aij = get_coeff( xic, xj, 1 )
                    if aij is None: aij = 0
                    aij_dict[self._a_indexer[i][j]] = aij
                    print self._a_indexer[i][j], ':', aij
                    lvi -= aij * xi * xj
            print self._r_indexer[i], ':', ri
            aij_dict[self._r_indexer[i]] = ri
            lvi -= ri * xi
            if lvi != 0:
                raise LotkaVolterraException( "Population dynamics has excess terms in " + str
            del vars[xi]
        if len(vars) > 0:
            raise LotkaVolterraException( "Population dynamics has extra variables: " + ', '.j
        print 'aij_dict:',aij_dict; sys.stdout.flush()
        self._A_bindings = Bindings( aij_dict )
    def set_population_indices(self, npi):
        super(DerivedLotkaVolterraModel, self).set_population_indices(npi)
        self.calculate_lv()
    def mutate(self, resident_index):
        mutant = self._original_model.mutate(resident_index)
        self.set_population_indices( self._original_model._population_indices )
        return mutant

class inner_curry(indexer):
    def __init__(self, ixr, j):
        self._f = ixr
        self. _j = j
    def __getitem__(self, i):
        return self._f[i][self._j]
```

```
class backward_curry_indexer_2d( indexer ):
    def __init__(self, ixr):
        self._f = ixr
    def __getitem__(self, j):
        # return a thing that maps i --> ixr[i][j]
        return inner_curry(self._f, j)

class LotkaVolterraAdaptiveDynamics( object ):
    '''Not an AdaptiveDynamicsModel class, but a helper that works with one.'''
    def __init__( self, ad, lv_model=None, r_name='r', a_name='a', r_name_indexer=None, a_name
        self._adaptivedynamics = ad

        print 'ad bindings:', self._adaptivedynamics._bindings
        print 'model bindings:', self._adaptivedynamics._popdyn_model._bindings
        print '_early_bindings:', self._adaptivedynamics._early_bindings
        print '_late_bindings:', self._adaptivedynamics._late_bindings

        if a_name_indexer is None: a_name_indexer = indexer_2d(a_name)
        if r_name_indexer is None: r_name_indexer = indexer(r_name)
        self._a_name_indexer = a_name_indexer
        self._r_name_indexer = r_name_indexer

        # make LV model here
        print 'make LV model'
        self._lv_model = DerivedLotkaVolterraModel( ad._popdyn_model, r_indexer=r_name_indexer

        print '_A_bindings:', self._lv_model._A_bindings

        # for now, this only supports AD with a single phenotype variable
        u_indexers = self._adaptivedynamics._phenotype_indexers

        if False:
            # sanity check the a values' functional form
            # using extended system, in case it's only 1-d otherwise
            extended_system = deepcopy( ad._popdyn_model )
            # TODO: mutate all populations, not just the first one
            mutant_index = extended_system.mutate( extended_system._population_indices[0] )
            extended_lv_model = DerivedLotkaVolterraModel( extended_system, r_name=r_name, a_n
            a_to_u_bindings = self._adaptivedynamics._bindings + self._adaptivedynamics._late_
            print 'a_to_u_bindings:', a_to_u_bindings
            u0s = column_vector( [ u_indexer[0] for u_indexer in u_indexers ] )
            uqis = column_vector( [ u_indexer[mutant_index] for u_indexer in u_indexers ] )
            a_func = a_to_u_bindings( extended_lv_model._indexers['a'][0][mutant_index] ).func
            print 'a_func:', a_func
            r_func = a_to_u_bindings( extended_lv_model._indexers['r'][0] ).function( *u0s )
            print 'r_func:', r_func
            for i in ad._popdyn_model._population_indices:
                uis = column_vector( u[i] for u in u_indexers )
                for j in ad._popdyn_model._population_indices:
                    ujs = column_vector( u[j] for u in u_indexers )
                    if a_to_u_bindings( extended_lv_model._indexers['a'][i][j] ) != a_func( *(
                        raise LotkaVolterraException(
                            'Model does not have consistent form a(u_i,u_j): ' +
                            str( a_to_u_bindings( extended_lv_model._indexers['a'][i][j] ) ) +
                            ' does not have the form of ' +
                            'a(%s, %s) = ' % (''.join(str(u[i]) for u in u_indexers), ''.join(
                            str( a_func( *(list(uis) + list(ujs)) ) ) )
```

```
                if a_to_u_bindings( extended_lv_model._indexers['r'][i] ) != r_func( *uis
                    raise LotkaVolterraException(
                        'Model does not have consistent form r(u_i): ' +
                        str( a_to_u_bindings( extended_lv_model._indexers['r'][i] ) ) +
                        ' does not have the form of ' +
                        str( r_func( *uis ) ) )
                for k in ad._popdyn_model._population_indices:
                    if len( extended_lv_model._A_bindings( extended_lv_model._indexers['a'
                        raise LotkaVolterraException( 'Model has super-quadratic terms in


        print 'make LV adaptive dynamics'
        print 'population vars', self._lv_model.population_vars()
        print 'population vars', self._lv_model._original_model.population_vars()
        formal_equilibrium = Bindings( { v : hat(v) for v in self._lv_model.population_vars()
        self._lv_adap = AdaptiveDynamicsModel(
            self._lv_model,
            [ self._lv_model._indexers['r'] ] + [ backward_curry_indexer_2d(self._lv_model._in
            equilibrium = formal_equilibrium )
        # todo: what to do with multiple phenotype indexers?
        print 'make LV evolution bindings'
        #ui = self._adaptivedynamics._phenotype_indexers[0]
        us = self._adaptivedynamics._phenotype_indexers
        # _A_to_function_bindings expands the LV equations
        # into functions of u_i, e.g. from a_i_j to a(u_i,u_j)
        # very necessary so that partial derivatives can be taken
        from sage.symbolic.function_factory import function
        _A_to_function_dict = [
          ( a_name_indexer[i][j],
            self._lv_model._A_bindings( function( str( a_name_indexer[i][j] ) )( *([u[i] for u
          for i in self._lv_model._population_indices
          for j in self._lv_model._population_indices ] + [
          ( r_name_indexer[i],
            self._lv_model._A_bindings( function( str( r_name_indexer[i] ) )( *(u[i] for u in
          for i in self._lv_model._population_indices ]
        print '_A_to_function_dict:', _A_to_function_dict
        self._A_to_function_bindings = Bindings( dict( _A_to_function_dict ) )

        A_late_bindings = self._lv_model._A_bindings + self._adaptivedynamics._bindings + self
        self._A_function_expansion_dict = dict( [
            ( a_name_indexer[i][j],
                A_late_bindings( a_name_indexer[i][j] )
                    .function( *([ u_indexer[i] for u_indexer in u_indexers ] + [ u_indexer[j]
                for i in self._lv_model._population_indices
                for j in self._lv_model._population_indices
        ] + [
            ( r_name_indexer[i],
                A_late_bindings( r_name_indexer[i] )
                    .function( *([ u_indexer[i] for u_indexer in u_indexers ]) ) )
                for i in self._lv_model._population_indices
        ] )
        print '_A_function_expansion_dict:', self._A_function_expansion_dict
        self._A_function_expansion_bindings = Bindings( FunctionBindings(
            self._A_function_expansion_dict ) )
        print '_A_function_expansion_bindings:', self._A_function_expansion_bindings
        # _phenotypes_to_fn_bindings: changes u_i to u_i(t)
        # is used in all the below methods, so that du_i(t)/dt works
```

```
        self._phenotypes_to_fn_bindings = Bindings( dict(
            ( u[i], function( str(u[i]), self._adaptivedynamics._time_variable, latex_name=lat
            for i in self._lv_model._population_indices
            for u in self._adaptivedynamics._phenotype_indexers ) )
        # _phenotypes_from_fn_bindings: the inverse, change u_i(t) to u_i
        self._phenotypes_from_fn_bindings = Bindings(
            { v:k for k, v in self._phenotypes_to_fn_bindings.items() } )
        print '_phenotypes_from_fn_bindings:', self._phenotypes_from_fn_bindings
    def A( self, i ):
        '''The 'interaction phenotype' vector of Lotka-Volterra coefficients
        that are affected by selection on population i.  These are r_i and a_ij
        for all j.'''
        # this is a vector, intended to be treated as a column vector.
        return vector(
            [ self._lv_model._indexers['r'][i] ] +
            [ self._lv_model._indexers['a'][i][j] for j in self._lv_model._population_indices
    def S( self, A ):
        '''The selection gradient corresponding to the 'interaction phenotype'
        A.'''
        # This is a vector, intended to be treated as a column vector.
        S = vector( [ self._lv_adap._S[a] for a in A ] )
        #print 'S:', S
        return S
    def d1A( self, i ):
        '''"Direct effect": derivative of A(u_i) wrt u_i with u_i in the "patient"
        position, as the first argument of a(.,.).'''
        # this is a matrix, with row indices the same as in A and column
        # indices indexing the entries of the u phenotype vector.
        A_ij = self._A_to_function_bindings( self.A(i) )
        uis = column_vector( [ u[i] for u in self._adaptivedynamics._phenotype_indexers ] )
        xx = self._adaptivedynamics.fake_population_index()
        uxs = column_vector( [ u[xx] for u in self._adaptivedynamics._phenotype_indexers ] )
        #print 'hack A_ij: from', A_ij
        A_ij_hacked = A_ij.apply_map( lambda x: x.subs_expr( function( str( self._a_name_index
        #print 'to', A_ij_hacked
        #print 'd1A: derivative of $%s$ wrt $%s$' % (latex(A_ij_hacked), latex(uis))
        d1a = matrix( [ [ aij.derivative( u ) for u in uis ] for aij in A_ij_hacked ] )
        #print 'is $%s$' % latex( d1a )
        d = d1a
        for ux, ui in zip( uxs, uis ):
            d = d.subs( ux == ui )
        #print '; $%s$' % latex( d )
        d = self._A_function_expansion_bindings( d )
        #print ': $%s$' % latex( d )
        return d
    def direct_effect( self, i ):
        # the component of change in A due to direct selection on population i
        # a vector
        return ( self._phenotypes_to_fn_bindings( self.d1A(i) ) * derivative(
                self._phenotypes_to_fn_bindings( vector( [ u[i] for u in self._adaptivedynamic
                self._adaptivedynamics._time_variable
            ) )
    def d2A_component( self, i, j ):
        '''Component of "Indirect effects" on "patient" u_i, due to "agent" u_j.
        When j is i, this includes only the "agent" role of u_i, which is where
        it appears as the second argument of a().'''
        # this is a matrix, with row indices the same as in A and column
```

```
        # indices indexing the entries of the u phenotype vector.
        A_ij = self._A_to_function_bindings( self.A(i) )
        u = self._adaptivedynamics._phenotype_indexers[0]
        if j == i:
            uis = [ u[i] for u in self._adaptivedynamics._phenotype_indexers ]
            xx = self._adaptivedynamics.fake_population_index()
            uxs = [ u[xx] for u in self._adaptivedynamics._phenotype_indexers ]
            A_ij_hacked = A_ij.apply_map( lambda x: x.subs_expr( function( str( self._a_name_i
            #print A_ij,', with', function( self._lv_model._a_name )( *(uis+uis) ), ' => ', fu
            #print 'd2A: derivative of $%s$ wrt $%s$' %(latex(A_ij_hacked), latex(uxs))
            d2A_hacked = matrix( [ [ aij.derivative( uxi ) for uxi in uxs ] for aij in A_ij_ha
            #print 'is $%s$' % latex(d2A_hacked)
            d2A = Bindings( { uxi:uii for uxi, uii in zip(uxs, uis) } )( d2A_hacked )
            #print 'on diagonal: $\partial_2 A = %s$\n\n' % latex( d2A )
        else:
            #print 'd2A: derivative of $%s$ wrt $%s$' %(latex(A_ij), latex(u[j]))
            d2A = matrix( [ [ aij.derivative( u[j] ) for u in self._adaptivedynamics._phenotyp
            #print 'is $%s$'% latex( d2A )
            #ltx.write( 'off diagonal: $\partial_2 A = %s$\n\n' % latex( d2A ) )
        return self._A_function_expansion_bindings( d2A )
    def indirect_effect( self, i ):
        # component of the motion of A_i due to selection on all "agents" that
        # act on population i
        # a vector
        components = [ self._phenotypes_to_fn_bindings( self.d2A_component( i, j ) ) * derivat
                self._phenotypes_to_fn_bindings( vector( [ u[j] for u in self._adaptivedynamic
                self._adaptivedynamics._time_variable
            ) for j in self._lv_model._population_indices ]
        #import operator
        return reduce( operator.add, components )
    def dudt_bindings( self ):
        # TODO: AD class should provide one of these, and
        # use it internally as well
        # TODO: make matrix * column_vector work
        dudts = {
            i:( SR('gamma') * hat( self._lv_model._population_indexer[i] ) *
                self.d1A(i).transpose() * self.S( self.A(i) ) )
              for i in self._lv_model._population_indices
        }
        #print 'dudts:', dudts
        return Bindings( {
            derivative(
                self._phenotypes_to_fn_bindings( us[i] ),
                self._adaptivedynamics._time_variable
            ): dudts[i][j]
              for i in self._lv_model._population_indices
              for j,us in enumerate(self._adaptivedynamics._phenotype_indexers)
        } )
    def dAdt( self, i ):
        # the motion of the 'interaction phenotype' A_i.  This is not computed
        # using the above components, but by using the chain rule with the
        # derivative of the constraint phenotype u_i.
        # A vector.
        Ai = self.A(i)
        Au = self._adaptivedynamics._bindings( self._lv_model._A_bindings( Ai ) )
        #import operator
        dAdus = [ self._phenotypes_to_fn_bindings( diff( Au, u[j] ) ) for u in self._adaptived
```

```
        dudts = [ diff( self._phenotypes_to_fn_bindings( u[j] ), self._adaptivedynamics._time_
        dAdts = [ vector( dAiduj * dujdt for dAiduj in dAduj ) for dAduj, dujdt in zip( dAdus,
        d = reduce( operator.add, dAdts )
        #print 'dAdt: diff of Au is $%s$' % latex( d )
        sys.stdout.flush()
        return d
    def A_index( self, j ):
        # help for using the above vectors: e.g. self.A(i)[self.A_index(j)]
        # is the a_ij term.
        try: self._A_index_dict.keys()
        except AttributeError:
            self._A_index_dict = dict( (v,i) for i,v in enumerate( [0] + self._lv_model._popul
        return self._A_index_dict[j]
    def A_pair( self, i, j ):
        return vector( [
            self.A(i)[self.A_index(j)],
            self.A(j)[self.A_index(i)]
        ] )
    def S_pair( self, i, j ):
        return vector( [
            self.S( self.A(i) )[self.A_index(j)],
            self.S( self.A(j) )[self.A_index(i)]
        ] )
    def D_pair( self, i, j ):
        return vector( [
            self.direct_effect( i )[self.A_index(j)],
            self.direct_effect( j )[self.A_index(i)]
        ] )
    def I_pair( self, i, j ):
        return vector( [
            self.indirect_effect( i )[self.A_index(j)],
            self.indirect_effect( j )[self.A_index(i)]
        ] )
    def dAdt_pair( self, i, j ):
        return vector( [
            self.dAdt( i )[self.A_index(j)],
            self.dAdt( j )[self.A_index(i)]
        ] )

def plot_aij_with_arrows( evol_trajectory, lvad, filename=None, scale=1, bindings=Bindings(),
    resolve_A_bindings = (
        lvad._A_to_function_bindings +
        lvad._A_function_expansion_bindings +
        lvad.dudt_bindings() +
        lvad._phenotypes_from_fn_bindings
    )
    aap = Graphics()
    for i in lvad._lv_model._population_indices:
        for j in lvad._lv_model._population_indices:
            aap += evol_trajectory.plot(
                lvad._lv_model._A_bindings( lvad._lv_model._a_indexer[i][j] ),
                lvad._lv_model._A_bindings( lvad._lv_model._a_indexer[j][i] ),
                color='red', **options
            )
            # plot 4 arrows per (i,j) point
            pp0 = evol_trajectory._timeseries[0] + bindings + lvad._adaptivedynamics._bindings
            Aij = (
```

```
                pp0( resolve_A_bindings( lvad.A(i)[lvad.A_index(j)] ) ),
                pp0( resolve_A_bindings( lvad.A(j)[lvad.A_index(i)] ) )
            )
            print 'A:', Aij
            S = (
                pp0( resolve_A_bindings( lvad.S( lvad.A(i) )[lvad.A_index(j)] ) ),
                pp0( resolve_A_bindings( lvad.S( lvad.A(j) )[lvad.A_index(i)] ) )
            )
            print 'S:', S
            if S[0] != 0 or S[1] != 0:
                aap += arrow( Aij, ( a+scale*s for a,s in zip(Aij, S) ), color='red' )
            D = (
                resolve_A_bindings( lvad.direct_effect( i )[lvad.A_index(j)] ),
                resolve_A_bindings( lvad.direct_effect( j )[lvad.A_index(i)] )
            )
            print 'D:', D
            D = (
                pp0( D[0] ),
                pp0( D[1] )
            )
            print 'D:', D
            if D[0] != 0 or D[1] != 0:
                aap += arrow( Aij, ( a+scale*d for a,d in zip(Aij, D) ), color='green' )
            I = (
                resolve_A_bindings( lvad.indirect_effect( i )[lvad.A_index(j)] ),
                resolve_A_bindings( lvad.indirect_effect( j )[lvad.A_index(i)] )
            )
            print 'I:', I
            I = (
                pp0( I[0] ),
                pp0( I[1] )
            )
            print 'I:', I
            if I[0] != 0 or I[1] != 0:
                aap += arrow( Aij, ( a+scale*i for a,i in zip(Aij, I) ), color='purple' )
            d = (
                pp0( resolve_A_bindings( lvad.dAdt( i )[lvad.A_index(j)] ) ),
                pp0( resolve_A_bindings( lvad.dAdt( j )[lvad.A_index(i)] ) )
            )
            print 'dAdt:', d
            if d[0] != 0 or d[1] != 0:
                aap += arrow( Aij, ( a+scale*d for a,d in zip(Aij, d) ), color='blue' )
    if filename is not None:
        aap.save( filename, figsize=(5,5), **options )
    return aap
```

Here are makefiles that are used by all the code on the various pages in this project.

```
%.sage.step : %.sage-inline sage-inline.mk
        echo '# produces: $*.sage.out.tex' > $@
        echo 'from dynamicalsystems import latex_output' >> $@
        echo 'ltx = latex_output( "$*.sage.out.tex" )' >> $@
        cat $< >> $@
        echo 'ltx.close()' >> $@

WW_SI_SOURCE_FILES = $(filter %.sage-inline,$(WW_THIS_PROJECT_SOURCE_FILES))
```

```
WW_SI_MK_FILES = $(patsubst %.sage-inline,%.sage.mk,$(WW_SI_SOURCE_FILES))
$(info Including .mk files from .sage-inline files: $(WW_SI_MK_FILES))
include $(WW_SI_MK_FILES)
```