Mathematical Programming for Ecological Optimization Problems

Richard Barnes

github.com/r-barnes/ecoopt

Contents

1 Introduction

An organism's life history is the sequence of events related to survival and reproduction that occur from birth through death. Considered another way, an organism's life history is the time series of how it allocates energy and resources over its lifespan. We expect that allocation strategies of an organism are shaped by evolution to maximize reproductive fitness.

In modeling life history, we would like to recover a policy function $\pi(X, t)$ which, given the organism's current allocations and anticipated future needs X, determines how incoming energy at time t should be

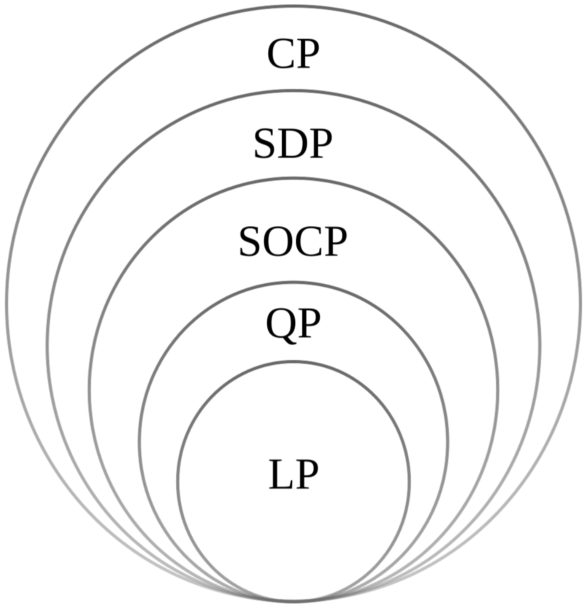


Figure 1: The hierarchy of convex programming showing Linear Programming (LP), Quadratic Programming (QP), Second-Order Cone Programming (SOCP), Semi-definite Programming (SDP), and Convex Programming (CP).

allocated. Finding such a function is challenging and the field has typically relied on optimal control theory and applications of Pontryagin's maximum principle to do so. This has required increasing mathematical sophistication over time [7, 9–11]. This paper recasts previous theory in the form of convex optimization problems which can be solved numerically. This has several advantages:

- Lower barriers to entry. Applying optimal control theory requires significant mathematical sophistication. This barrier to entry makes it unnecessarily challenging for ecologists to include theory in their work.
- Faster ideation and exploration. Even for those who have mastered the techniques of optimal control theory, applying it to a particular case can still be time-consuming and seemingly simple modifications to a model can rapidly move it into the realm of intractibility. The techniques here make it easy to investigate model variants quickly.
- Powerful solvers. The techniques presented here rely on a powerful toolbox of solvers which can quickly find optimal numerical solutions to even large problems. This means that instantiating seemingly complex problems in computer code is easy, even for beginners.
- Guarantees. Most of the examples in this paper translate life history problems defined by differential equations into convex optimization problems. The solvers for these problems have several desirable properties. (1) If a problem is non-convex, the solver can detect this and refuses to attempt a solution. (2) If the problem is convex the solver is guaranteed to either find the optimal solution or return a certificate that the problem is either unbounded (arbitrarily good solutions are possible) or infeasible (no solution is possible).

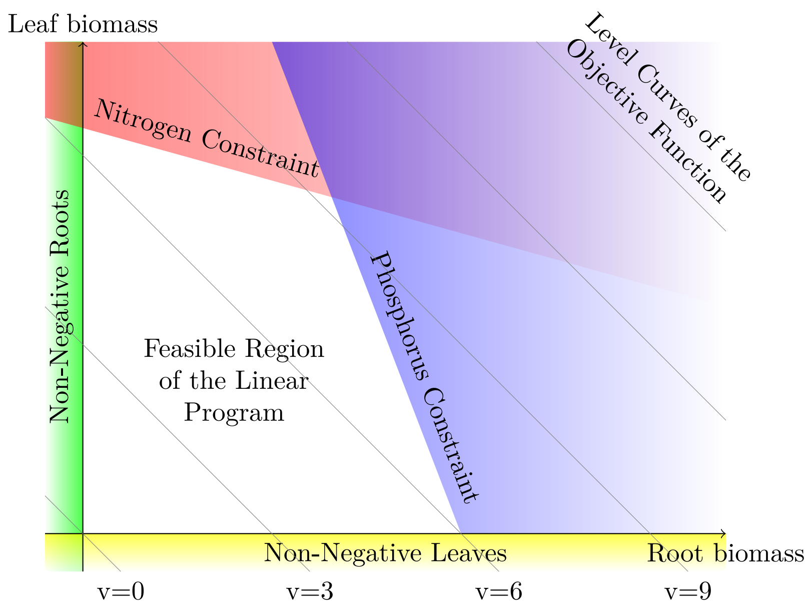


Figure 2: A graphical representation of a linear program.

2 Mathematical Programming

The basic idea behind *mathematical programming* is that we will specify an *objective function* whose value we wish to minimize (or maximize) by finding the best values of a set of *decision variables* whose values are subject to *constraints* which combine the decision variables and *parameters* (fixed numeric values which may vary from one instance of the problem to the next) using standard mathematical functions and either inequalities ($<, \leq, \geq, >$) or equalities ($=$). Once the objective and constraints have been specified, we can pass the problem off a powerful software program called a solver which determines the optimal values of the decision variables.

Unfortunately, except for special cases, solving mathematical programs is extremely hard. Fortunately, the special cases are vast and include many problems of interest. Below we discuss some special cases of interest here moving from the most- to least-constrained problem space, as shown in Figure 1.

2.1 Linear Programming

The simplest case is *linear programming*. In a linear program, the objective and every constraint consists of linear equations. Figure 2 shows a graphical representation of a particular linear program described below and hints out how it might be solved.

Imagine we have a plant whose reproductive success is determined by the biomass of its roots and leaves. The plant's goal is to grow so that the amount of root and leaf biomass it has maxmizes its reproductive output. The particular amounts of biomass it produces are its strategy.

If we were to plot aspects of this plant, we could do so by putting root biomass (R) on the x-axis and leaf biomass on the y-axis (L). With no further information, the space of possible solutions therefore spans the entire R-L plane. Obviously, though, no part of our plant cannot have negative biomass, so we add the constraints $R \geq 0$ and $L \geq 0$. This rules out three quadrants of the plane, which we indicate by adding green and yellow shading to Figure 2. The best strategy for our plant, whatever it is, must lie in the upper right quadrant. Now, let us say that a plant requires phosphorus P to produce its leaves and roots and that

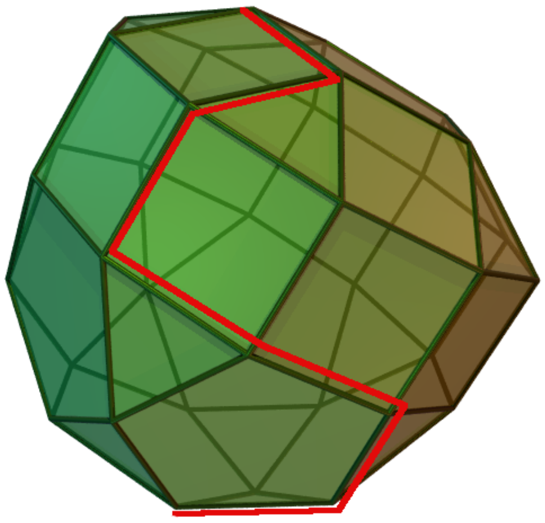


Figure 3: The simplex method in three dimensions.

the phosphorus requirements of the two differ; this implies the constraint $aR + bL \leq P$. This constraint rules out any solution in the blue-shaded region. A similar constraint might apply to nitrogen, ruling out solutions in the red-shaded region.

Having come this far, we now have a clearly defined *feasible region*, shown in white in Figure 2. The best strategy of the plant must lie somewhere in this region because any strategy outside of it would violate at least one constraint.

Finally, let us say that reproductive output, formally known as the objective function, is some linear function $cR + bL$ of leaf and root biomass—if this seems overly simplistic, remember, this is an example to illustrate mathematics, not a model we are actually interested in using. If we consider the level curves of the objective function—the lines obtained in Figure 2 by taking $cR + bL = 0$, $cR + bL = 3$, and so on—we see that they define both a gradient and intersect the constraints.

With some reflection, we can see that the maximum value of the objective function must occur at the intersection of two or more constraints. It is obvious from the picture that the objective value cannot have a maximum value in the interior of the feasible region since it has a gradient there and any point in the interior is adjacent to another interior or boundary point that would be farther along this gradient. Therefore, we know the objective is maximized on the boundary. The isolines of the objective could be exactly parallel to one of the boundary segments, but, in this case, both end points of the segment will have the same objective value and we can choose on arbitrarily.

This implies a simple algorithm for solving linear programs: find any vertex on the boundary, consider vertices linked to it by constraints, move to the vertex with the highest objective value, and repeat until no neighbouring vertex is better than the one you're at. As shown in Figure 3, in higher dimensions the feasible region is bounded by planes or hyperlanes and the feasible region, rather than being a polygon, is called a polytope. Still, the algorithm is the same: find a vertex and walk along edges until the best vertex is found.

The algorithm described above is called the *simplex algorithm*. Interestingly, while it is known that all linear programs can be solved efficiently (ie, in polynomial time) using another technique called the ellipsoidal algorithm, there are problems for which the simplex algorithm will perform very badly. In practice, however, these problems are rare and must be deliberately constructed. It's been said that at one point more compute

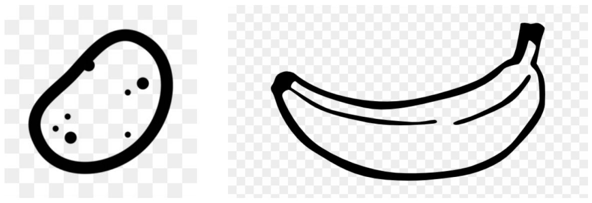


Figure 4: Examples of a convex space (a potato) and a non-convex space (a banana).

power was devoted to running the simplex algorithm than any other program—a testament both to the practical value of the algorithm as well as linear programming itself.

2.2 Convex Programming

As Figure 4 shows, the mathematical difference between a potato and a banana is that the potato is convex whereas the banana is non-convex; that is, any two points within the potato can be connected with a line which does not pass outside of the potato. In contrast, the banana contains points which can only be connected by a line passing outside of the banana.

Some reflection on our linear programming example reveals that the feasible region of a linear program is always a convex polytope. This is not an accident. If we draw isolines over a banana-shaped region, we find vertices of equal value that cannot be reached from each other. In such a situation the simplex algorithm no longer works and we have entered the *much* harder-to-solve space of non-convex programming. While every convex problem is, in some sense, the same, each non-convex problem is non-convex in its own way.

Convexity is key to efficient mathematical programming. Since the intersection of two convex sets is itself a convex set, we can compose convex objects to form feasible spaces with properties that make them amenable to efficient maximization. Unfortunately, we don't know efficient solution algorithms for all convex objects, so in practice we often have to find a way to re-express our problems using convex forms that we know.

Classes for which we do have efficient algorithms in include:

- *Linear programs (LP)*, as above.
- *Quadratic programs (QP)*, which includes linear programming as a special case, and is the class of all programs with linear constraints and an objective function of the form $\frac{1}{2}x^TQx + q^Tx$ where Q is a constant matrix, q is a constant vector, and x is a vector variable.
- *Quadratically constrained quadratic programs (QCQP)*, which are similar to quadratic programs, but may include quadratic constraints.
- *Second-order cone programs (SOCP)*, which can have linear or quadratic objectives and constraints of the form $\|Ax + b\|_2 \leq cx + d$, where A is a constant matrix, b and c are constant vectors, d is a constant, and x is a variable vector.
- *Semi-definite programs (SDP)* where we wish to find a positive semi-definite matrix that satisfies an objective and constraints.
- *Relative Entropy Programming (EXP)*, a subset of cone programming (CP) containing SOCPs defined by constraints of the form $ye^{xy} \leq z, y > 0$. See Chandrasekaran and Shah [3] for an overview.
- *Power Cone Programming (POW)*, defined by constraints of the form $\prod_i x_i^{\alpha_i} \geq \sqrt{\sum_i x_i^2}, x_i \geq 0$.
- *Geometric Programming (GP)*, geometric programming is defined by an objective $f_0(x)$ and constraints $f_i(x) \leq 1$ and $g_j(x) = 1$ where f are posynomials and g are monomials. Monomials take the form

Solver	LP	QP	SOCP	SDP	EXP	POW	MIP	Cost
CBC[1]	X						X	Free
GLPK[2]	X							Free
GLPK_MI[3]	X						X	Free
OSQP[4]	X	X						Free
CPLEX[5]	X	X	X				X	Free for Academics, \$199/mo
NAG[6]	X	X	X					
ECOS[7]	X	X	X		X			Free
GUROBI[8]	X	X	X				X	Free for Academics, \$\$
MOSEK[9]	X	X	X	X	X	X	X*	\$2,050 + \$512/yr
CVXOPT[10]	X	X	X	X				Free
SCS[11]	X	X	X	X	X	X		Free
SCIP[12]	X	X	X				X	Free
XPRESS[13]	X	X	X				X	Free for Academics, \$\$

Table 1: Solvers for mathematical programming and their capabilities. (*) Except mixed-integer SDP.

$cx_1^{a_1}x_2^{a_2}\dots$ with $c > 0$ and $a_i \in \mathcal{R}$. Posynomials are sums of monomials. A good review of geometric programming is given by Boyd et al. [1].

As detailed below, LPs are the easiest and fastest programs to solve and can include millions of variables and constraints. QP and QCQP programs are harder, so the programs we can solve with them are smaller. SOCP is harder still, but still very do-able using even free software, though the size of the programs we can solve is somewhat smaller. SDPs, EXP, and POW are active areas of research and even relatively small problems can be difficult.

The most exotic programs we'll use in this paper are SOCPs.

2.3 Mixed-Integer Programming

Integer programming (IP) arises when an optimization problem has one or more variables which are constrained to take only integer values. When such variables are mixed with continuous linear variables we have Mixed-Integer Linear Programming (MILP). If second-order cones are involved, then we have a MISOCP. The general class of such problems—those with both discrete and continuous variables—is called mixed-integer programming (MIP).

Integer programming belongs to a class of problems called NP for which there are no known efficient algorithms and for which many people think there will *never* be efficient algorithms. Nonetheless, many solvers are able to solve small instances of integer problems, but often only at the cost of substantial compute time. In this paper we try to avoid MIPs when we can or limit the number of discrete variables when we can't.

2.4 Solvers

Table 1 lists various open source and commercial solvers. As discussed in § 2.2, there is a gradient to the ease with which problems can be solved. Even large instances of linear programs can be solved for free by any of a large variety of software. In constrast, if we wish to solve an MISOCP problem, the only free options are SCIP, XPRESS, and CPLEX. Obtaining a CPLEX academic license is challenging and SCIP is brutally slow for this sort of problem, leaving XPRESS as the only viable option. In this paper, we focus on classes of problems for which good solvers are available.

2.5 Domain Specific Languages

Domain Specific Languages (DSLs) are programming languages or libraries that are designed to make expressing a particular type of problem easier. All of the solvers mentioned above have their own DSL, but using the solver's DSL directly means you're stuck with that solver even if it turns out not to work well for your particular problem. Fortunately, there are DSLs which allow you to express problems in such a way that they are easily passed to *any* solver. In this paper we will use two DSLs both for practical purposes (sometimes one works better than the other) as well as to demonstrate the generic nature of the methods we discuss.

- **cvxpy** (`https://www.cvxpy.org/`) is a Python-based DSL used for expressing convex and MIP programs. It's easy to use and provides helpful feedback on whether or not programs are convex and, if not, where the non-convexity has arisen. It has a large developer community and strong ties to academic research groups who regularly add new features to it.
- **JuMP** (`https://jump.dev/JuMP.jl/stable/`) is a Julia-based DSL used for expressing arbitrary mathematical programs, including convex and MIP. It has a somewhat steeper learning curve than cvxpy, but with the advantage of being able to handle much larger programs than cvxpy.

3 Life History Problems

3.1 What is the optimal single-season growth strategy?

Mirmirani and Oster [9] pose the following problem: consider a plant whose life cycle plays out over a single season of length T. Let $P(t)$ be the plant's biomass at time t starting at an initial value of $P(0) = P_0$ (the seed weight). During the season the plant can choose to allocate its resources (photosynthate) to make more biomass (increasing its growth rate) or it can allocate resources to reproductive biomass $S(t)$ (seeds) which directly increase its reproductive fitness. If the supply of resources is constant, what is the optimal reproductive strategy of the plant?

We can formalize the problem like so:

$$\begin{aligned}
\max_{u(t)} \quad & S(T) \\
\text{s.t.} \quad & \dot{P} = \hat{r}u(t)P - \mu P, \\
& \dot{S} = \tilde{r}(1-u(t))P - \nu S, \\
& P, S \geq 0, \\
& P(0) = P_0, \\
& S(0) = 0, \\
& u(t) \in [0,1]
\end{aligned} \tag{1}$$

where $\hat{r}$ and $\tilde{r}$ are conversion efficiencies, μ and ν are loss of biomass (perhaps by grazing, tissue senescence, or seed predation), and u is a *control variable* bound to the range $[0,1]$ indicating a hard trade-off between incoming energy (photosynthate) being allocated to productive versus reproductive tissues.

Mirmirani and Oster [9] show how to solve this problem using techniques from optimal control theory; in order to solve it as a mathematical program, we need to make a few changes which will be standard for the rest of this paper. Our first step is to **discretize the problem** using the forward Euler method (see **??**)

this gives the following system:

$$\begin{array}{lrll}
\max\limits_{u(t)} & S(T) & & \\
\text{s.t.} & P_{t+1} = & P_t + \Delta t\left(\hat{r} u_t P_t - \mu P_t\right) & \forall t, \\
& S_{t+1} = & S_t + \Delta t\left(\tilde{r}(1-u_t) P_t - \nu S_t\right) & \forall t, \\
& P_t, S_t \geq & 0 & \forall t, \\
& P_{t=0} = & P_0, & \\
& S_{t=0} = & 0, & \\
& u_t \in & [0,1] & \forall t
\end{array} \tag{2}$$

where Δt a constant scalar representing the length of the time interval between any two time points t and $t+1$.

Next, we need to **add a dimension to the control**. In our current system $u_t P_t$ and $(1-u_t)P_t$ are both products of variables. From § 2, we recognize these as forming at least a quadratic system, which is hard to solve; if we retain the product we also have to worry about the convexity of the system. It is best to eliminate such products where we can. In this case, it is easy to do so. Our strategy will be to add a new control variable (equivalent to adding a dimension to the control vector) and require that the control vector sum to the incoming energy (P_t, in this case) at each timestep. This gives

$$\begin{array}{lrll}
\max\limits_{u(t)} & S(T) & & \\
\text{s.t.} & P_{t+1} = & P_t + \Delta t\left(\hat{r} u_{1,t} - \mu P_t\right) & \forall t, \\
& S_{t+1} = & S_t + \Delta t\left(\tilde{r} u_{2,t} - \nu S_t\right) & \forall t, \\
& P_t, S_t \geq & 0 & \forall t, \\
& P_{t=0} = & P_0, & \\
& S_{t=0} = & 0, & \\
& u_{1,t} + u_{2,t} = & P_t & \forall t
\end{array} \tag{3}$$

From § 2, we recognize this as an LP and therefore know that it can be solved reliably and quickly. In this paper's modeling language the problem instantiates as:

```python
def MirmiraniOster1978(T: float = 8.0, dt: float = 0.05) -> Problem:
  p = Problem(tmin=0.0, tmax=T, desired_tstep=dt)

  rhat   = p.add_parameter("rhat", value=0.5)
  rtilde = p.add_parameter("rtilde", value=0.2)
  μ      = p.add_parameter("μ", value=0.1)
  ν      = p.add_parameter("ν", value=0.1)
  P0     = p.add_parameter("P0", value=0.05)

  u = p.add_control_var("u", dim=2, lower_bound=0)
  P = p.add_time_var("P", lower_bound=0, initial=P0)
  S = p.add_time_var("S", lower_bound=0, initial=0)

  for _, ti in p.time_indices():
    p.constrain_control_sum_at_time(u, ti, P[ti])
    p.dconstraint(P, ti, rhat   * u[ti,0] - μ * P[ti])
    p.dconstraint(S, ti, rtilde * u[ti,1] - ν * S[ti])

```

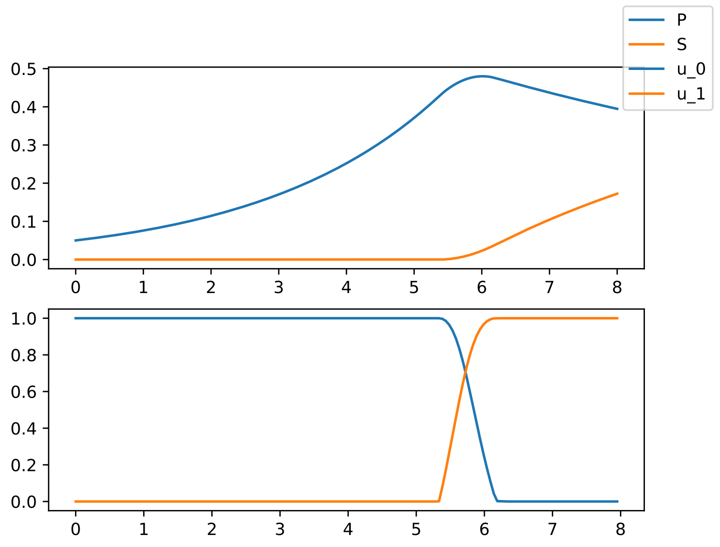


Figure 5: The result of evaluating the model described in § 3.1 with parameter values $\hat{r} = 0.5, \tilde{r} = 0.2, \mu = 0.1, \nu = 0.1, T = 8.0, P_0 = 0.05$.

```
    p.objective(Maximize(S[-1]))

    return p
```

Solving this gives the result shown in Figure 5. The growth pattern of the plant shows that initially all growth is allocated to the vegetative tissue with a smooth transition to vegetative tissue later in the growing season. The plant's vegetative tissue begins to decline in mass at that point due to the non-zero value of μ indicating tissue senescence or predation.

3.2 Recovering the control

In § 3.1 we changed the control from a multiplicative to an additive operation in order to form a convex problem. For making control plots such as those shown in Figure 5 it is useful to map the control back to the range $[0, 1]$ by taking

$$\hat{u}_{1,t} = \frac{u_{1,t}}{\sum_i u_{i,t}} = \frac{u_{1,t}}{\|u_{*,t}\|_1} \quad \forall t \tag{4}$$

3.3 When should a plant flower?

Iwasa [5] considers the question of when a plant should flower, a problem first address by Cohen (1971, 1976). Let $F(t)$ be the plant's vegetative biomass (leaves, stems and roots) and daily production (photosynthesis minus respiration) be modeled by

$$g(F) = \frac{aF}{1 + hF} \tag{5}$$

a saturating function such as this is desirable because it captures effects such as self-shading and local resource depression. Production can be allocated either to increasing vegetative biomass or reproductive biomass R according to a *control variable* $u(t)$. We therefore wish to solve the problem

$$\begin{aligned} \max_{u(t)} \quad & R(T) \\ \text{s.t.} \quad & \dot{F} = (1-u)g(F), \\ & \dot{R} = ug(F), \\ & F, R \geq 0, \\ & F(0) = F_0, \\ & R(0) = 0, \\ & u(t) \in [0,1] \end{aligned} \tag{6}$$

As usual, we will discretize and add a dimension to the control variable; however, our growth function $g(F)$ is nonlinear. We have two ways of dealing with this. Since the growth function shares a form with the Michaelis–Menten kinetics equation, we can use techniques described in § 5.3 to convert the growth function into an SOCP constraint at the cost of some computational speed. Alternatively, we could convert the function into a piecewise-linear representation using techniques described in § 5.5, at the cost of some accuracy. If we take the former route, our final program is:

$$\begin{aligned} \max_{u(t)} \quad & R(T) \\ \text{s.t.} \quad & F_{t+1} = F_t + \Delta t \cdot u_{t,1} && \forall t, \\ & R_{t+1} = R_t + \Delta t \cdot u_{t,2} && \forall t, \\ & F_t, R_t \geq 0 && \forall t, \\ & F_{t=0} = F_0, \\ & R_{t=0} = 0, \\ & u_{t,1} + u_{t,2} = g_t && \forall t, \\ & \left\| \begin{bmatrix} (a/h)F_t \\ (1/h)g_t \\ (a/h)(1/h) \end{bmatrix} \right\|_2 \leq (a/h)(1/h) + (a/h)F_t - (1/h)g_t && \forall t \end{aligned} \tag{7}$$

While the mathematics here may seem complex, the simple Python-based modeling language we've designed for this paper makes the problem easy to express:

```python
def Iwasa2000_when_flower(T: float = 8.0, dt: float = 0.05) -> Problem:
  p = Problem(tmin=0.0, tmax=T, desired_tstep=dt)

  a = p.add_parameter("a", value=0.1)
  h = p.add_parameter("h", value=1.0)
  F0 = p.add_parameter("F0", value=0.5)

  u = p.add_control_var("u", dim=2, lower_bound=0)
  F = p.add_time_var("F", lower_bound=0, initial=F0)
  R = p.add_time_var("R", lower_bound=0, initial=0)
  g = p.add_time_var("g", lower_bound=0, anchor_last=True)

  for _, ti in p.time_indices():
    p.michaelis_menten_constraint(g[ti], F[ti], β1=h, β2=1.0, β3=a)
    p.constrain_control_sum_at_time(u, g[ti], ti)
    p.dconstraint(F, ti, u[ti,0])
```

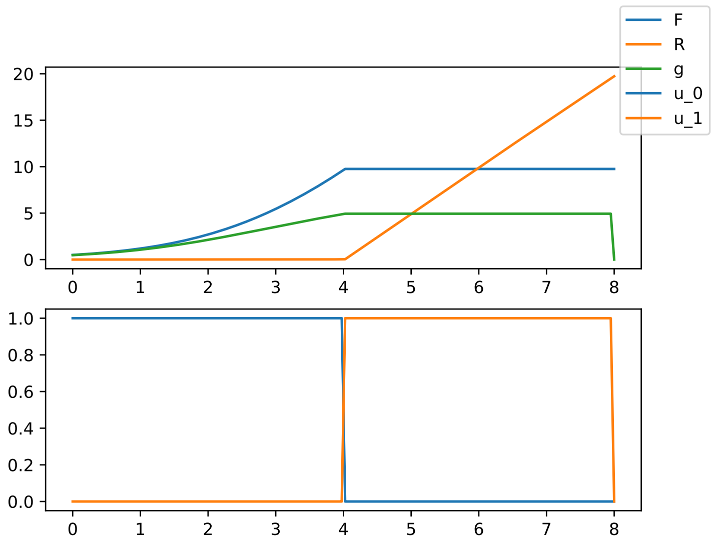


Figure 6: The result of running the model described in § 3.3 using the parameter values $a = 0.1, h = 1, T = 8, F(0) = 0.5$, following [5]. The control variables are normalized.

```
    p.dconstraint(R, ti, u[ti,1])

  optval = p.solve(Maximize(R[-1]))

  return p
```

The result of evaluating this model is shown in Figure 6. Analyzing the output, we see that the normalized control variables take values of either zero or one; this behaviour is called a "bang–bang" control. All of the plant's energy is initially used to build additional vegetative tissue (capital production); at the appropriate time all production is switched to building reproductive tissues (e.g., flowering).

3.4 Shoot–root balance

Another problem from [8]. Let X_1, X_2, and R be the shoot, root, and accumulated reproductive biomass of a plant within a single growing season. Daily photosynthesis $g(X_1, X_2)$ will increase with the shoot biomass X_1 because more leaves means more photosynthesis; it will also increase with the root biomass X_2 because more roots means more water and nutrients. But if insufficient resources are devoted to either leaves or roots, then the plant's growth as a whole will be retarded. We can formalize the problem as:

$$\begin{aligned}
\max_{u(t)} \quad & R(T) \\
\text{s.t.} \quad & \dot{X}_1 = u_1 g(X_1, X_2), \\
& \dot{X}_2 = u_2 g(X_1, X_2), \\
& \dot{R} = u_3 g(X_1, X_2), \\
& X_1, X_2, R \geq 0, \\
& u_1 + u_2 + u_3 = 1, \\
& X_1(0) = X_{1,0}, \\
& X_2(0) = X_{2,0}, \\
& R(0) = 0, \\
& u_0, u_1, u_2 \in [0, 1]
\end{aligned} \tag{8}$$

where

$$g(X_1, X_2) = \frac{1}{(a_1/LX_1^{b_1}) + (a_2/WX_2^{b_2})} \tag{9}$$

where L is the light intensity and W is the soil moisture.

If we let the shoot S be X_1 and the root T be X_2 then we can transform the problem as follows:

$$\begin{aligned}
\max_{u(t)} \quad & R(T) \\
\text{s.t.} \quad & S_{t+1} = S_t + \Delta t \cdot u_1, \\
& T_{t+1} = T_t + \Delta t \cdot u_2, \\
& R_{t+1} = R_t + \Delta t \cdot u_3, \\
& S, T, R \geq 0, \\
& u_1 + u_2 + u_3 = g(S, T), \\
& S_{t=0} = S_0, \\
& T_{t=0} = T_0, \\
& R_{t=0} = 0
\end{aligned} \tag{10}$$

This problem is straight-forward except for Equation 9, which is non-convex (Figure 7). We can see this easily by plotting a cross-section where $X_1 = 0.5$ as shown in Figure 8. To handle this situation, we sample the function $g(X_1, X_2)$ in the range $X_1 = [0, X_{1,M}]$ and $X_2 = [0, X_{2,M}]$ for some large values of $X_{1,M}$ and $X_{2,M}$ (Figure 9a) and wrap the resulting point cloud in a convex hull (Figure 9b). This returns a set of triangle simplices containing the function. Each simplex is associated with a normal vector and an offset. If we eliminate those simplices whose normal vectors have negative z components this eliminates all the simplices beneath $g(\cdot)$ (Figure 9c). The remaining simplices form an upper bound for $g(\cdot)$. The normals of the simplices can then be used to define a set of hyperplanes expressing this upper bound in a form amenable to mathematical programming.

All this results in a large linear program which presents a formidable challenge for cvxpy's compiler. Therefore, we use Julia to gain additional performance. Following [8], we choose parameter values of $a_1 = 2, a_2 = 60, b_1 = 0.5, b_2 = 2, L = 1, W = 1, X_{1,0} = 1, X_{2,0} = 2.5$ to get the following program. Running it gives us the result shown in Figure 10.

```julia
function gfunc(x1::Vector{Float64}, x2::Vector{Float64})::Vector{Float64}
  a1 = 2.0;
```

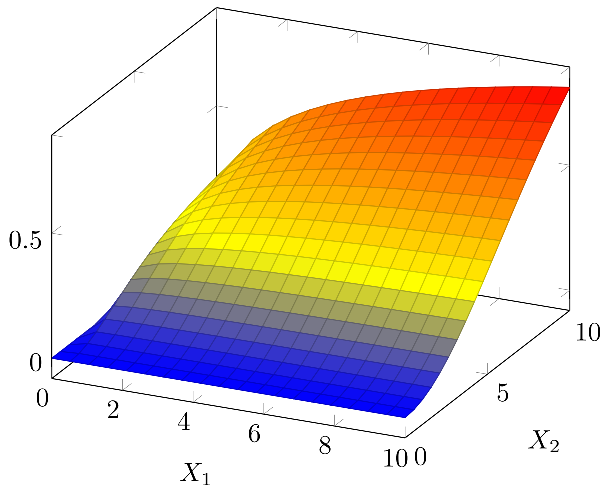


Figure 7: Surface plot of Equation 9.

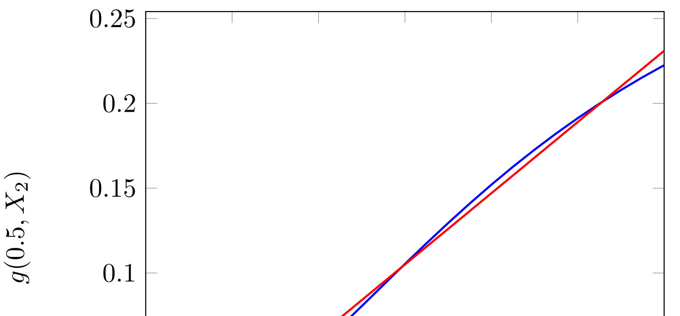
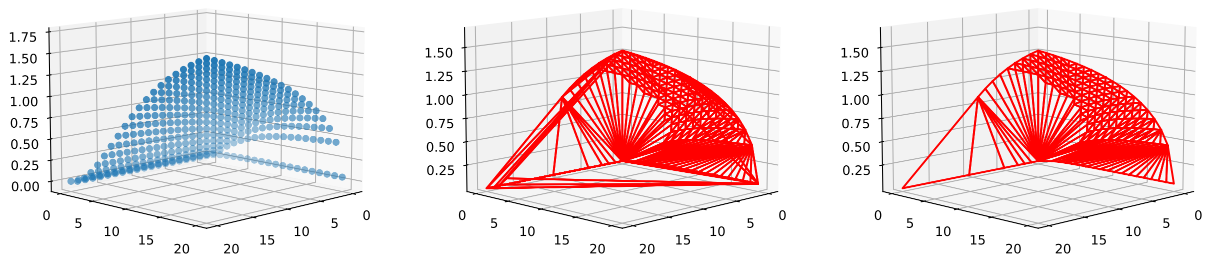


Figure 8: Cross-sectional plot of Equation 9 (blue) showing that it is non-convex (the red line passes out of the function's domain).



(a) (b) (c)

Figure 9: Using a convex hull to approximate a non-convex function. **(a)** points the function is sampled at. **(**b) the full convex hull including all simplices; note the triangles crossing the along the bottom of the graph. **(**c) the convex upper bound; note that there are no triangles crossing the bottom of the graph now.

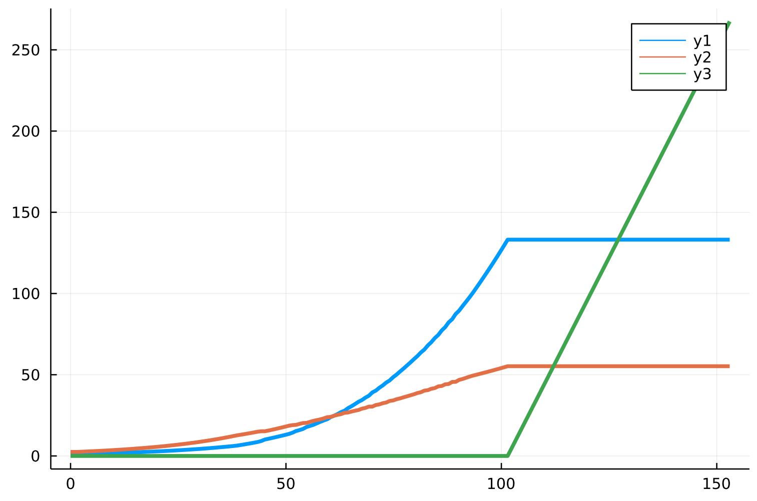


Figure 10: The optimal strategy for the model described in § 3.4.

```julia
  a2 = 60.0;
  b1 = 0.5;
  b2 = 2.0;
  L = 1.0;
  W = 1.0;
  1.0 ./ ( a1./L./(x1.^b1) .+ a2./W./(x2.^b2) );
end

# Form an N x 2 matrix of sample points
pts = vec(collect.(
  Iterators.product(LinRange(0.01,100,80), LinRange(0.01,100,80))
));
pts = reduce(hcat, pts)';
# Get values of `gfunc at these points`
zz = gfunc(pts[:,1], pts[:,2]);

# Get a convex hull to approximate `gfunc`
hull_pts = hcat(pts,zz);

# Build the model
dt=0.8;
model = Model(with_optimizer(
  ECOS.Optimizer, maxit=10000, feastol=1e-4, reltol=1e-3, abstol=1e-3
));
timeseries = LinRange(0.0, 153.0, Int(round((153.0-0.0)/dt)));

N = length(timeseries);
@variable(model, u[1:N, 1:3], lower_bound=0);
@variable(model, X1[1:N],     lower_bound=0);
@variable(model, X2[1:N],     lower_bound=0);
@variable(model, R[1:N],      lower_bound=0);
@variable(model, g[1:N],      lower_bound=0);

@constraint(model, X1[1]==1.0);
@constraint(model, X2[1]==2.5);
@constraint(model, R[1]==0.0);

scatter(hull_pts[:,1], hull_pts[:,2], hull_pts[:,3]);

hull = chull(hull_pts);
good_facets = hull.facets[hull.facets[:,3].>0,:];

for ti in 1:(N-1)
  for f in eachrow(good_facets)
    @constraint(
      model, f[1] * X1[ti] + f[2] * X2[ti] + f[3] * g[ti] + f[4] <= 0
    );
  end
  @constraint(model, sum(u[ti,:])==g[ti]);
  @constraint(model, X1[ti+1] == X1[ti] + dt * u[ti, 1]);
  @constraint(model, X2[ti+1] == X2[ti] + dt * u[ti, 2]);
```

3.5 A Single Species Living Over Multiple Seasons

Iwasa and Cohen [6] present a model of a plant which can live for n growing seasons ($0 \leq n \leq \infty$), each of length T. The plant's vegetative tissue F is the sole input to its growth function

$$g(F) = \frac{fF}{1 + hF} \tag{11}$$

where f and h are constants; this is the same function discussed in § 3.3. Growth is constrained such that

$$0 \leq \dot{F} \leq aF_n + b \tag{12}$$

The plant also has a storage tissue S which encompasses bulbs, tubers, rhizomes, and similar structures which retain energy and nutrients between growing seasons, though some fraction $1 - \gamma$ of this tissue is lost between growing seasons. At the end of growing season n a portion $0 \leq R_N \leq S_n$ of the storage tissue can retroactively be declared reproductive tissue.

A plausible model of growth in a single season is then

$$\begin{aligned} \max_{u(t)} \quad & S(T) \\ \text{s.t.} \quad & \dot{F} = u(t)(aF + b), \\ & \dot{S} = g(F) - u(t)(aF + b), \\ & F(0) = 0, \\ & S(0) = S_0, \\ & F, S \geq 0, \\ & u(t) \in [0, 1] \end{aligned} \tag{13}$$

Note that this allows biomass to move from S to F, since $g(F)$ enters the model as an addition to S and $u(t)(aF + b)$ subracts from the combined pool of S and $g(F)$. Since this models a single season maximizing $S(T)$ is the same as maximizing the reproductive tissue.

To construct the multi-season model, we link single-season models together:

$$F_n(0) = 0 \tag{14}$$
$$S_{n+1}(0) = \gamma(S_n(T) - R_n) \tag{15}$$
$$R_n(0) = 0 \tag{16}$$

The plant attempts to maximize its lifetime reproductive output:

$$R_{\text{net}} = \sum_{n=0}^{\infty} \sigma^n R_n \tag{17}$$

where $0 < \sigma < 1$ combines the annual survival rate with any future discounting. Note that if $R_1 = S_1(T)$ after optimization, then the model is indicating an annual life strategy. Similarly, if $R_2 = S_2(T)$, then the plant is biennial; otherwise, the strategy is perennial.

After discretization and application of the techniques in § 5.3 to Equation 11 we get the following mathe-

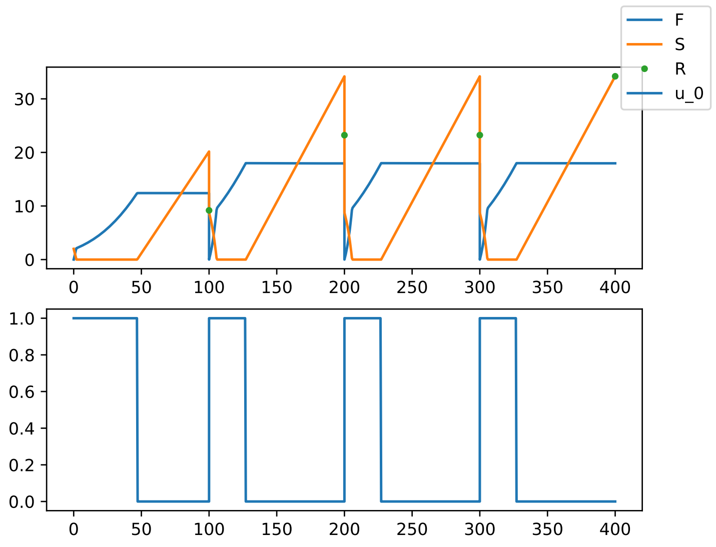


Figure 11: The result of evaluating the model described in § 3.5 with parameter values $a = 0.2, b = 1, f = 0.05, h = 0.05, T = 100, S_0 = 2, \gamma = 0.8$ over four years.

matical program:

$$
\begin{aligned}
\max_{u(t)} \quad & \sum_{n=0}^{Y} \sigma^n R_n \\
\text{s.t.} \quad & F_{n,t+1} = F_{n,t} + \Delta t \cdot u_{n,t}, \\
& S_{n,t+1} = S_{n,t} + \Delta t \left(g_{n,t} - u_{n,t} \right), \\
& \left\| \begin{bmatrix} (f/h)F_t \\ (1/h)g_{n,t} \\ (f/h)(1/h) \end{bmatrix} \right\|_2 \le (f/h)(1/h) + (f/h)F_t - (1/h)g_{n,t} \quad \forall t, \\
& F_{n,0} = 0, \\
& S_{0,0} = S_0, \\
& S_{n+1,0} = \gamma(S_{n,T} - R_n), \\
& F, S, R \ge 0, \\
& u_{n,t} \ge 0, \\
& u_{n,t} \le aF_{n,t} + b
\end{aligned}
\tag{18}
$$

Though we would like $Y = \infty$, our mathematical modeling framework cannot handle this. However, this is not a severe restriction. Consider a relatively large value for σ such as 0.85. After 10 years the discount rate is 0.19 and, after 20 years, it has become 0.038. That is, the plant's reproductive value 20 years out is worth only 3.8% of the value of the plant's reproductive output during its first year of life. σ is necessarily greater than zero, otherwise the plant has no incentive to grow; σ is also necessarily less than one because the future is risky and no plant can guarantee its survival.

In this paper's modeling language the problem instantiates as:

```
def Iwasa1989_multi_season(
  T: float = 100.0, years: int = 8, dt: float=0.5
) -> Problem:
  p = Problem(tmin=0.0, tmax=T, desired_tstep=dt, years=years, seasonize=True)

  a  = p.add_parameter("a", value = 0.2)
  b  = p.add_parameter("b", value = 1)
  f  = p.add_parameter("f", value = 0.05)
  h  = p.add_parameter("h", value = 0.05)
  S0 = p.add_parameter("S0", value = 2)
  y  = p.add_parameter("y", value = 0.8)

  u = p.add_control_var("u", dim=1, lower_bound=0)
  F = p.add_time_var("F", lower_bound=0, initial=0)
  S = p.add_time_var("S", lower_bound=0, initial=S0)
  g = p.add_time_var("g", lower_bound=0, anchor_last=True)
  R = p.add_year_var("R", lower_bound=0)

  for n, ti in p.time_indices():
    p.constrain_control_sum_at_time(u, a*F[n,ti]+b, n=n, t=ti)
    p.dconstraint(F, (n,ti), u[n,ti,0])
    p.dconstraint(S, (n,ti), g[n,ti] - u[n,ti,0])
    p.michaelis_menten_constraint(g[n,ti], F[n,ti], β1=f, β2=1.0, β3=h)

  for n in p.year_indices():
    p.constraint(F[n, 0] == 0)
    p.constraint(R[n] <= S[n,-1])
    if n < p.years-1:
      p.constraint(S[n+1, 0] == y * (S[n,-1] - R[n]))

  p.objective(Maximize(p.time_discount("R", 0.85)))

  return p
```

Solving this gives the result shown in Figure 11. The growth pattern of the plant shows that during its first year of life it preserves about half of its storage to jump-start growth in subsequent years. During the next few years, the plant enters a recurring pattern which is only broken during the last year when the finite time horizon cause it to put all of its energy into reproduction. The plant's control variable is again bang-bang.

3.6 Environmental Effects

Mironchenko and Kozłowski [10] present a flexible growth model coupled with more realistic depictions of seasonality and time-varying effects. This flexibility gives them the ability to determine under what conditions a variety of life histories are optimal. The model is

$$\begin{aligned}\dot{x}_1 &= v_1(t)g(x_3) - \mu(t)x_1\\ \dot{x}_2 &= (v(t) - v_1(t))\,g(x_3)\\ \dot{x}_3 &= \zeta(t)f(x_1) - v(t)g(x_3) - \omega(t)x_3\end{aligned} \tag{19}$$

where x_1 is vegetative tissue, x_2 is reproductive tissue, x_3 is storage tissue, $v \in [0,1]$ and $v_1 \in [0,v]$ are control variables allocating storage tissue at a maximal rate of $g(x_3)$, $f(x_1)$ is the rate of photosynthesis in optimal environment conditions, $\zeta(t)$ models the dependence of the photosynthetic rate on the environment ($\zeta(t) = 0$ implies no photosynthesis is possible), $\mu(t)$ is the loss rate of vegetative tissues, and $\omega(t)$ is the loss rate of storage tissues.

The optimal rate of photosynthesis, $f(x)$, always has the form

$$f(x) = \frac{ax}{bx+k} \tag{20}$$

while the achievable fraction of photosynthesis, $\zeta(t)$, remains the same throughout the systems Mironchenko and Kozłowski test:

$$\zeta(t) = 0.2 + 0.8 \left| \sin\left(\frac{\pi}{12} t\right) \right| \tag{21}$$

its periodic form might represent, e.g., seasonal temperature variation. μ and ω often have periodic forms as well.

To convert this model to our form, we note that the control variables v and v_1 apportion up to $g(x_3)$ energy between x_1 and x_2. Therefore, we can replace them with two control variables $u_1, u_2 \geq 0$ constrained such that $u_1 + u_2 \leq g(x_3)$. Doing this and discretizing per usual gives us the system:

$$\begin{aligned}
\max_{u(t)} \quad & x_{2,T} \\
\text{s.t.} \quad & x_{1,t+1} = x_{1,t} + \Delta t \cdot (u_{1,t} - \mu(t) x_1), \\
& x_{2,t+1} = x_{2,t} + \Delta t \cdot u_{2,t}, \\
& x_{3,t+1} = x_{3,t} + \Delta t \cdot (\zeta(t) f(x_{1,t}) - u_{1,t} - u_{2,t} - \omega(t) x_{3,t}), \\
& x_{1,0} = 0.05m, \\
& x_{2,0} = 0, \\
& x_{3,0} = 0.95m, \\
& u_{1,t} + u_{2,t} \leq g(x_3)
\end{aligned} \tag{22}$$

where m is the initial seed mass. The introduction of the time-varying functions ζ, ω, μ don't make our modeling job more difficult. Since we've discretized over time, we can just evaluate these functions at each time point and insert the result as a constant into our model.

3.6.1 Inducing Multiple Reproduction Periods

What is the effect of periodic vegetative tissue loss on an annual, as might be caused by herbivores grazing repeatedly over the same area as it regrows? To answer this, Mironchenko and Kozłowski instantiate the above model for two environments

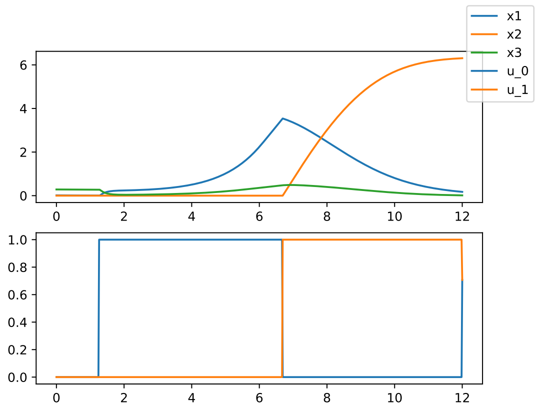
Variable	**Stable Environment**	**Unstable Environment**
$\omega(t) =$	0.05	$t/(1+0.5t)$
$\mu(t) =$	$0.8\lvert\cos(\frac{\pi}{12}t)\rvert$	$1.8\lvert\cos(\pi t)\rvert$
$g(x_3) =$	$5x_3$	$16x_3$
$f(x_1) =$	$1.5x/(1+0.03x_1)$	$4x/(1+0.03x_1)$
Outcome	Single reproductive period	Multiple reproductive periods

Note that while the stable environment has a $\mu(t)$ that's periodic over the course of a year, as would be the case for seasonal vegetative loses, the unstable environment has a $\mu(t)$ that varies monthly, perhaps as levels of herbivory fluctuate. To survive this stability the plant in the unstable environment needs a higher maximum storage conversion rate $g(x_3)$ and greater optimal photosynthetic efficiency $f(x_1)$. The unstable plant's $\omega(t)$ was chosen by Mironchenko and Kozłowski as a saturating function to help emphasize the control variables and costates.

In this paper's modeling language the above becomes:

```
def MironchenkoFigure2(is_stable: bool) -> Problem:
  """Values and equations are drawn from Mironchenko's source code TODO"""
  def zeta(t: float) -> float:
    return 0.2 + 0.8 * abs(math.sin(math.pi / 12 * t))

  def mu(t: float) -> float:
    if is_stable:
      return 0.8 * abs(math.cos(math.pi / 12 * t))
    else:
      return 1.8 * abs(math.cos(math.pi * t))

  def omega(t: float) -> float:
    return 0.05 if is_stable else t / (1 + 0.5 * t)

  def g(x: Variable):
    return 5 * x if is_stable else 16 * x

  seed_mass = 0.3

  p = Problem(tmin=0.0, tmax=12, desired_tstep=0.02)

  u  = p.add_control_var("u", dim=2, lower_bound=0)
  x1 = p.add_time_var("x1", lower_bound=0, initial=0.05*seed_mass)
  x2 = p.add_time_var("x2", lower_bound=0, initial=0.00*seed_mass)
  x3 = p.add_time_var("x3", lower_bound=0, initial=0.95*seed_mass)
  f  = p.add_time_var("f",  lower_bound=0, anchor_last=True)

  for _, ti in p.time_indices():
    t = p.idx2time(ti)
    p.michaelis_menten_constraint(
      f[ti],  x1[ti], β1=1.5 if is_stable else 4, β2=1.0, β3=0.3
    )
    p.constrain_control_sum_at_time(u, g(x3[ti]), ti)
    p.dconstraint(x1, ti, u[ti,0] - mu(t) * x1[ti])
    p.dconstraint(x2, ti, u[ti,1])
    p.dconstraint(
      x3, ti, zeta(t) * f[ti] - u[ti,0] - u[ti,1] - omega(t) * x3[ti]
    )

  status, optval = p.solve(
    Maximize(x2[-1]), solver="ECOS", verbose=True, max_iters=10000,
    abstol=1e-3, reltol=1e-3, feastol=1e-4
  )

  print(status)

  return p
```

Running the above program gives us the result shown in Figure 12. Note that the low levels of vegetative tissue lost in the stable environment result in continuous reproductive output, even though the loss levels are fluctuating. Even though the unstable environment is more favourable for the plant, having greater photosynthesis and faster storage conversion, strong fluctuations in vegetative tissue loss result in pulsed reproductive output.


(a)

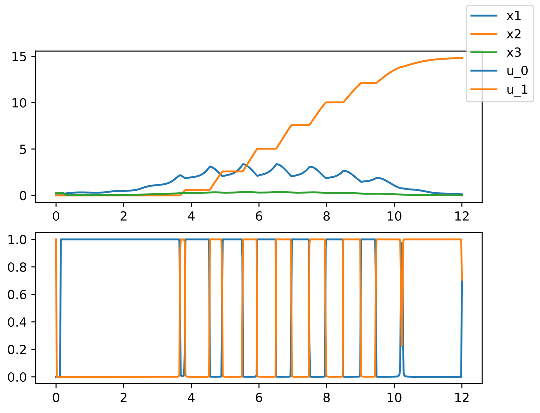

(b)

Figure 12: The result of evaluating the model described in § 3.6.1. **(a)** shows an annual plant growing in a stable environment whereas **(b)** shows an annual plant growing in an unstable environment. The results match Figure 2 of [10].

3.6.2 Perennial Strategies

We can generate a variety of perennial strategies by varying the environment:

Variable	**Constantly Harsh**	**Constant Mortality**	**Pleasant**	**Very Pleasant**
Seed Mass	0.3	Same	Same	Same
Seed Fractions	$< 5\%, 0\%, 95\% >$	Same	Same	Same
$\omega(t) =$	0.15	0, see below	1	0.1
$\mu(t) =$	$0.4\lvert\cos(\frac{\pi}{12}t)\rvert$	$0.4\lvert\cos(\pi t)\rvert$	$0.1\lvert\cos(\pi t)\rvert$	$0.4\lvert\cos(\pi t)\rvert$
$g(x_3) =$	$2.5x_3$	Same	Same	Same
$f(x_1) =$	$0.5/(1 + 0.1x_1)$	Same	Same	$0.5/(1 + 0.01x_1)$
Outcome	Storage	No storage	Storage	Monocarpy

We can instantiate these environments in this paper's modeling language through minor modifications to the program shown in § 3.6.1.

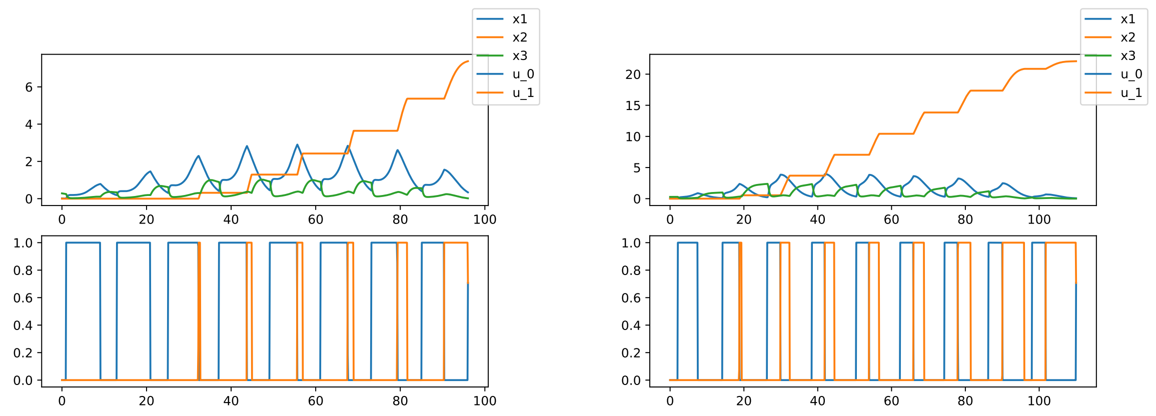
As shown in Figure 13, the "Constantly Harsh" environment causes the plant to lose significant vegetative tissue over winters. It makes up for this by moving energy into storage. The "Constant Mortality" produces a similar result via a different means. For this, we set $\omega(t) = 0$, but model the plant has having a 3% chance of dying at any given moment. This implies that the probability $L(t)$ of surviving to an age t is given by $L(t) = e^{-0.03t}$. The plant's goal is then to maximize its expected lifetime fitness, which is given by

$$\int_0^T L(t)\dot{x}_2(t)dt \tag{23}$$

incorporating this into our modeling language is straight-forward

```
discount = np.exp(-0.03 * p.timeseries)
status, optval = p.solve(Maximize(x2 @ discount), solver="ECOS", verbose=True)
```

Contrast this with the the "Pleasant" and "Very Pleasant" environments, as shown in Figure 14. The decreased over-winter tissue loss (μ) in the pleasant environment means that even though storage losses (ω) are smaller, the plant's best strategy is to accept some tissue losses rather than waste energy on using storage. In the "Very Pleasant" environment, the plant is again subject to higher over-winter tissue loss (μ),



(a) (b)

Figure 13: The result of evaluating the “Constantly Harsh” **(a)** and “Constant Mortality” **(b)** environments described in § 3.6.2. Both environments have similar optimal strategies. The results match Figure 5a of [10] exactly and Figure 5e nearly (see the note on reproducibility TODO).

but storage costs are very low. The biggest difference, though is that the plant has minimal self-shading. This is modeled by choosing a maximum photosynthetic rate (f) with a denominator $(1 + 0.01x_1)$ that is small for most values of x_1 the plant will encounter.

4 Inverse Problems

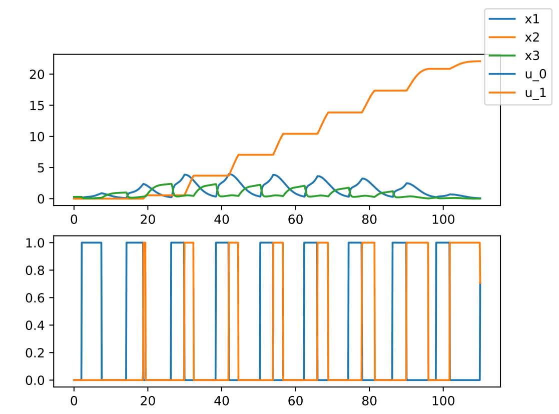
Given a noisy sampling of a timeseries can we recover the underlying system? To explore this, let’s start with a system where we know the system. We’ll use the simple single-season system described in § 3.1.

To make a noisy problem, we solve the single-season system, add normally distributed noise to one of the result variables, and then sample a few data points to get a vector of scalar values $P_{\text{noise,sampled}}$. Previously, our objective was to maximize the amount of seeds at the end of the season $\max S(T)$. Our new objective tries to match the system, as described by the vector of variables P, to the data. This gives us the objective

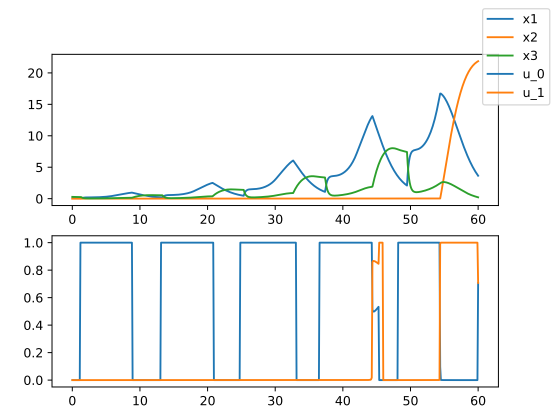
$$\min_u \|P_{\text{noise,sampled}} - P\|_2 \tag{24}$$

we can implement this as follows:

```python
# Solve the original system
problem = MirmiraniOster1978()
problem.solve()

# Get a copy of original result for plotting
P_original, S_original = problem["P"].value.copy(), problem["S"].value.copy()

# Take a subset of the points and add noise to them
P_with_noise = P_original + np.random.normal(
  scale=0.05, size=P_original.shape
)
sample = np.random.choice(
  list(range(P_original.shape[0])), replace=False, size=10
```



(a)



(b)

Figure 14: The result of evaluating the “Pleasant” **(a)** and “Very Pleasant” **(b)** environments in § 3.6.2. The plant requires uses almost no storage in the pleasant environment while in the very pleasant environment it chooses monocarpy as a strategy. The results match Figure 4 and Figure 5c of [10].

```
)

```

running this gives us the result shown in Figure 15a.

The optimal solution of the system for this new least-squares objective is overfit: the system is able to follow the noise and sparsity of the data. However, we have a strong prior that the system has a bang-bang control. Bürger et al. [2] provides us a method for recovering the optimal bang-bang control strategy. First, we solve the problem with no constraints on the control. This gives us continuous-valued outputs for the controls. Second, we solve a *combinatorial integral approximation* (CIA) problem. The inputs for this problem are the discretized timepoints which we used to approximation the differential equations and the values of the controls, mapped to the range $[0, 1]$, resulting from the relaxed problem. The CIA problem allows us to specify constraints such as the number of times a control should switch from on to off. Since we have a prior expectation from § 3.1 that this should happen only once per growing season, we add that as a constraint. The result of solving the CIA problem is a set of bang-bang control values. Third, and finally, we solve our least-squares problem again, but this time with the constraint that the control must follow the bang-bang values we obtained from the CIA.

These steps require some subtly since a key part step in the conversion from differential to convex forms so far has been manipulating the control from a multiplicative to an additive factor. § 3.2 explains how to recover $[0, 1]$ controls to get the input to the CIA. We then note that the control constraint for § 3.1 is

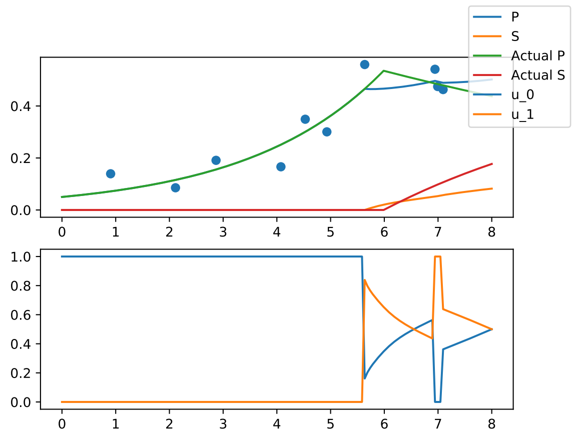
$$u_{1,t} + u_{2,t} = P_t \quad \forall t \tag{25}$$

If we call the bang-bang controls returned from the CIA $\hat{u}$, then the following additional constraints ensure that our system follows strategy prescribed by the CIA

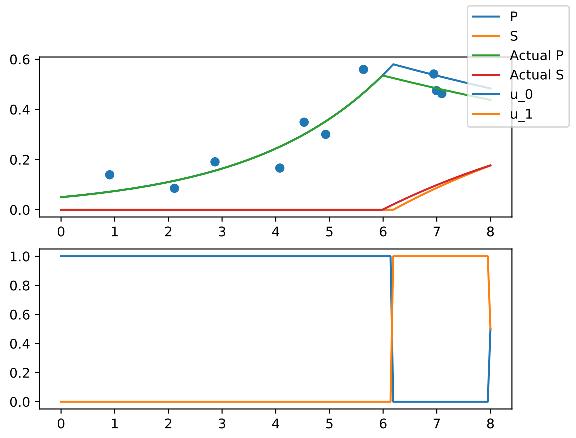
$$\begin{aligned} u_{1,t} &= \hat{u}_{1,t} P_t \quad \forall t \\ u_{2,t} &= \hat{u}_{2,t} P_t \quad \forall t \end{aligned} \tag{26}$$

We can implement all this as follows:

```
fig.savefig("imgs/Mirmirani_inverse_bad_fit.pdf", bbox_inches='tight')

```



(a)



(b)

Figure 15: The result of solving the inverse problem described in § 4. **(a)** shows the result when the inverse problem is solved without any constraints on the control variable. **(b)** shows the result when the inverse problem is solved with the control variable constrained to switch only two times.

```


# Solve problem with noise and no constraints on the control
# This is a "relaxed" version of the problem we want to solve
problem.objective(Minimize(diff_orig_noise))
problem.solve(solver="ECOS")

# Use Combinatorial Integer Approximation to find the best control strategy
# given the relaxed solution
ba = pycombina.BinApprox(
  problem.timeseries,
  problem.get_normalized_control_value("u")[:-1,:]
)
ba.set_n_max_switches([1,1])

milp = pycombina.CombinaMILP(ba)
milp.solve()

# Now, resolve the problem, but constraining the control based on the CIA
problem = MirmiraniOster1978(constrain=True)
```

As shown in Figure 15, using the CIA least-squares approach allows us to recover almost exactly the original system.

5 Techniques

5.1 Bilinear Constraints

A bilinear constraint is one in which we require that $w = xy$ where both x and y are variables. A common way to handle such a constraint is by enclosing it in concave and convex bounds known as McCormick envelopes, as shown in Figure 16. Unfortunately, these bounds are often loose. This can be handled either by iteratively

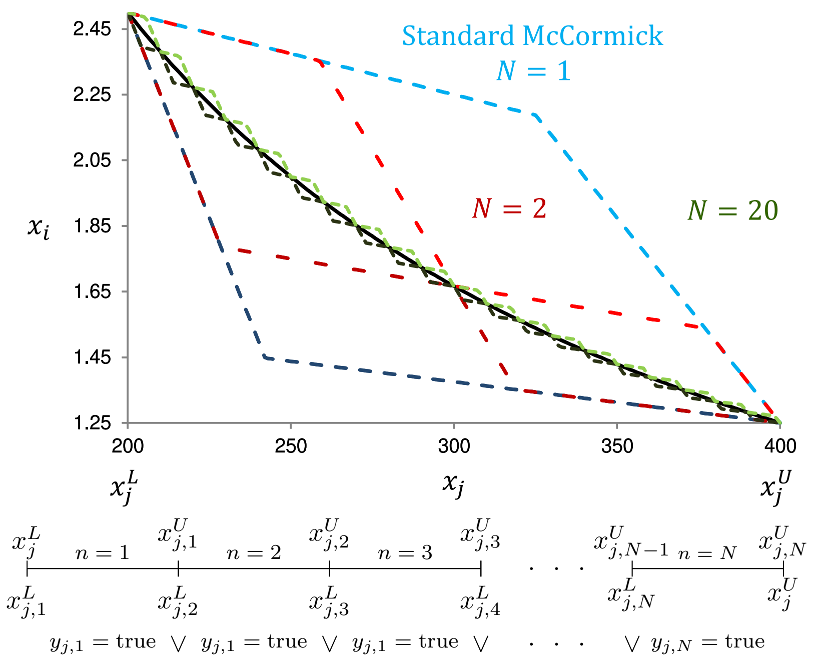


Figure 16: Piecewise McCormick Envelopes

refining the bounds or by using mixed-integer programming to generate a series of small envelopes. This can be expressed via a *generalized disjunctive program (GDP)*:

$$x_{jn}^L = x_j^L + (x_j^U - x_j^L)\frac{n-1}{N} \tag{27}$$

$$x_{jn}^U = x_j^L + (x_j^U - x_j^L)\frac{n}{N} \tag{28}$$

$$x_i^L \le x_i \le x_i^U \tag{29}$$

$$\bigvee_n \begin{bmatrix} y_{jn} \\ w_{ij} \ge x_i \cdot x_{jn}^L + x_i^L \cdot x_j - x_i^L \cdot x_{jn}^L \\ w_{ij} \ge x_i \cdot x_{jn}^U + x_i^U \cdot x_j - x_i^U \cdot x_{jn}^U \\ w_{ij} \le x_i \cdot x_{jn}^L + x_i^U \cdot x_j - x_i^U \cdot x_{jn}^L \\ w_{ij} \le x_i \cdot x_{jn}^U + x_i^L \cdot x_j - x_i^L \cdot x_{jn}^U \\ x_{jn}^L \le x_j \le x_{jn}^U \end{bmatrix} \tag{30}$$

$$y_{jn} \in \{\text{True}, \text{False}\} \tag{31}$$

$$\Omega(y) = \text{True} \tag{32}$$

While some optimization DSLs can handle GDPs natively, we can re-express this is a MILP so that we solve it using a wide variety of solvers and DSLs [4].

$$x_{jn}^L = x_j^L + (x_j^U - x_j^L)\frac{n-1}{N} \tag{33}$$

$$x_{jn}^U = x_j^L + (x_j^U - x_j^L)\frac{n}{N} \tag{34}$$

$$x_i^L \leq x_i \leq x_i^U \tag{35}$$

$$-w_{ij} + x_i \cdot x_{jn}^L + x_i^L \cdot x_j - x_i^L \cdot x_{jn}^L \leq M(1 - y_{jn}) \tag{36}$$

$$-w_{ij} + x_i \cdot x_{jn}^U + x_i^U \cdot x_j - x_i^U \cdot x_{jn}^U \leq M(1 - y_{jn}) \tag{37}$$

$$w_{ij} - x_i \cdot x_{jn}^L - x_i^U \cdot x_j + x_i^U \cdot x_{jn}^L \leq M(1 - y_{jn}) \tag{38}$$

$$w_{ij} - x_i \cdot x_{jn}^U - x_i^L \cdot x_j + x_i^L \cdot x_{jn}^U \leq M(1 - y_{jn}) \tag{39}$$

$$x_{jn}^L - x_j \leq M(1 - y_{jn}) \tag{40}$$

$$x_j - x_{jn}^U \leq M(1 - y_{jn}) \tag{41}$$

$$\sum_j y_{jn} = 1 \tag{42}$$

$$y_{jn} \in \{0, 1\} \tag{43}$$

5.2 Hyperbolic constraints

5.2.1 As SOCPs

For a vector $\vec{w}$ a constraint of the form

$$\vec{w}^T\vec{w} = \|\vec{w}\|_2^2 \leq xy, \quad x \geq 0, \quad y \geq 0 \tag{44}$$

can be transformed into the SOCP constraint

$$\left\| \begin{bmatrix} 2\vec{w} \\ x - y \end{bmatrix} \right\|_2 \leq x + y \tag{45}$$

Note that the optimal value of a scalar w produced this way is the squart root of xy.

5.3 Michaelis–Menten Kinetics

The Michaelis–Menten relates the rate y of an enzymatic reaction to the concentration x of a substrate, as follows

$$Y = \frac{\beta_1 X}{\beta_2 + X} \tag{46}$$

where β_1 and β_2 are positive parameters.

Equation 46 most often appears in our context as a saturating limit to growth, e.g., growth $\leq \frac{\beta_1 X}{\beta_2 + X}$. Since optimizing reproductive fitness often means maximizing growth, we anticipate that the inequality will be binding. Equation 46 is therefore equivalent to the hyperbolic constraint

$$\hat{X}^2 + \hat{Y}^2 \leq (\hat{X} - \hat{Y})(2\beta_1\beta_2 + \hat{X} - \hat{Y}) \tag{47}$$

$$0 \leq \hat{X} - \hat{Y} \tag{48}$$

were $\hat{X} = \beta_1 X$ and $\hat{Y} = \beta_2 Y$; this can be proven by simplifying Equation 47. Converting this to the standard SOC form using Equation 45 gives

$$\left\| \begin{bmatrix} \hat{X} \\ \hat{Y} \\ \beta_1\beta_2 \end{bmatrix} \right\|_2 \leq \beta_1\beta_2 + \hat{X} - \hat{Y} \tag{49}$$

and substituting in $\hat{X}$ and $\hat{Y}$ gives

$$\left\| \begin{bmatrix} \beta_1 X \\ \beta_2 Y \\ \beta_1 \beta_2 \end{bmatrix} \right\|_2 \leq \beta_1\beta_2 + \beta_1 X - \beta_2 Y \tag{50}$$

Note that if our M-M kinetics equation is instead

$$Y = \frac{\beta_1 X}{\beta_2 + \beta_3 X} \tag{51}$$

we can convert to the canonical form by taking

$$Y = \frac{(\beta_1/\beta_3)X}{(\beta_2/\beta_3) + X} \tag{52}$$

An application of this technique to gradostat design is discussed in [12].

5.4 SOS2 Constraints

An SOS2 constraint is defined by specifying a set of variables $\{t_1, t_2, \ldots, t_n\}$ and is equivalent to the following three constraints:

- $t_1, t_2, \ldots, t_n \geq 0$
- $t_1 + t_2 + \ldots + t_n = 1$
- At most two adjacent variables, t_i and potentially t_{i+1} can be non-zero

SOS2 constraints are useful for modeling piecewise-linear functions, as described in § 5.5. If a solver doesn't have SOS2 constraints but has binary variables, these can be used to construct an equivalent, though less efficient, formulation.

5.5 Linearizing Functions

One way to handle a non-linear function $f(x)$ is to approximate it as a piecewise-linear function of n line segments using a standard λ-form transformation [13]. For $f(x)$ in the range $[a, b]$ and a step size $s = (b-a)/n$, we add the following constraints to the model:

$$x = \sum_{i=0}^{n-1} (a + is)\lambda_i \tag{53}$$

$$y = \sum_{i=0}^{n-1} f(a + is)\lambda_i \tag{54}$$

$$1 = \sum_{i=0}^{n-1} \lambda_i \tag{55}$$

$$\text{at most two adjacent } \lambda_i \text{ can be nonzero} \tag{56}$$

now, y is a linear approximation of $f(x)$ and can be used in its place anywhere in the model. The constraint given by Equation 56 is nonlinear. However, if $f(x)$ has diminishing returns and we are maximizing our objective then this constraint is guaranteed to be true and need not be explicitly included in the model [13]. (Once the model has been solved, the values of λ can be checked as a verification measure.) For cases where the above doesn't hold, most solvers include *SOS2* constraints as a way to enforce Equation 56 (see § 5.4).

References

[1] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 8(1):67, Apr 2007. ISSN 1573-2924. doi: 10.1007/s11081-007-9001-7. URL https://doi.org/10.1007/s11081-007-9001-7.

[2] A. Bürger, C. Zeile, M. Hahn, A. Altmann-Dieses, S. Sager, and M. Diehl. pycombina: An open-source tool for solving combinatorial approximation problems arising in mixed-integer optimal control. *IFAC-PapersOnLine*, 53(2):6502–6508, 2020. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2020.12.1799. URL https://www.sciencedirect.com/science/article/pii/S2405896320324083. 21st IFAC World Congress.

[3] V. Chandrasekaran and P. Shah. Relative entropy optimization and its applications. *Mathematical Programming*, 161(1):1–32, Jan 2017. ISSN 1436-4646. doi: 10.1007/s10107-016-0998-2. URL https://doi.org/10.1007/s10107-016-0998-2.

[4] I. E. Grossmann and J. P. Ruiz. Generalized disjunctive programming: A framework for formulation and alternative algorithms for MINLP optimization. In J. Lee and S. Leyffer, editors, *Mixed Integer Nonlinear Programming*, pages 93–115, New York, NY, 2012. Springer New York. ISBN 978-1-4614-1927-3.

[5] Y. Iwasa. Dynamic optimization of plant growth. *Evolutionary Ecology Research*, 2(4):437–455, 2000. URL http://www.cabdirect.org/abstracts/20000711553.html.

[6] Y. Iwasa and D. Cohen. Optimal growth schedule of a perennial plant. *American Naturalist*, pages 480–505, 1989. URL http://www.jstor.org/stable/2462084.

[7] Y. Iwasa and D. Cohen. Optimal growth schedule of a perennial plant. *American Naturalist*, pages 480–505, 1989. URL http://www.jstor.org/stable/2462084.

[8] Y. Iwasa and J. Roughgarden. Shoot/root balance of plants: optimal growth of a system with many vegetative organs. *Theoretical Population Biology*, 25(1):78–105, 1984. doi: 10.1016/0040-5809(84)90007-8. URL http://www.sciencedirect.com/science/article/pii/0040580984900078.

[9] M. Mirmirani and G. Oster. Competition, kin selection, and evolutionary stable strategies. *Theoretical Population Biology*, 13(3):304–339, 1978. doi: 10.1016/0040-5809(78)90049-7. URL http://dx.doi.org/10.1016/0040-5809(78)90049-7.

[10] A. Mironchenko and J. Kozłowski. Optimal allocation patterns and optimal seed mass of a perennial plant. *Journal of theoretical biology*, 354:12–24, 2014. doi: 10.1016/j.jtbi.2014.03.023.

[11] A. Pugliese and J. Kozłowski. Optimal patterns of growth and reproduction for perennial plants with persisting or not persisting vegetative parts. *Evolutionary Ecology*, 4(1):75–89, 1990. doi: 10.1007/BF02270717. URL http://link.springer.com/article/10.1007/BF02270717.

[12] J. Taylor and A. Rapaport. Second-order cone optimization of the gradostat, 2021.

[13] H. P. Williams. *Model building in mathematical programming.* John Wiley & Sons, 2013.