Symplectic integrators and optimal control

Cross, Mathew I.

2005

MIMS EPrint: **2006.34**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

ISSN 1749-9097

SYMPLECTIC INTEGRATORS AND OPTIMAL CONTROL

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2005

Mathew I. Cross
School of Mathematics

Contents

Words: 16008

List of Tables

List of Figures

Abstract

When selecting a method to integrate an ODE system, it is intuitively clear that preservation of geometric properties is desirable. The particular subclasses of ODE systems we will consider are *Lagrangian* and *Hamiltonian systems*. The dynamical equations for these derive from variational principles, and we obtain structure preserving integrators by discretizing the principles rather than the ODEs they generate. We demonstrate some advantages that these *symplectic integrators* have over methods that are more rudimentary by looking at some examples from optimal control theory.

Our major motivation for considering symplectic integrators is solving an *image registration* problem, where, using the least effort, we associate a set of landmark points on one image to a corresponding set of points on another. A mathematical formulation of this problem is as a Hamiltonian system; this becomes apparent once we realize that we are computing the motion of particles (the initial landmark points) under some appropriate potential function. We investigate the performance of symplectic methods on this, more complex, problem. We show that by formulating the problem as a system of nonlinear equations rather than one of optimal control, the explicit Euler method performs better than the symplectic integrators, especially on a set of data points generated by a real experiment. We give some evidence that the higher-order methods in MATLAB's ODE suite may be better still, but we do not pursue this line of investigation in any detail.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Chapter 1

Introduction

Image registration involves transforming, or *warping*, a given image so that it becomes similar to a reference image. The mapping that relates points in the two images is called the *warp*. The manner in which the warp is done depends on the application; areas where the procedure is commonly encountered include medicine and computer vision.

In medicine we may want to study the development of a tumour in a patient. Suppose we have a series of scans taken at different times using different equipment, and possibly where the patient's orientation varies. Owing to these variations (caused by the generation of the images), we need to transform the scans so that we can concentrate only on the important medical phenomena. It is likely that we would have to resort to *nonrigid registration*—nonlinear warping rather than just rotations or translations—to correct the local differences in the images. Then, tissue at a certain position in the most recent image can be related to tissue at the same position in an earlier, reference, image.

Other examples mean that real-time registration is sometimes necessary. For example, changes in the shape of a brain during neurosurgery need to be corrected during the surgery. In such applications a delicate balance must be made between accuracy and efficiency of the model.

The type of registration of interest to us is *control point* registration; this is easy to describe and results in a physically meaningful warp [15]. It is based on identifying a

finite set of features common to the two images, and then requiring that corresponding features are mapped to each other under the warp. As a trivial example, suppose we start with the upper image[1] in Figure 1.1, and that we wish to rearrange its pixels and associated colour intensities to obtain the lower. By selecting suitable control

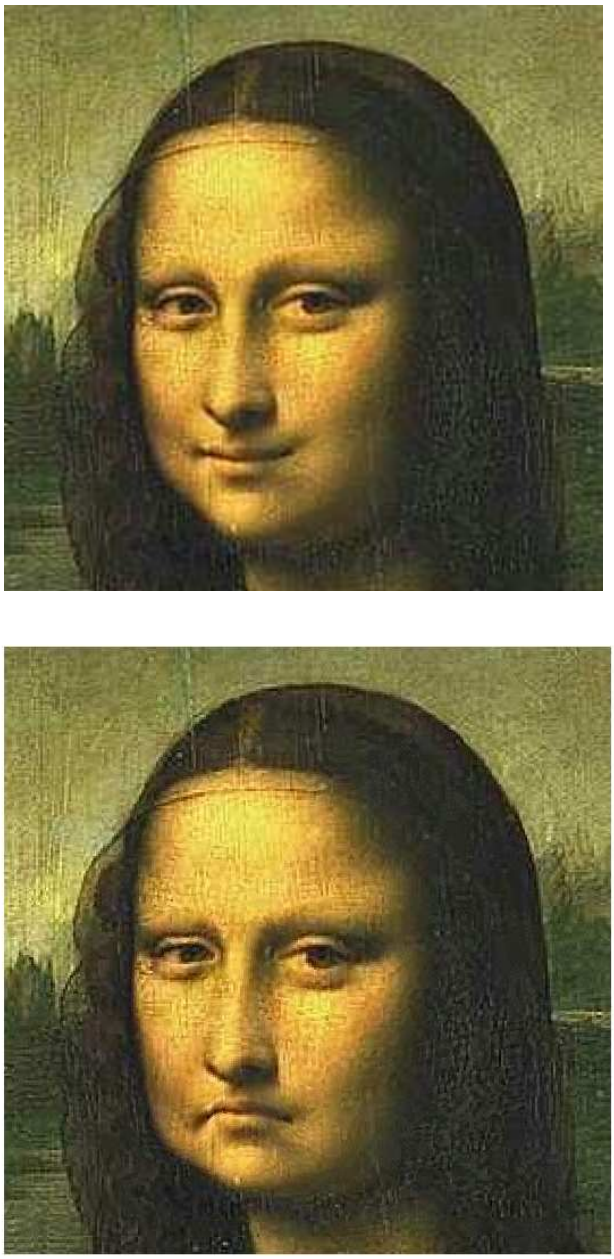


Figure 1.1: Top: image to be warped. Bottom: reference (target) image

points around the eyes and corners of the mouth of the subject, we would like to turn happiness into sadness while preserving all other areas of the image.

[1]Source: `http://commons.wikimedia.org/wiki/Image:Mona_Lisa.jpg`.

One of the more successful mathematical formulations of control point registration is as an optimal control problem. This involves the minimization of an energy associated to the warp subject to the differential equations that describe the movement of the control points. Our interest lies with this aspect of the problem, so we refer the reader who wishes to learn more about applications to Modersitzki's *Numerical Methods for Image Registration* [15], and the references therein.

Let us motivate our approach to modelling the above system by considering the motion of a simple pendulum.

Example 1.0.1. Assume that we have a pendulum whose bob has mass $m = 1$ and whose rod is massless of length $\ell = 1$. Take the acceleration due to gravity to be $g = 1$. Denote by $q(t)$ the angular displacement of the rod from the vertical, and by $p(t)$ the pendulum's momentum (Figure 1.2). The total energy of the system, the

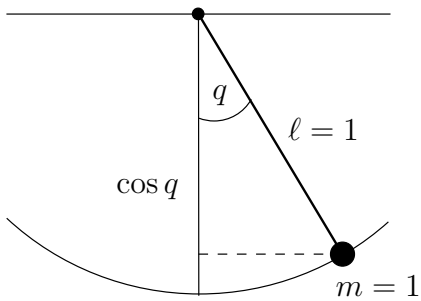


Figure 1.2: Simple pendulum: configuration

sum of its kinetic and potential energies, is then

$$H(q,p) = T(p) + U(q) = \frac{1}{2}p^2 - \cos q. \tag{1.0.1}$$

Our pendulum is an example of a *Hamiltonian system*, and H is called the *Hamiltonian*. The dynamics are governed by the equations

$$\begin{aligned} \begin{pmatrix} \dot{q} \\ \dot{p} \end{pmatrix} &= \begin{pmatrix} \partial H/\partial p \\ -\partial H/\partial q \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \nabla H(q,p), \end{aligned} \tag{1.0.2}$$

with the dot $\dot{}$ denoting differentiation with respect to t. In our case, the equations of motion become

$$(\dot{q}, \dot{p}) = (p, -\sin q). \tag{1.0.3}$$

Denote by Q the space in which q lies. Owing to the periodicity in q of (1.0.3) it is natural to take Q to be the unit circle $\mathbb{S}^1$, so that (q, p) lies on the cylinder $\mathbb{S}^1 \times \mathbb{R}$. However, we will take $(q, p) \in \mathbb{R}^2$, simply because (q, p)-space is then easier to plot. Defining $\boldsymbol{z}(t) = (q(t), p(t))^{\mathrm{T}}$ and $\boldsymbol{f}(\boldsymbol{z}) = (p, -\sin q)^{\mathrm{T}}$, the system (1.0.3) has the form

$$\dot{\boldsymbol{z}} = \boldsymbol{f}(\boldsymbol{z}). \tag{1.0.4}$$

Let us experiment by using some simple numerical methods to integrate this system. We select a constant step-size h and write $\boldsymbol{z}_k \stackrel{\text{def}}{=} \boldsymbol{z}(kh)$ for $k \geq 0$. We use the following methods.

$$\begin{aligned}
\boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + h\boldsymbol{f}(\boldsymbol{z}_k) && \text{(explicit Euler)},\\
\boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + h\boldsymbol{f}(\boldsymbol{z}_{k+1}) && \text{(implicit Euler)},\\
\boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + h\boldsymbol{f}(q_k, p_{k+1}) && \text{(symplectic Euler)},\\
\boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + h\boldsymbol{f}\big((\boldsymbol{z}_{k+1} + \boldsymbol{z}_k)/2\big) && \text{(implicit midpoint rule)}.
\end{aligned}$$

(Note that the symplectic Euler method treats q by the explicit and p by the implicit Euler method.)

The observation that H is constant along the solution curves of (1.0.2) allows us to describe the exact trajectories of the system. Indeed,

$$\frac{\mathrm{d}}{\mathrm{d}t} H(q(t), p(t)) = \dot{q}\frac{\partial H}{\partial q} + \dot{p}\frac{\partial H}{\partial p} = 0,$$

if q and p satisfy (1.0.2), and we say that the total energy is a *conserved quantity* of the system. From this observation the exact trajectories are the level curves of $p^2/2 - \cos q$. Figure 1.3 plots, in $\mathbb{R}^2$, the exact trajectories and the numerical solutions of the system (1.0.3). For the explicit and implicit Euler methods we take $h = 0.2$, and $\boldsymbol{z}_0 = (0.5, 0)$ and $(1.5, 0)$ respectively; for the other two methods we take $h = 0.3$, and $\boldsymbol{z}_0 = (0, 0.7)$, $(0, 1.4)$ and $(0, 2.1)$. The trajectory for the explicit (respectively implicit) Euler method spirals out from (respectively in to) the origin. The other two

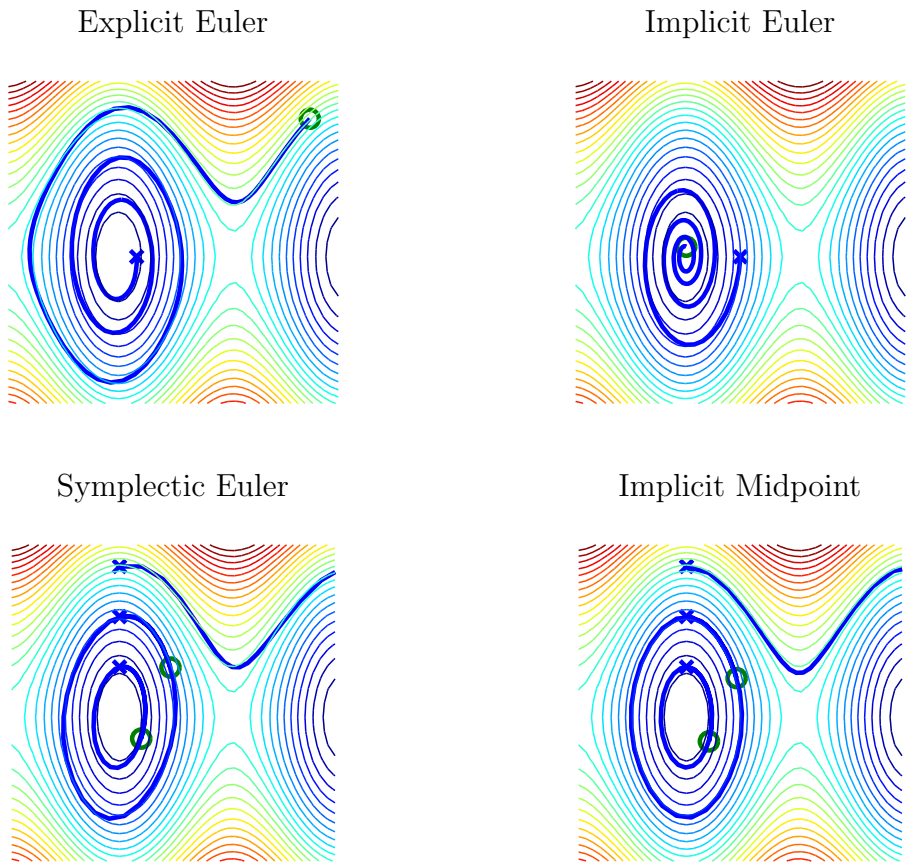


Figure 1.3: Simple pendulum: trajectories

methods show the correct qualitative behaviour, with the trajectory for the implicit midpoint rule agreeing with the exact solution to a greater degree than that of the symplectic Euler method.

Recall that the *flow* of the system (1.0.4) is the mapping φ_t that sends a $\boldsymbol{z}_0 \in \mathbb{R}^2$ to the point obtained by 'flowing' for time t along the system's solution curve. That is, $\varphi_t(\boldsymbol{z}_0) = \boldsymbol{z}(t)$, whenever $\boldsymbol{z}(0) = \boldsymbol{z}_0$. The exact flow of a Hamiltonian system with one degree of freedom is *area-preserving*, in the sense that

$$\det \frac{\partial \varphi_t}{\partial(q_0, p_0)} = 1 \qquad \text{for all } t.$$

This formula is easily verified by hand. We shall see in Theorem 2.2.3 that, for a Hamiltonian system with greater than one degree of freedom, an appropriate higher-dimensional notion of area is preserved by the exact flow.

Now, the *numerical flow* of a numerical method is the mapping $\Phi_h : \boldsymbol{z}_k \mapsto \boldsymbol{z}_{k+1}$. For our pendulum example we see that the explicit Euler method is *not* area-preserving; viz.,

$$\det \frac{\partial \Phi_{\mathrm{eE},h}}{\partial(q_0, p_0)} = \begin{vmatrix} 1 & h \\ -h\cos q_0 & 1 \end{vmatrix} = 1 + h^2 \cos q_0.$$

A similar calculation can be carried out for the implicit Euler method, where the determinant is

$$\det \frac{\partial \Phi_{\mathrm{iE},h}}{\partial(q_0, p_0)} = (1 + h^2 \cos q_1)^{-1}.$$

However, the symplectic Euler method *is* area-preserving:

$$\begin{pmatrix} 1 & -h \\ 0 & 1 \end{pmatrix} \frac{\partial \Phi_{\mathrm{sE},h}}{\partial(q_0, p_0)} = \begin{pmatrix} 1 & 0 \\ -h\cos q_0 & 1 \end{pmatrix},$$

thus $\det(\partial \Phi_{\mathrm{sE},h}/\partial(q_0, p_0)) = 1$. We shall look at these ideas again in a more general setting (2.2.7), and we will see in Section 2.4 that the implicit midpoint rule has similar geometric properties.

To summarize: the pendulum example shows that, besides the explicit and implicit Euler methods not being good choices for this problem, the results from the symplectic Euler method and implicit midpoint rule agree well with the exact flow

of the system, with the midpoint rule agreeing more closely. As we will see in Section 2.5, the midpoint rule has a greater order of accuracy than the other three methods. Furthermore, the symplectic Euler method and implicit midpoint rule are area-preserving, just as the exact flow is. Such observations provide incentive for the study of *geometric integrators*: numerical methods that preserve geometric properties of the exact flow of a differential equation.

As an overview of this thesis, in Chapter 2 we introduce some ideas from Lagrangian and Hamiltonian mechanics, and from their discrete counterparts. This leads to a discussion of variational and symplectic integrators, which are of primary interest to us. Chapter 3 presents some examples from the field of optimal control, and serves as preparation for an attack on the main registration problem. This problem is introduced in more detail in Chapter 4, where we show that it corresponds to the Hamiltonian system (4.1.13), whose MATLAB implementation is then described. In Chapter 5 we see how simple geometric integrators applied to this system compare with the standard discretization of the dynamics, which is by the explicit Euler method. Appendix A describes an experiment in the same vein as those in Chapter 3, while Appendix B outlines why some aspects of the Hamiltonian formulation of the image registration problem make sense. All code used to produce the examples in this thesis can be found in Appendices C–E.

The numerical experiments in this work use MATLAB version 7.0.4.352 (R14) Service Pack 2, running on a Linux workstation with 2.4 GHz AMD processor and 2 GB of RAM.

Chapter 2

Mechanics and Integrators

We introduce two viewpoints for describing mechanical systems: *Lagrangian mechanics* in Section 2.1 and *Hamiltonian mechanics* in Section 2.2. Continuous and discrete formulations are covered, and in Section 2.3 we describe how the theory is modified in the presence of external forcing. Section 2.4 discusses some properties of numerical methods derived from these systems. Then, in the context of Example 1.0.1, Section 2.5 describes how we can find *modified equations* that are better satisfied by these methods.

2.1 Lagrangian Mechanics

Lagrangian mechanics is a reformulation of classical mechanics. In it, the trajectory of an object is derived by finding the path that minimizes the *action*, which is the integral over time of the *Lagrangian* of the system.

We begin with a space Q (sometimes called the *configuration space*) with coordinates $(q_1, \ldots, q_n)$ describing the position of the system, and we construct the *state space* TQ with (abstract) coordinates $(q_1, \ldots, q_n, \dot{q}_1, \ldots, \dot{q}_n)$. It is usual to write $\boldsymbol{q} = (q_1, \ldots, q_n)^{\mathrm{T}}$, and we call n the number of *degrees of freedom* of the system. The *Lagrangian* is a map $L : TQ \to \mathbb{R}$, and in classical mechanics it usually appears as kinetic minus potential energy

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = T(\boldsymbol{q}, \dot{\boldsymbol{q}}) - U(\boldsymbol{q}),$$

with $\dot{\boldsymbol{q}} = \mathrm{d}\boldsymbol{q}/\mathrm{d}t$. However, the Lagrangian need not always come from a concrete mechanical system.

The coordinates $q_i(t)$ satisfy the *Euler–Lagrange* equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0 \qquad i = 1, \ldots, n. \tag{2.1.1}$$

These equations arise from the extremization of the *action functional*

$$A(\boldsymbol{q}) = \int_0^T L(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t))\,\mathrm{d}t \tag{2.1.2}$$

over all curves $\boldsymbol{q}(t)$ such that $\boldsymbol{q}(0) = \boldsymbol{q}^0$ and $\boldsymbol{q}(T) = \boldsymbol{q}^1$, for given points $\boldsymbol{q}^0, \boldsymbol{q}^1 \in Q$. The principle that this action is extremized by the motion is called *Hamilton's principle.*

Example 2.1.1. Consider n particles moving in $\mathbb{R}^2$. Choose $Q = \mathbb{R}^{2n}$, and take L to be the energy difference

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\dot{\boldsymbol{q}}^{\mathrm{T}} M \dot{\boldsymbol{q}} - V(\boldsymbol{q}),$$

where M is a symmetric positive-definite matrix of masses and V is some potential-function. Then the Euler–Lagrange equations become

$$M\ddot{\boldsymbol{q}} = -\nabla V(\boldsymbol{q}), \tag{2.1.3}$$

which is Newton's second law for the motion of particles in a potential field V.

A naïve approach to deriving a numerical method for integrating a Lagrangian system is to discretize the Euler–Lagrange equations (2.1.1). However, if we discretize the variational formulation from the outset, then the resulting method preserves important geometric properties of the mechanical system, such as those in Chapter 1. We will explain more about this in Section 2.4. A second advantage is that in certain situations the variational method generates a smaller mathematical system than does a direct discretization of the dynamics: thus, solution of this system is more efficient.

We now construct the discrete formulation of Lagrangian mechanics. Again take a configuration space Q, but now let the state space be the *discrete state space* $Q \times Q$.

Choose a time-step $h \in \mathbb{R}$, and replace a path $\boldsymbol{q} : [0, T] \to Q$ by a discrete path $\boldsymbol{q}_d : \{0, h, \ldots, Nh = T\} \to Q$, with $\boldsymbol{q}_d(kh)$ being thought of as an approximation to $\boldsymbol{q}(kh)$. Write $\boldsymbol{q}_d(kh) = \boldsymbol{q}_k$ (the bold type-face serving to distinguish this quantity from the kth coordinate q_k). In this way, we are regarding the nearby points $\boldsymbol{q}_{k+1}$ and $\boldsymbol{q}_k$ as being a discrete analogue of the velocity vector $\dot{\boldsymbol{q}}(kh)$. A function $L_d : Q \times Q \to \mathbb{R}$ is called a *discrete Lagrangian*. We need not take L_d to depend on h, since we will be considering only constant step-sizes. The *discrete action functional* is

$$A_d(\boldsymbol{q}_d) = \sum_{k=0}^{N-1} L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}),$$

so we can interpret $L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1})$ as an approximation to the (continuous) action functional along the curve segment between $\boldsymbol{q}_k$ and $\boldsymbol{q}_{k+1}$. Extremizing A_d over discrete paths whose boundary points $\boldsymbol{q}_0$ and $\boldsymbol{q}_N$ are held fixed gives the *discrete Euler–Lagrange equations*

$$D_2 L_d(\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) + D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = 0 \qquad k = 1, \ldots, N-1, \tag{2.1.4}$$

with D_i denoting differentiation with respect to the ith component. Given initial conditions $(\boldsymbol{q}_0, \boldsymbol{q}_1)$, by solving the (perhaps nonlinear) system (2.1.4) for $\boldsymbol{q}_{k+1}$ we obtain an update map called the *discrete Lagrangian map*

$$\Phi_{L_d} : (\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) \mapsto (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}). \tag{2.1.5}$$

Example 2.1.2. Returning to the n-particle Lagrangian, suppose we discretize the action A by the rectangle rule; write $\boldsymbol{v}_{k+1/2} = (\boldsymbol{q}_{k+1} - \boldsymbol{q}_k)/h$. Then

$$L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = hL(\boldsymbol{q}_k, \boldsymbol{v}_{k+1/2}).$$

The derivatives of L_d are

$$\begin{aligned} D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) &= h\frac{\partial L}{\partial \boldsymbol{q}}(\boldsymbol{q}_k, \boldsymbol{v}_{k+1/2}) - \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}_k, \boldsymbol{v}_{k+1/2}) \\ &= -h\nabla V(\boldsymbol{q}_k) - M\boldsymbol{v}_{k+1/2}, \\ D_2 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) &= \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}_k, \boldsymbol{v}_{k+1/2}) \\ &= M\boldsymbol{v}_{k+1/2}. \end{aligned}$$

The discrete Euler–Lagrange equations thus read

$$M\left(\frac{\boldsymbol{q}_{k+1} - 2\boldsymbol{q}_k + \boldsymbol{q}_{k-1}}{h^2}\right) = -\nabla V(\boldsymbol{q}_k), \tag{2.1.6}$$

a centred-difference discretization of Newton's equations (2.1.3). From (2.1.6) comes the (explicit) discrete Lagrangian map

$$(\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) \mapsto (\boldsymbol{q}_k, 2\boldsymbol{q}_k - \boldsymbol{q}_{k-1} - h^2 M^{-1}\nabla V(\boldsymbol{q}_k)). \tag{2.1.7}$$

2.2 Hamiltonian Mechanics

Hamiltonian mechanics is a second reformulation of classical mechanics. It originally arose from Lagrangian mechanics, but each field can be studied independently, and sometimes it is not possible to translate between the two. The dictionaries for doing this, if they exist, are called the *Legendre transforms* (2.2.2) and (2.2.3), and sometimes it is easier to approach a problem from one of the two perspectives than the other. Hamiltonian mechanics is related more closely than Lagrangian mechanics to the physical idea of energy, as we commented in Chapter 1.

Given a configuration space Q, we construct the *phase space* T^*Q with coordinates $(q_1, \ldots, q_n, p_1, \ldots, p_n)$. The coordinate p_i can be interpreted as the momentum of the ith particle. The *Hamiltonian* is a map $H : T^*Q \to \mathbb{R}$, and we have already seen its classical manifestation as total energy: $H(\boldsymbol{q}, \boldsymbol{p}) = T(\boldsymbol{q}, \boldsymbol{p}) + U(\boldsymbol{q})$. Again, the Hamiltonian need not come from a concrete mechanical system.

The Hamiltonian analogue of the Euler–Lagrange equations are *Hamilton's equations*, which we met in (1.0.2). These equations are

$$\dot{\boldsymbol{q}} = \frac{\partial H}{\partial \boldsymbol{p}}(\boldsymbol{q}, \boldsymbol{p}), \quad \dot{\boldsymbol{p}} = -\frac{\partial H}{\partial \boldsymbol{q}}(\boldsymbol{q}, \boldsymbol{p}).$$

Writing $\boldsymbol{z} = (\boldsymbol{q}, \boldsymbol{p})^{\mathrm{T}}$ we get

$$\dot{\boldsymbol{z}} = \begin{pmatrix} 0 & I_n \\ -I_n & 0 \end{pmatrix} \nabla H(\boldsymbol{z}), \tag{2.2.1}$$

with I_n being the $n \times n$ identity matrix. The flow $\varphi_H : T^*Q \times \mathbb{R} \to T^*Q$ of the system (2.2.1) is called the *Hamiltonian flow*. Hamilton's equations can be derived from a

variational principle on $(\boldsymbol{q}, \boldsymbol{p})$-space in a similar way to how the Euler–Lagrange equations are derived on $(\boldsymbol{q}, \dot{\boldsymbol{q}})$-space, and the two systems are often equivalent [11]; see (2.2.5).

In order to relate the two formulations of mechanics, we use the *Legendre transform*. Given a Lagrangian L, this is the map $\mathbb{F}L : TQ \to T^*Q$ that satisfies

$$\mathbb{F}L : (\boldsymbol{q}, \dot{\boldsymbol{q}}) \mapsto (\boldsymbol{q}, \boldsymbol{p}) = \left(\boldsymbol{q}, \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}, \dot{\boldsymbol{q}})\right). \tag{2.2.2}$$

Analogously, given a Hamiltonian H, the *inverse Legendre transform* is the map

$$\mathbb{F}H : (\boldsymbol{q}, \boldsymbol{p}) \mapsto (\boldsymbol{q}, \dot{\boldsymbol{q}}) = \left(\boldsymbol{q}, \frac{\partial H}{\partial \boldsymbol{p}}(\boldsymbol{q}, \boldsymbol{p})\right). \tag{2.2.3}$$

We will assume that $\mathbb{F}L$ is a global isomorphism, so that the inverse Legendre transform is precisely the inverse of the Legendre transform.

Using the Legendre transform it is possible to express a Hamiltonian $H(\boldsymbol{q}, \boldsymbol{p})$ in terms of a Lagrangian $L(\boldsymbol{q}, \dot{\boldsymbol{q}})$:

$$H(\boldsymbol{q}, \boldsymbol{p}) = \boldsymbol{p} \cdot \dot{\boldsymbol{q}} - L(\boldsymbol{q}, \dot{\boldsymbol{q}}), \tag{2.2.4}$$

with $\dot{\boldsymbol{q}}$ written as a function of $\boldsymbol{q}$ and $\boldsymbol{p}$, employing (2.2.2). If $\mathbb{F}L$ is a global isomorphism, then the Euler–Lagrange equations are equivalent to Hamilton's equations, as can easily be verified. Indeed, applying the chain rule to (2.2.4) and then using the Legendre transform (2.2.2),

$$\frac{\partial H}{\partial \boldsymbol{q}} = \boldsymbol{p} \cdot \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{q}} - \frac{\partial L}{\partial \boldsymbol{q}} - \frac{\partial L}{\partial \dot{\boldsymbol{q}}} \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{q}} = -\frac{\partial L}{\partial \boldsymbol{q}}, \tag{2.2.5a}$$

and

$$\frac{\partial H}{\partial \boldsymbol{p}} = \dot{\boldsymbol{q}} + \boldsymbol{p} \cdot \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{q}} - \frac{\partial L}{\partial \dot{\boldsymbol{q}}} \frac{\partial \dot{\boldsymbol{q}}}{\partial \boldsymbol{q}} = \dot{\boldsymbol{q}}. \tag{2.2.5b}$$

Hence

$$\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial L}{\partial \dot{\boldsymbol{q}}} - \frac{\partial L}{\partial \boldsymbol{q}} = 0 \iff \dot{\boldsymbol{q}} = \frac{\partial H}{\partial \boldsymbol{p}}, \quad \dot{\boldsymbol{p}} = -\frac{\partial H}{\partial \boldsymbol{q}} \quad \text{and} \quad \boldsymbol{p} = \frac{\partial L}{\partial \dot{\boldsymbol{q}}}.$$

Example 2.2.1. Returning to the n particles of Example 2.1.1, the total energy is

$$E(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2} \dot{\boldsymbol{q}}^{\mathrm{T}} M \dot{\boldsymbol{q}} + V(\boldsymbol{q}).$$

Relating momentum and velocity via $\boldsymbol{p} = M\dot{\boldsymbol{q}}$ we obtain the total energy as a function of $\boldsymbol{q}$ and $\boldsymbol{p}$:

$$H(\boldsymbol{q}, \boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^{\mathrm{T}} M^{-1}\boldsymbol{p} + V(\boldsymbol{q}),$$

and Hamilton's equations (2.2.1) are

$$\dot{\boldsymbol{q}} = M^{-1}\boldsymbol{p}, \quad \dot{\boldsymbol{p}} = -\nabla V(\boldsymbol{q}).$$

This is Newton's second law once more (cf. (2.1.3)). The Euler–Lagrange equations for this dynamical system encode the same information as Hamilton's equations, as long as we translate using $\boldsymbol{p} \leftrightarrow M\dot{\boldsymbol{q}}$.

We now describe the generalized notion of area-preservation alluded to in Chapter 1. Let V be a $2n$-dimensional vector space with basis $\{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_n\}$. Denote the dual basis for V^* by $\{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n\}$, and define a bilinear map $\omega : V \times V \to \mathbb{R}$ by

$$\omega(\boldsymbol{v}_1, \boldsymbol{v}_2) = \sum_{i=1}^{n} \alpha_i(\boldsymbol{v}_1)\beta_i(\boldsymbol{v}_2) - \beta_i(\boldsymbol{v}_1)\alpha_i(\boldsymbol{v}_2). \tag{2.2.6}$$

This bilinear map acts by

$$\omega(\boldsymbol{A}_i, \boldsymbol{B}_j) = -\omega(\boldsymbol{B}_j, \boldsymbol{A}_i) = \delta_{ij},$$
$$\omega(\boldsymbol{A}_i, \boldsymbol{A}_j) = \omega(\boldsymbol{B}_i, \boldsymbol{B}_j) = 0,$$

and thus its action is uniquely determined by the antidiagonal matrix

$$J = \begin{pmatrix} 0 & I_n \\ -I_n & 0 \end{pmatrix}.$$

Call a linear map $\boldsymbol{F} : V \times V \to V \times V$ *symplectic*[1] if

$$\omega(\boldsymbol{F}(\boldsymbol{\xi}), \boldsymbol{F}(\boldsymbol{\eta})) = \omega(\boldsymbol{\xi}, \boldsymbol{\eta}) \qquad \text{for all } \boldsymbol{\xi}, \boldsymbol{\eta} \in V \times V. \tag{2.2.7}$$

If $n = 1$, then $\omega(\boldsymbol{\xi}, \boldsymbol{\eta})$ simply computes the area of the parallelogram spanned by $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ [6]. Thus, in this case, a symplectic map is a map that preserves areas.

[1] The Greek adjective *symplektos* means 'twisted'.

An similar definition can be made for differentiable *non*linear maps between (even-dimensional) Euclidean spaces, since differentiable functions are locally linear. Let $U \subset \mathbb{R}^{2n}$. Call a differentiable nonlinear map $\boldsymbol{F} : U \to \mathbb{R}^{2n}$ *symplectic* if

$$\omega\left(\frac{\partial \boldsymbol{F}}{\partial(\boldsymbol{q},\boldsymbol{p})}\boldsymbol{\xi}, \frac{\partial \boldsymbol{F}}{\partial(\boldsymbol{q},\boldsymbol{p})}\boldsymbol{\eta}\right) = \omega(\boldsymbol{\xi},\boldsymbol{\eta}) \qquad \text{for all } \boldsymbol{\xi},\boldsymbol{\eta} \in \mathbb{R}^{2n}. \tag{2.2.8}$$

Equivalently, $\boldsymbol{F}$ is symplectic if

$$\left(\frac{\partial \boldsymbol{F}}{\partial(\boldsymbol{q},\boldsymbol{p})}\right)^{\mathrm{T}} J \frac{\partial \boldsymbol{F}}{\partial(\boldsymbol{q},\boldsymbol{p})} = J. \tag{2.2.9}$$

As with the linear case, (2.2.8) has an interpretation in terms of area-preservation [6].

The important thing that we will now get out of this is: symplecticity is a characteristic property of the flow of a Hamiltonian system. In other words, locally at least, an ODE system is Hamiltonian if and only if its flow is symplectic. A system $\dot{\boldsymbol{z}} = \boldsymbol{f}(\boldsymbol{z})$, with $\boldsymbol{f} : U \to \mathbb{R}^{2n}$, is said to be *locally Hamiltonian* if every $\boldsymbol{z}_0 \in U$ has a neighbourhood on which $\dot{\boldsymbol{z}} = J\,\nabla H$ for some H that is C^2. To show this, we need a result that is a corollary of the Poincaré Lemma of differential topology, proven here in the language of this thesis.

Lemma 2.2.2. *Suppose $N \subset \mathbb{R}^n$ is open, and let $\boldsymbol{g} : N \to \mathbb{R}^n$ be a C^1 function whose Jacobian $\partial\boldsymbol{g}/\partial\boldsymbol{x}$ is symmetric for all $\boldsymbol{x} \in N$. Then every $\boldsymbol{x}_0 \in N$ has a neighbourhood on which there exists a function $H(\boldsymbol{x})$ satisfying $\boldsymbol{g}(\boldsymbol{x}) = \nabla H(\boldsymbol{x})$.*

Proof. Without loss of generality, take $\boldsymbol{x}_0 = 0$. Select a ball $B(\boldsymbol{x}_0) \subset N$ and define

$$H(\boldsymbol{x}) = \int_0^1 \boldsymbol{x}^{\mathrm{T}}\boldsymbol{g}(t\boldsymbol{x})\,\mathrm{d}t + c, \qquad \boldsymbol{x} \in B(\boldsymbol{x}_0).$$

We can then verify that H is of the required form:

$$\begin{aligned}
\frac{\partial H}{\partial x_k}(\boldsymbol{x}) &= \int_0^1 g_k(t\boldsymbol{x}) + \boldsymbol{x}^{\mathrm{T}}\frac{\partial \boldsymbol{g}}{\partial x_k}(t\boldsymbol{x})t\,\mathrm{d}t \\
&= \int_0^1 g_k(t\boldsymbol{x}) + \boldsymbol{x}^{\mathrm{T}}\nabla g_k(t\boldsymbol{x})t\,\mathrm{d}t \qquad \text{by symmetry of the Jacobian,} \\
&= \int_0^1 \frac{\mathrm{d}}{\mathrm{d}t}\big(t g_k(t\boldsymbol{x})\big)\,\mathrm{d}t = g_k(\boldsymbol{x}).
\end{aligned}$$

□

Theorem 2.2.3. *Given a C^1 function $\boldsymbol{f} : U \to \mathbb{R}^{2n}$, the ODE system $\dot{\boldsymbol{z}} = \boldsymbol{f}(\boldsymbol{z})$ is locally Hamiltonian if and only if its flow φ_t is symplectic on U for all sufficiently small t.*

Proof. For necessity, first notice that $\partial\varphi_t/\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)$ satisfies

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)} &= \frac{\partial}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\Big(J\,\nabla H(\varphi_t(\boldsymbol{z}_0))\Big)\\ &= J\,\nabla^2 H(\varphi_t(\boldsymbol{z}_0))\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)},\end{aligned} \tag{2.2.10}$$

with ∇^2 denoting the Hessian matrix. Then we find that

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\left(\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)^{\mathrm{T}} J\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)\right) &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)^{\mathrm{T}} J\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)\\ &\quad+\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)^{\mathrm{T}} J\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)\\ &= 0,\end{aligned}$$

upon substituting (2.2.10) and using that $J^{\mathrm{T}}J = I_{2n}$ and $J^2 = -I_{2n}$. But φ_0 is the identity map, so the relation

$$\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right)^{\mathrm{T}} J\left(\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)}\right) = J \tag{2.2.11}$$

is true for $t = 0$, and then we are done, as long as the flow remains inside U.

Sufficiency is comparably straightforward, provided we use Lemma 2.2.2. The Jacobian in this case satisfies

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)} = \frac{\partial\boldsymbol{f}}{\partial(\boldsymbol{q},\boldsymbol{p})}(\varphi_t(\boldsymbol{z}_0))\frac{\partial\varphi_t}{\partial(\boldsymbol{q}_0,\boldsymbol{p}_0)},$$

since we are no longer assuming (but are trying to prove) that $\boldsymbol{f} = J\,\nabla H$. Differentiating (2.2.11) at $t = 0$ gives

$$\left(\frac{\partial\boldsymbol{f}}{\partial(\boldsymbol{q},\boldsymbol{p})}(\boldsymbol{z}_0)\right)^{\mathrm{T}} J + J\frac{\partial\boldsymbol{f}}{\partial(\boldsymbol{q},\boldsymbol{p})}(\boldsymbol{z}_0) = 0.$$

Since $J^{\mathrm{T}} = -J$, this just says that $J^{\mathrm{T}}(\partial\boldsymbol{f}/\partial(\boldsymbol{q},\boldsymbol{p}))(\boldsymbol{z}_0)$ is symmetric for all $\boldsymbol{z}_0$. Lemma 2.2.2 then provides us with a function $H(\boldsymbol{z})$ whose gradient is $J^{\mathrm{T}}\boldsymbol{f}(\boldsymbol{z})$. Since $J^{-\mathrm{T}} = J$, the proof is complete. □

An important characterization of symplectic maps lies in the existence of *generating functions* for them.

Theorem 2.2.4. *Fix a Hamiltonian system, a time interval* $[t_0, t_1]$*, and write* $\boldsymbol{z}(t_0) = (\boldsymbol{q},\boldsymbol{p})$ *and* $\boldsymbol{z}(t_1) = (\boldsymbol{s},\boldsymbol{r})$*. A map* $\varphi : (\boldsymbol{q},\boldsymbol{p}) \mapsto (\boldsymbol{s},\boldsymbol{r})$ *is symplectic if and only if there exists locally a function* $S(\boldsymbol{q},\boldsymbol{p})$ *satisfying*

$$\boldsymbol{r}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{s} - \boldsymbol{p}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{q} = \mathrm{d}S.$$

Proof. Ultimately, this is an application of Lemma 2.2.2. Substituting the Jacobian of φ into (2.2.11) shows that φ is symplectic if and only if

$$\boldsymbol{s}_{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{r}_{\boldsymbol{q}} = \boldsymbol{r}_{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{q}}, \quad \boldsymbol{r}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{q}} - I = \boldsymbol{s}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{r}_{\boldsymbol{q}} \quad \text{and} \quad \boldsymbol{s}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{r}_{\boldsymbol{p}} = \boldsymbol{r}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{p}}, \tag{2.2.12}$$

with subscripts denoting the relevant partial derivatives. Since $\mathrm{d}\boldsymbol{s} = \boldsymbol{s}_{\boldsymbol{q}}\,\mathrm{d}\boldsymbol{q} + \boldsymbol{s}_{\boldsymbol{p}}\,\mathrm{d}\boldsymbol{p}$,

$$\boldsymbol{r}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{s} - \boldsymbol{p}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{q} = (\boldsymbol{r}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{q}} - \boldsymbol{p}^{\mathrm{T}}, \boldsymbol{r}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{p}}) \begin{pmatrix} \mathrm{d}\boldsymbol{q} \\ \mathrm{d}\boldsymbol{p} \end{pmatrix}.$$

The Jacobian of the coefficient vector from the right-hand side is

$$\begin{pmatrix} \boldsymbol{r}_{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{q}} & \boldsymbol{r}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{q}} - I \\ \boldsymbol{r}_{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{p}} & \boldsymbol{r}_{\boldsymbol{p}}^{\mathrm{T}}\boldsymbol{s}_{\boldsymbol{p}} \end{pmatrix} + \sum_i r_i \nabla^2 s_i,$$

which is symmetric if and only if φ is symplectic, owing to the relations (2.2.12). By Lemma 2.2.2, $\boldsymbol{r}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{s} - \boldsymbol{p}^{\mathrm{T}}\,\mathrm{d}\boldsymbol{q}$ is a total derivative if and only if φ is symplectic. □

Consider the coordinate transformation $\boldsymbol{z} \mapsto \widetilde{\boldsymbol{z}} = (\boldsymbol{s}, \boldsymbol{q})$. As long as $\partial\boldsymbol{s}/\partial\boldsymbol{p}$ is invertible this is well-defined. Given a scalar function $S(\boldsymbol{s}, \boldsymbol{q})$, the relations

$$\boldsymbol{r} = \frac{\partial S}{\partial \boldsymbol{s}}(\boldsymbol{s}, \boldsymbol{q}), \quad \boldsymbol{p} = -\frac{\partial S}{\partial \boldsymbol{q}}(\boldsymbol{s}, \boldsymbol{q}) \tag{2.2.13}$$

allow us to construct a symplectic transformation $(\boldsymbol{q}, \boldsymbol{p}) \mapsto (\boldsymbol{s}, \boldsymbol{r})$ [6]. Conversely, the theorem just proven shows that every symplectic map is 'generated', through (2.2.13), by some function $S(\boldsymbol{s}, \boldsymbol{q})$.

We now move on and introduce some ideas from discrete Hamiltonian mechanics. Given a discrete Lagrangian L_d, define the *discrete Legendre transforms* mapping the discrete state space $Q \times Q$ to T^*Q by

$$\begin{aligned} \mathbb{F}^+ L_d &: (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \mapsto (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1}) = \big(\boldsymbol{q}_{k+1}, D_2 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1})\big), \\ \mathbb{F}^- L_d &: (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \mapsto (\boldsymbol{q}_k, \boldsymbol{p}_k) = \big(\boldsymbol{q}_k, -D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1})\big). \end{aligned} \tag{2.2.14}$$

As with the continuous case, we will assume here that both maps are global isomorphisms.

The discrete Legendre transforms give a useful interpretation of the discrete Euler–Lagrange equations (2.1.4). Write the momenta at the right and left endpoints of $[kh, (k+1)h]$ as

$$p_{k,k+1}^+ = \mathbb{F}^+ L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \qquad \text{and} \qquad p_{k,k+1}^- = \mathbb{F}^- L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}),$$

respectively. The kth discrete Euler–Lagrange equation is then merely the statement that the momentum at time kh evaluated from $[(k-1)h, kh]$ is the same as that evaluated from $[kh, (k+1)h]$:

$$\begin{aligned} 0 &= D_2 L_d(\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) + D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = \mathbb{F}^+ L_d(\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) - \mathbb{F}^- L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \\ &= p^+_{k-1,k} - p^-_{k,k+1}. \end{aligned}$$

We call this phenomenon *momentum matching*. Since the momentum $\boldsymbol{p}_k$ is thus uniquely defined, we may view a discrete trajectory $\{\boldsymbol{q}_k\}_{k=0}^N$ in Q as either a trajectory $\big\{(\boldsymbol{q}_k, \boldsymbol{q}_{k+1})\big\}_{k=0}^{N-1}$ in $Q \times Q$, or as a trajectory $\big\{(\boldsymbol{q}_k, \boldsymbol{p}_k)\big\}_{k=0}^N$ in T^*Q.

Starting with a discrete Lagrangian L_d, we define the *discrete Hamiltonian map* $\widetilde{\Phi}_{L_d} : T^*Q \to T^*Q$ by

$$\widetilde{\Phi}_{L_d} = \mathbb{F}^- L_d \circ \Phi_{L_d} \circ (\mathbb{F}^- L_d)^{-1}, \tag{2.2.15}$$

which, in coordinates, is a one-step update map $\widetilde{\Phi}_{L_d} : (\boldsymbol{q}_k, \boldsymbol{p}_k) \mapsto (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1})$, where

$$\boldsymbol{p}_k = -D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}), \quad \boldsymbol{p}_{k+1} = D_2 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}).$$

Example 2.2.5. The discrete Legendre transforms for the n-particle Lagrangian with the rectangle rule discretization (Example 2.1.2) are the maps

$$\begin{aligned} \boldsymbol{p}_k &= M\boldsymbol{v}_{k+1/2} + h\nabla V(\boldsymbol{q}_k), \qquad (2.2.16a) \\ \boldsymbol{p}_{k+1} &= M\boldsymbol{v}_{k+1/2}. \end{aligned}$$

To form the discrete Hamiltonian map (2.2.15) for this Lagrangian, we solve (2.2.16a) for $\boldsymbol{q}_{k+1}$, then evaluate $\boldsymbol{q}_{k+2}$ via (2.1.7), and finally use (2.2.16a) again (with $\boldsymbol{q}_{k+1}$ and $\boldsymbol{q}_{k+2}$) to find $\boldsymbol{p}_{k+1}$.

If evaluating the discrete Hamiltonian map in three steps, as in Example 2.2.5, is not suitable, then an alternative is the two-step formulation

$$(\boldsymbol{q}_k, \boldsymbol{p}_k) \xrightarrow{(\mathbb{F}^- L_d)^{-1}} (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \xrightarrow{\mathbb{F}^+ L_d} (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1}), \tag{2.2.17}$$

to which (2.2.15) is equivalent [12].

2.3 Forcing

Numerical methods in discrete mechanics can be applied to problems involving external forcing of a system. Just as it is important in the unforced setting to use numerical methods that respect the geometry of the system, by adding forcing to the above schemes we can generate methods for forced systems that are more robust than other classical methods. Much greater detail is entered into in the monograph of Marsden and West [12].

Following the structure of the preceding material, we begin with Lagrangian systems. A *Lagrangian force* is a map $f_L : TQ \to T^*Q$ satisfying

$$f_L : (\boldsymbol{q}, \dot{\boldsymbol{q}}) \mapsto (\boldsymbol{q}, f_L(\boldsymbol{q}, \dot{\boldsymbol{q}}))$$

(in other words, f_L preserves the copies of Q that exist (locally) within TQ). In the presence of such a force, we extremize the functional of (2.1.2) with the addition of a force term. If we extremize by considering variations $\boldsymbol{q}(t) + \epsilon\delta\boldsymbol{q}(t)$ such that $\delta\boldsymbol{q}(0) = \delta\boldsymbol{q}(T) = 0$, then we seek curves $\boldsymbol{q}$ such that

$$\delta \int_0^T L(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t))\,\mathrm{d}t + \int_0^T f_L(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t)) \cdot \delta\boldsymbol{q}(t)\,\mathrm{d}t = 0. \tag{2.3.1}$$

Integration by parts then yields the *forced Euler–Lagrange equations*

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{\boldsymbol{q}}} - \frac{\partial L}{\partial \boldsymbol{q}} - f_L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = 0. \tag{2.3.2}$$

In the discrete case, the continuous force is replaced by two *discrete Lagrangian forces* $f_d^+, f_d^- : Q \times Q \to T^*Q$ such that we approximate the virtual work by

$$\int_{kh}^{(k+1)h} f_L(\boldsymbol{q}, \dot{\boldsymbol{q}}) \cdot \delta\boldsymbol{q}(t)\,\mathrm{d}t \approx f_d^+(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \cdot \delta\boldsymbol{q}_{k+1} + f_d^-(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \cdot \delta\boldsymbol{q}_k. \tag{2.3.3}$$

The discrete variational analogue of the variational principle (2.3.1) gives rise to the *forced discrete Euler–Lagrange equations*

$$D_2L_d(\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) + D_1L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) + f_{k-1}^+ + f_k^- = 0 \qquad k = 1, \ldots, N-1, \tag{2.3.4}$$

where $f_k^\pm \stackrel{\text{def}}{=} f_d^\pm(\boldsymbol{q}_k, \boldsymbol{q}_{k+1})$. System (2.3.4) in turn gives rise to the *forced discrete Lagrangian map*

$$\Phi_{L_d}^f : (\boldsymbol{q}_{k-1}, \boldsymbol{q}_k) \mapsto (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}). \tag{2.3.5}$$

From the Hamiltonian perspective, a *Hamiltonian force* is a map $f_H : T^*Q \to T^*Q$ satisfying $(\boldsymbol{q}, \boldsymbol{p}) \mapsto \big(\boldsymbol{q}, f_H(\boldsymbol{q}, \boldsymbol{p})\big)$, and the *forced Hamilton's equations* are the system

$$\dot{\boldsymbol{z}} = J\,\nabla H(\boldsymbol{z}) + \big(0, f_H(\boldsymbol{z})\big)^{\mathrm{T}}. \tag{2.3.6}$$

Using $\mathbb{F}L$ we have

$$f_L = f_H \circ \mathbb{F}L,$$

and if the Lagrangian and Hamiltonian are related by (2.2.4), then the forced Euler–Lagrange and forced Hamilton's equations are equivalent [12].

We can define *forced discrete Legendre transforms* by

$$\begin{aligned}
\mathbb{F}^{f+}L_d &: (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \mapsto (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1}) = \big(\boldsymbol{q}_{k+1}, D_2L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) + f_k^+\big),\\
\mathbb{F}^{f-}L_d &: (\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) \mapsto (\boldsymbol{q}_k, \boldsymbol{p}_k) = \big(\boldsymbol{q}_k, -D_1L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) - f_k^-\big),
\end{aligned}$$

which lead to the *forced discrete Hamiltonian map*

$$\widetilde{\Phi}^f_{L_d} = \mathbb{F}^{f-}L_d \circ \Phi^f_{L_d} \circ (\mathbb{F}^{f-}L_d)^{-1}, \tag{2.3.7}$$

given in coordinates by $\widetilde{\Phi}^f_{L_d} : (\boldsymbol{q}_k, \boldsymbol{p}_k) \mapsto (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1})$, with

$$\boldsymbol{p}_k = -D_1L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) - f_k^-, \quad \boldsymbol{p}_{k+1} = D_2L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) + f_k^+.$$

2.4 Integrators

We now discuss the second theme of this chapter, which is the useful properties of the numerical methods that were defined (abstractly) in the preceding sections.

We met several examples of *integrators* in Chapter 1: an integrator is a numerical method that approximates a continuous ODE system. The discrete Lagrangian and Hamiltonian maps (2.1.5) and (2.2.15) are then further examples of integrators, as are their forced counterparts (2.3.5) and (2.3.7). Our interest lies with integrators that preserve the geometric properties of Lagrangian and Hamiltonian systems. When dealing with Hamiltonian systems, Theorem 2.2.3 justifies the use of *symplectic* numerical methods: a numerical method is said to be *symplectic* if its flow is symplectic

whenever the method is applied to a smooth Hamiltonian system. Example 1.0.1 demonstrated the qualitative behaviour of some simple numerical methods applied to the pendulum problem.

Example 2.4.1. Given a Hamiltonian system with Hamiltonian H, we can prove symplecticity of the *symplectic Euler method*

$$\Phi_h(\boldsymbol{z}_k) = \boldsymbol{z}_k + hJ\,\nabla H(\boldsymbol{q}_k, \boldsymbol{p}_{k+1})$$

by differentiating with respect to $\boldsymbol{z}_k$:

$$\begin{pmatrix} I & -hH_{\boldsymbol{pp}} \\ 0 & I + hH_{\boldsymbol{qp}} \end{pmatrix} \frac{\partial \Phi_h}{\partial \boldsymbol{z}_k} = \begin{pmatrix} I + hH_{\boldsymbol{qp}} & 0 \\ -hH_{\boldsymbol{qq}} & I \end{pmatrix},$$

with the second-partials being evaluated at $(\boldsymbol{q}_k, \boldsymbol{p}_{k+1})$. Then (2.2.9) is easily verified.

Example 2.4.2. The *implicit midpoint rule* is another symplectic method. This integrator is

$$\Phi_h(\boldsymbol{z}_k) = \boldsymbol{z}_k + hJ\,\nabla H\big((\boldsymbol{z}_{k+1} + \boldsymbol{z}_k)/2\big) \tag{2.4.1}$$

and its Jacobian satisfies

$$\left(I - \frac{h}{2} J\,\nabla^2 H\right) \frac{\partial \Phi_h}{\partial \boldsymbol{z}_k} = \left(I + \frac{h}{2} J\,\nabla^2 H\right).$$

Using the results of Section 2.2 we can go further, and show that the abstract discrete Hamiltonian map is symplectic. Suppose that a discrete path $\{\boldsymbol{q}_k\}_{k=0}^N$ satisfies the discrete Euler–Lagrange equations (2.1.4). Then

$$\begin{aligned} \boldsymbol{p}_{k+1} &= D_2 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = \frac{\partial}{\partial \boldsymbol{q}_{k+1}} L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}), \\ \boldsymbol{p}_k &= -D_1 L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = \frac{\partial}{\partial \boldsymbol{q}_k} L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}), \end{aligned}$$

so that, since $\{\boldsymbol{q}_k\}$ solves (2.1.4),

$$\mathrm{d}L_d = \boldsymbol{p}_{k+1}\,\mathrm{d}\boldsymbol{q}_{k+1} - \boldsymbol{p}_k\,\mathrm{d}\boldsymbol{q}_k.$$

By Theorem 2.2.4, the map $(\boldsymbol{q}_k, \boldsymbol{p}_k) \mapsto (\boldsymbol{q}_{k+1}, \boldsymbol{p}_{k+1})$ is symplectic, and its generating function is L_d. Furthermore, every symplectic map has a generating function, and so

can be viewed as resulting from Hamilton's principle via the discrete Lagrangian that is the said generating function. If a numerical method is derived from a variational principle, it is called a *variational integrator*. The above shows that symplectic and variational integrators are one and the same.

Example 2.4.3. To write the midpoint rule (2.4.1) as a variational integrator, assume we are given a Hamiltonian H and a Lagrangian L defined by (2.2.4). Consider the discrete Lagrangian

$$L_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}) = hL(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}), \tag{2.4.2}$$

where we write $\boldsymbol{q}_{k+1/2}$ for $(\boldsymbol{q}_{k+1} + \boldsymbol{q}_k)/2$ (the notation $\boldsymbol{v}_{k+1/2}$ was introduced in Example 2.1.2). The discrete Legendre transforms (2.2.14) are

$$\begin{aligned} \boldsymbol{p}_k &= -\frac{h}{2}\frac{\partial L}{\partial \boldsymbol{q}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}) + \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}), \\ \boldsymbol{p}_{k+1} &= \frac{h}{2}\frac{\partial L}{\partial \boldsymbol{q}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}) + \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}), \end{aligned} \tag{2.4.3}$$

from which we find

$$\frac{\boldsymbol{p}_{k+1} - \boldsymbol{p}_k}{h} = \frac{\partial L}{\partial \boldsymbol{q}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}), \tag{2.4.4a}$$

$$\frac{\boldsymbol{p}_{k+1} + \boldsymbol{p}_k}{2} = \frac{\partial L}{\partial \dot{\boldsymbol{q}}}(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}). \tag{2.4.4b}$$

Following the earlier notational simplification, in future write $\boldsymbol{p}_{k+1/2}$ for $(\boldsymbol{p}_{k+1}+\boldsymbol{p}_k)/2$ and $\boldsymbol{a}_{k+1/2}$ for $(\boldsymbol{p}_{k+1} - \boldsymbol{p}_k)/h$. Equation (2.4.4b) says that

$$(\boldsymbol{q}_{k+1/2}, \boldsymbol{p}_{k+1/2}) = \mathbb{F}L(\boldsymbol{q}_{k+1/2}, \boldsymbol{v}_{k+1/2}),$$

and then (2.2.5a) implies that (2.4.4a) is equivalent to the equation

$$-\partial H/\partial \boldsymbol{q}(\boldsymbol{q}_{k+1/2}, \boldsymbol{p}_{k+1/2}) = \boldsymbol{a}_{k+1/2},$$

while (2.2.5b) implies that

$$\partial H/\partial \boldsymbol{p}(\boldsymbol{q}_{k+1/2}, \boldsymbol{p}_{k+1/2}) = \boldsymbol{v}_{k+1/2}.$$

Thus the discrete Lagrangian (2.4.2) has as discrete Hamiltonian map the implicit midpoint rule.

In the sequel we will use the term *(implicit) midpoint rule* to mean the ODE method (2.4.1) or the discrete Lagrangian (2.4.2), whenever this is clear from context.

Example 2.4.4. For the n-particle Lagrangian $L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = (\dot{\boldsymbol{q}}^{\mathrm{T}} M \dot{\boldsymbol{q}})/2 - V(\boldsymbol{q})$, the discrete Legendre transforms (2.4.3) are

$$\boldsymbol{p}_k = M\boldsymbol{v}_{k+1/2} + \frac{h}{2}\nabla V(\boldsymbol{q}_{k+1/2}), \quad \boldsymbol{p}_{k+1} = M\boldsymbol{v}_{k+1/2} - \frac{h}{2}\nabla V(\boldsymbol{q}_{k+1/2}).$$

2.5 Backward Error Analysis

Backward error analysis involves finding a differential equation whose exact flow equals the numerical flow of a given method. We will use it to explain some of the features of Figure 1.3, but the theory can be furthered to explain the long-time behaviour of geometric integrators [5], [6].

Consider an ODE system $\dot{\boldsymbol{z}} = \boldsymbol{f}(\boldsymbol{z})$ with $\boldsymbol{f}$ smooth. Given a numerical method $\Phi_h : \boldsymbol{z}_k \mapsto \boldsymbol{z}_{k+1}$, we can find a differential equation whose flow $\widetilde{\varphi}_t$ satisfies $\widetilde{\varphi}_h(\boldsymbol{z}_0) = \Phi_h(\boldsymbol{z}_0)$ by considering Taylor expansions. Assume that Φ_h can be expanded as

$$\Phi_h(\boldsymbol{z}_0) = \boldsymbol{z}_0 + h\boldsymbol{f}(\boldsymbol{z}_0) + h^2\boldsymbol{d}_2(\boldsymbol{z}_0) + h^3\boldsymbol{d}_3(\boldsymbol{z}_0) + \cdots,$$

the functions $\boldsymbol{d}_j$ being known. By Taylor-expanding the exact flow of the *modified equation*

$$\dot{\boldsymbol{z}} = \boldsymbol{f}(\boldsymbol{z}) + h\boldsymbol{f}_2(\boldsymbol{z}) + h^2\boldsymbol{f}_3(\boldsymbol{z}) + \cdots \tag{2.5.1}$$

and comparing like powers of h we can derive recurrence relations for the unknown functions $\boldsymbol{f}_j(\boldsymbol{z})$ [6]. In the example that follows, the $\boldsymbol{f}_j$ are computed from first principles.

Example 2.5.1. In addition to the theoretical justification presented in this chapter, this example provides concrete evidence of the worth of symplectic methods. We carry out a backward error analysis of the methods that were applied to the pendulum problem in Chapter 1.

Integrate the system $\dot{\boldsymbol{z}} = J(p, -\sin q)^{\mathrm{T}}$ using the family of integrators

$$\begin{aligned} v_{1/2} &= \alpha p_0 + (1-\alpha)p_1, \\ a_{1/2} &= -\sin((1-\alpha)q_0 + \alpha q_1), \end{aligned} \tag{2.5.2}$$

where $\alpha \in [0,1]$. When $\alpha = 0$ we recover the symplectic Euler method, and $\alpha = 1/2$ gives the implicit midpoint rule ($\alpha = 1$ is a second type of symplectic Euler method that will not feature elsewhere).

The numerical flow of (2.5.2) has Taylor expansion

$$\begin{aligned} \Phi_h(\boldsymbol{z}_0) = \Phi_0(\boldsymbol{z}_0) &+ h\frac{\mathrm{d}}{\mathrm{d}h}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) + \frac{h^2}{2}\frac{\mathrm{d}^2}{\mathrm{d}h^2}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) \\ &+ \frac{h^3}{6}\frac{\mathrm{d}^3}{\mathrm{d}h^3}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) + \mathcal{O}(h^4). \end{aligned} \tag{2.5.3}$$

Routine calculation verifies that the derivatives of Φ_h are

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}h}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) &= (p_0, -\sin q_0)^{\mathrm{T}}, \\ \frac{\mathrm{d}^2}{\mathrm{d}h^2}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) &= -2((1-\alpha)\sin q_0, \alpha p_0 \cos q_0)^{\mathrm{T}}, \\ \frac{\mathrm{d}^3}{\mathrm{d}h^3}\bigg|_{h=0}\Phi_h(\boldsymbol{z}_0) &= 3(2\alpha(\alpha-1)p_0\cos q_0, 2\alpha(1-\alpha)\sin q_0 \cos q_0 + \alpha^2 p_0^2 \sin q_0)^{\mathrm{T}}. \end{aligned}$$

Taylor-expanding the exact flow $\widetilde{\varphi}_h$ of the modified equation (2.5.1) yields

$$\begin{aligned} \widetilde{\varphi}_h(\boldsymbol{z}_0) = \boldsymbol{z}_0 &+ h\big(\boldsymbol{f}(\boldsymbol{z}_0) + h\boldsymbol{f}_2(\boldsymbol{z}_0) + h^2\boldsymbol{f}_3(\boldsymbol{z}_0)\big) \\ &+ \frac{h^2}{2}\frac{\mathrm{d}}{\mathrm{d}t}\bigg|_{t=0}(\boldsymbol{f}(\boldsymbol{z}) + h\boldsymbol{f}_2(\boldsymbol{z})) + \frac{h^3}{6}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\bigg|_{t=0}\boldsymbol{f}(\boldsymbol{z}) + \mathcal{O}(h^4). \end{aligned}$$

The derivatives of $\boldsymbol{f}$ are

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\bigg|_{t=0}\boldsymbol{f}(\boldsymbol{z}) &= \begin{pmatrix} 0 & 1 \\ -\cos q_0 & 0 \end{pmatrix}\dot{\boldsymbol{z}}_0, \\ \frac{\mathrm{d}^2}{\mathrm{d}t^2}\bigg|_{t=0}\boldsymbol{f}(\boldsymbol{z}) &= \begin{pmatrix} 0 & 0 \\ \dot{q}_0 \sin q_0 & 0 \end{pmatrix}\dot{\boldsymbol{z}}_0 + \begin{pmatrix} 0 & 1 \\ -\cos q_0 & 0 \end{pmatrix}\ddot{\boldsymbol{z}}_0. \end{aligned}$$

Since $\boldsymbol{z}$ satisfies the modified equation (2.5.1),

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\bigg|_{t=0}\boldsymbol{f}(\boldsymbol{z}) &= \big(-\sin q_0 + hf_{2,2}(\boldsymbol{z}_0), -p_0\cos q_0 - h\cos(q_0)f_{2,1}(\boldsymbol{z}_0)\big)^{\mathrm{T}} + \mathcal{O}(h^2), \\ \frac{\mathrm{d}^2}{\mathrm{d}t^2}\bigg|_{t=0}\boldsymbol{f}(\boldsymbol{z}) &= (-p_0\cos q_0, p_0^2 \sin q_0 + \sin q_0 \cos q_0)^{\mathrm{T}} + \mathcal{O}(h), \end{aligned}$$

where $f_{2,i}$ denotes the ith component of $\boldsymbol{f}_2$. Therefore

$$\begin{aligned}\widetilde{\varphi}_h(\boldsymbol{z}_0) &= \boldsymbol{z}_0 + h\boldsymbol{f}(\boldsymbol{z}_0) + h^2\boldsymbol{f}_2(\boldsymbol{z}_0) - \frac{h^2}{2}(\sin q_0, p_0\cos q_0)^{\mathrm{T}} \\ &\quad + h^3\boldsymbol{f}_3(\boldsymbol{z}_0) + \frac{h^3}{2}\frac{\mathrm{d}}{\mathrm{d}t}\bigg|_{t=0}\boldsymbol{f}_2(\boldsymbol{z}) + \frac{h^3}{2}\big(f_{2,2}(\boldsymbol{z}_0), -\cos q_0 f_{2,1}(\boldsymbol{z}_0)\big)^{\mathrm{T}} \\ &\quad + \frac{h^3}{6}(-p_0\cos q_0, p_0^2\sin q_0 + \sin q_0\cos q_0)^{\mathrm{T}} + \mathcal{O}(h^4).\end{aligned} \tag{2.5.4}$$

We can now compare like powers of h in (2.5.3) and (2.5.4), and doing so we discover first that

$$\boldsymbol{f}_2(\boldsymbol{z}_0) = (\alpha - 1/2)(\sin q_0, -p_0\cos q_0)^{\mathrm{T}},$$

and thus that the time-derivative of $\boldsymbol{f}_2(\boldsymbol{z})$ must be

$$\frac{\mathrm{d}}{\mathrm{d}t}\bigg|_{t=0}\boldsymbol{f}_2(\boldsymbol{z}) = (\alpha - 1/2)(p_0\cos q_0, p_0^2\sin q_0 + \sin q_0\cos q_0)^{\mathrm{T}} + \mathcal{O}(h).$$

We then deduce that

$$\boldsymbol{f}_3(\boldsymbol{z}_0) = \alpha(\alpha - 1/6)(p_0\cos q_0, -\sin q_0\cos q_0 + p_0^2\sin q_0/2)^{\mathrm{T}} + \mathcal{O}(h).$$

For the symplectic Euler method given by $\alpha = 0$, we thus have the modified equation

$$\dot{\boldsymbol{z}} = \begin{pmatrix} p \\ -\sin q \end{pmatrix} + \frac{h}{2}\begin{pmatrix} -\sin q \\ p\cos q \end{pmatrix} + \frac{h^2}{12}\begin{pmatrix} 2p\cos q \\ (p^2 - 2\cos q)\sin q \end{pmatrix} + \mathcal{O}(h^3). \tag{2.5.5}$$

Direct computation shows that this modified equation is Hamiltonian with

$$\widetilde{H}(q,p) = \frac{1}{2}p^2 - \cos q - \frac{h}{2}p\sin q + \frac{h^2}{12}(p^2 - \cos q)\cos q + \mathcal{O}(h^3).$$

Therefore the exact solutions of (2.5.5) stay on the level curves of $\widetilde{H}$, which agree with those of H to $\mathcal{O}(h)$.

The implicit midpoint rule ($\alpha = 1/2$) can be treated similarly. The modified Hamiltonian $\widetilde{H}$ this time is

$$\widetilde{H}(q,p) = \frac{1}{2}p^2 - \cos q - \frac{h^2}{48}(2p^2 - \cos 2q) + \mathcal{O}(h^3), \tag{2.5.6}$$

which agrees with the exact Hamiltonian H to order $\mathcal{O}(h^2)$. Consequently, the trajectories of the implicit midpoint rule agree with the exact trajectories more closely than do those of the symplectic Euler.

We see in the same way that the modified equation for the explicit Euler method is

$$\dot{\boldsymbol{z}} = \begin{pmatrix} p \\ -\sin q \end{pmatrix} + \frac{h}{2}\begin{pmatrix} \sin q \\ p\cos q \end{pmatrix} + \frac{h^2}{12}\begin{pmatrix} -4p\cos q \\ (p^2 + 4\cos q)\sin q \end{pmatrix} + \mathcal{O}(h^3), \tag{2.5.7}$$

and for implicit Euler the same but with h replaced by $-h$. The $\mathcal{O}(h)$ term of (2.5.7) causes the origin to be a source for the explicit Euler and a sink for the implicit Euler method, behaviour that we previously observed. Indeed, the divergence of the vector field $\dot{\boldsymbol{z}}$ in (2.5.7) is $h\cos q + (h^2 p \sin q)/2 + \mathcal{O}(h^3)$.

A more in-depth treatment of *modified Hamiltonians* is given in the book of Hairer, Lubich and Wanner [6]. In particular, they show that symplectic methods applied to Hamiltonian systems have modified equations that are Hamiltonian, and they describe how to derive the modified Hamiltonian from the generating function of the method.

Chapter 3

Numerical Experiments I

It is time to apply the theory from Chapter 2 to some practical problems. Forced systems occur frequently in optimal control theory, which is briefly introduced in Section 3.1. Our first control experiment is with a (linearized) pendulum in Section 3.2; a second and more advanced experiment presented in Section 3.3 concerns the propulsion-controlled orbital transfer of a satellite. Another illustration of the theory can be found in Appendix A, where we configure a group of hovercraft with minimal control effort.

In this (and subsequent) chapters, the reader may refer to the online documentation by The MathWorks, Inc., for more information on any of MATLAB's native functions that we use [13].

3.1 Optimal Control Theory

Let Q be the configuration space for a given mechanical system. We wish to move the system along a curve $\boldsymbol{q}(t) \in Q$, where $t \in [0, T]$, by applying a force f chosen such that a given cost functional

$$J(\boldsymbol{q}, f) = \int_0^T C(\boldsymbol{q}(t), \dot{\boldsymbol{q}}(t), f(t))\,\mathrm{d}t \tag{3.1.1}$$

is minimized. Here, C can be thought of as measuring the instantaneous cost. This problem is one of infinite-dimensional minimization with constraints given by the

dynamics of the motion; that is, the curve $\boldsymbol{q}$ is constrained by the forced Euler–Lagrange equations (2.3.2), where the Lagrangian is that of the system which is being considered.

To make this problem finite-dimensional we could discretize the cost functional and the underlying equations of motion. More desirable for the constraints is to use the techniques of Chapter 2, and to let the forced discrete Euler–Lagrange equations (2.3.4) be the discrete equations of motion. In the notation of Section 2.3, the cost (3.1.1) is discretized by

$$J_d(\boldsymbol{q}_d, f_d) = \sum_{k=0}^{N-1} C_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}, f_k, f_{k+1}),$$

where the discrete cost function C_d is such that

$$C_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}, f_k, f_{k+1}) \approx \int_{kh}^{(k+1)h} C(\boldsymbol{q}, \dot{\boldsymbol{q}}, f)\,\mathrm{d}t. \tag{3.1.2}$$

The control problem is a boundary value problem, and so the boundary conditions at $t = 0$ and $t = T$ need to be taken into account. The conditions on position, $\boldsymbol{q}(0) = \boldsymbol{q}^0$ and $\boldsymbol{q}(T) = \boldsymbol{q}^1$, can be given to the optimizer as linear equality constraints, but the conditions on velocity must be handled more carefully. Let $\dot{\boldsymbol{q}}(0) = \dot{\boldsymbol{q}}^0$ and $\dot{\boldsymbol{q}}(T) = \dot{\boldsymbol{q}}^1$. We require that the discrete momenta at each endpoint match the exact momenta there, so we insist that

$$\begin{aligned} \mathbb{F}L(\boldsymbol{q}_0, \dot{\boldsymbol{q}}_0) &= \mathbb{F}^{f-}L_d(\boldsymbol{q}_0, \boldsymbol{q}_1), \\ \mathbb{F}L(\boldsymbol{q}_N, \dot{\boldsymbol{q}}_N) &= \mathbb{F}^{f+}L_d(\boldsymbol{q}_{N-1}, \boldsymbol{q}_N). \end{aligned}$$

These become the *discrete boundary conditions*

$$\begin{aligned} D_2L(\boldsymbol{q}_0, \dot{\boldsymbol{q}}_0) + D_1L_d(\boldsymbol{q}_0, \boldsymbol{q}_1) + f_0^- &= 0, \\ -D_2L(\boldsymbol{q}_N, \dot{\boldsymbol{q}}_N) + D_2L_d(\boldsymbol{q}_{N-1}, \boldsymbol{q}_N) + f_{N-1}^+ &= 0. \end{aligned} \tag{3.1.3}$$

3.2 Linearized Pendulum

Our first numerical experiment concerns the motion of a forced linearized simple pendulum. We start with the set-up of Example 1.0.1, from which the Lagrangian is

readily seen to be

$$L(q, \dot{q}) = \frac{1}{2}\dot{q}^2 + \cos q.$$

In order to make a closed-form exact solution available for the optimal control problem that we are going to consider, we 'linearize' by approximating $\cos q \approx 1 - q^2/2$. We apply a control force $f(t)$ to the system, with the intention of using f to bring the pendulum to a given final position at $T = \pi/2$. Let $q^0 = 3\pi/5$, $q^1 = 2\pi/5$, and take the boundary velocities to be zero: $\dot{q}^0 = \dot{q}^1 = 0$. The cost is simply the (squared) L^2-norm of f:

$$J(q, f) = \int_0^{\pi/2} f(t)^2 \, \mathrm{d}t.$$

We compare three approaches to solving the problem: a discrete Euler–Lagrange approach using the midpoint rule (2.4.2); applying the explicit Euler method to the forced Hamilton's equations; and applying the midpoint rule to these forced equations. For the first case, the derivatives of L_d are (cf. Example 2.4.3)

$$\begin{aligned} D_1 L_d(q_k, q_{k+1}) &= -\frac{h}{2} q_{k+1/2} - v_{k+1/2}, \\ D_2 L_d(q_k, q_{k+1}) &= -\frac{h}{2} q_{k+1/2} + v_{k+1/2}, \end{aligned}$$

since $D_1 L(q, \dot{q}) = -q$ and $D_2 L(q, \dot{q}) = \dot{q}$. Using the midpoint rule to approximate the work, we find

$$\int_{kh}^{(k+1)h} f(t) \cdot \delta q(t) \, \mathrm{d}t \approx h f_{k+1/2} \cdot \delta q_{k+1/2} = \frac{h}{2} f_{k+1/2} \cdot \delta q_k + \frac{h}{2} f_{k+1/2} \cdot \delta q_{k+1}.$$

Thus our discrete Lagrangian forces (2.3.3) are $f_k^- = f_k^+ = h f_{k+1/2}/2$. Similarly, the discrete cost (3.1.2) is

$$C_d(q_k, q_{k+1}, f_k, f_{k+1}) = h C(q_{k+1/2}, v_{k+1/2}, f_{k+1/2}).$$

In summary, we have the equality-constrained nonlinear optimization problem

minimize

$$J_d(q_d, f_d) = h \sum_{k=0}^{N-1} (f_{k+1/2})^2 \tag{3.2.1}$$

subject to the equations of motion (3.1.3) and (2.3.4), and with boundary constraints $q_0 = q^0$, $q_N = q^1$.

The forced Hamilton's equations[1] (2.3.6) for the system are $\dot{q} = p$ and $\dot{p} = -q + f$, so that the explicit Euler method gives constraints

$$v_{k+1/2} = p_k, \quad a_{k+1/2} = -q_k + f_k, \tag{3.2.2}$$

and along with a rectangle rule discretization of the relevant integrals (see Example 2.1.2), the discrete cost functional is

$$J_d(q_d, f_d) = h \sum_{k=0}^{N-1} f_k^2. \tag{3.2.3}$$

With this numerical method there are no additional boundary conditions for the velocity. We can evaluate the constraints on the boundary momenta by using the (continuous) Legendre transform for the system. These constraints are then combined with those on the positions before being passed to the solver. In this case we have $p_0 = \dot{q}^0$ and $p_N = \dot{q}^1$.

Let us recap the methods being used.

Euler Minimize discrete cost functional (3.2.3) subject to Hamilton's equations, explicit Euler discretization (3.2.2) of DEs.

Exact Compute exact solution ((3.2.5), below).

Ham. Eq'ns Minimize discrete cost functional (3.2.1) subject to Hamilton's equations, midpoint rule discretization (2.4.1) of DEs.

Var. Minimize discrete cost functional (3.2.1) subject to discrete Euler–Lagrange constraints, midpoint rule discretization (2.4.2) of Lagrangian.

We implement the numerical methods in MATLAB with a common driver file, Appendix D.1.3, and we solve using `fmincon`. The function files for the problem are made common, with the user specifying the chosen discretization with a string flag. The gradients of the cost and constraints are straightforward to compute and code, so we inform `fmincon` that these are available, and we use the solver's default tolerances

[1] We should remark that f is independent of q and $\dot{q}$, so $f = f_L = f_H$.

of 10^{-6}. Initial guesses for the optimal trajectories are given by linear interpolation between the boundary values, with the guesses for the initial f_k being uniformly zero.

Optimal control has well-developed theory describing the solution of continuous problems. We remarked earlier that our pendulum system can be described by a closed-form exact solution. It is with this that we compare the results from the above discretizations. To derive the exact solution we use *Pontryagin's Maximum Principle* [16]. We do not state the principle here, but merely point out its application in the course of our computations.

Hamilton's equations write the linearized pendulum as a first order ODE system. For (q^*, f^*) to be optimal, the Maximum Principle asserts the existence of nontrivial real scalars ψ_1, ψ_2 and a scalar function $\mathcal{H}$ given by

$$\mathcal{H}(\psi_1, \psi_2, q, p, f) = -f^2 + \psi_1 p + \psi_2(f - q),$$

with $\dot{\psi}_1 = -\partial\mathcal{H}/\partial q = \psi_2$ and $\dot{\psi}_2 = -\partial\mathcal{H}/\partial p = -\psi_1$. This ODE system for the ψs has solution

$$\psi_1 = c_1 \cos t + c_2 \sin t,$$
$$\psi_2 = c_2 \cos t - c_1 \sin t.$$

Pontryagin's Principle states further that $\mathcal{H}$ achieves its f-maximum at the optimal force f^*. Since there are no constraints on f, we differentiate $\mathcal{H}$ with respect to f and discover a maximum at $f = \psi_2/2$. Substituting this optimal f, the pendulum's motion, as a second-order system, is thus

$$\ddot{q} = -\frac{c_1}{2} \sin t + \frac{c_2}{2} \cos t - q. \tag{3.2.4}$$

The method of undetermined coefficients allows us to find the particular solution of (3.2.4). The optimal solution to our control problem becomes

$$q^*(t) = \kappa_1 \cos t + \kappa_2 \sin t + \frac{c_1}{4} t \cos t + \frac{c_2}{4} t \sin t, \quad p^*(t) = \dot{q}^*(t),$$

with the constant cs and κs determined using the boundary conditions. Substituting $t = 0$ and $t = \pi/2$ we get a linear system for the constants that is easy to solve

exactly. Finally, we can compute the exact cost J^* for the system by integrating $(f^*)^2$. We find

$$J^* = \frac{\pi}{16}(c_1^2 + c_2^2) - \frac{c_1 c_2}{4}. \tag{3.2.5}$$

A comparison of the variational method with the explicit Euler and exact solutions is given in Figure 3.1. Notice the absence of any 'Exact' times-taken, since these values are independent of the number of time-steps; see Appendix D.1.6.

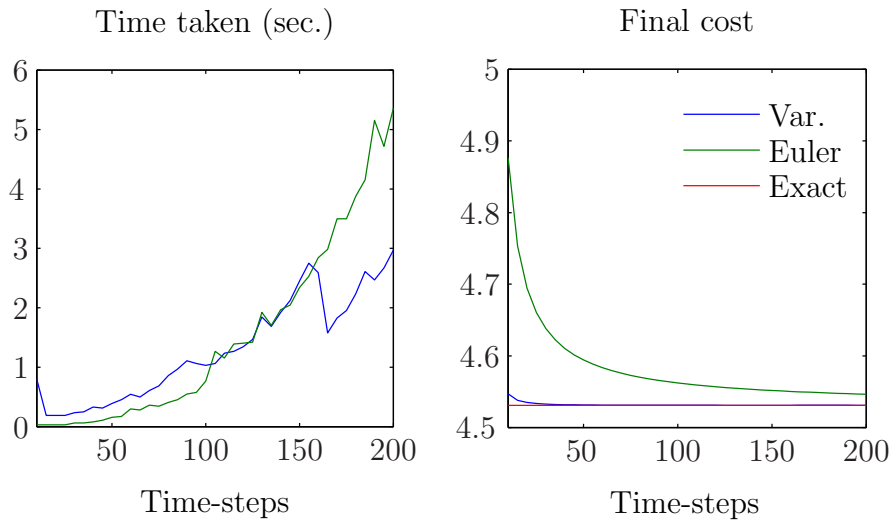


Figure 3.1: Linearized pendulum: variational midpoint vs. explicit Euler vs. exact

The cost for the midpoint rule is almost uniformly equal to the exact cost, even for large step-sizes. The explicit Euler cost tends to the exact value somewhat asymptotically; this is concurrent with our expectations, with the implicit midpoint rule being a 'better' method for Hamiltonian systems. As we saw in (2.5.6), the modified Hamiltonian for the midpoint rule agrees with the exact Hamiltonian to $\mathcal{O}(h^2)$, while for the explicit Euler method the agreement is only to $\mathcal{O}(h)$. We do not see such bad behaviour for the Euler method as in Figure 1.3 because our final state is predetermined, so the trajectories cannot spiral out of control.

Although the explicit Euler approach seems to be faster to begin with, this is certainly offset by the need to take a large number of time-steps to extract an accurate answer. It is expected that the midpoint solution should take longer for small N, since the discrete Euler–Lagrange constraints form a more complicated (nonlinear) system than those of the Euler discretization. However, as N increases we begin to

see the benefits of the fact that the variational optimization problem is half the size of the finite difference one.

Figure 3.2 compares the results from the variational method with the method whose constraints are a midpoint discretization of Hamilton's equations.

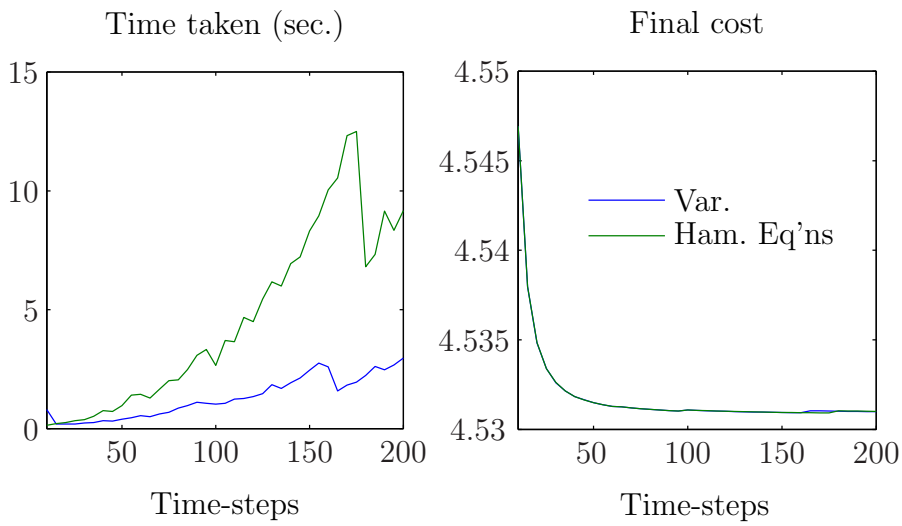


Figure 3.2: Linearized pendulum: midpoint, variational constraints vs. finite diff.

We see that the costs are in close agreement, but the smaller variational system is solved more quickly.

3.3 Orbital Transfer

The second experiment is a variation on the theme of the first. We wish to optimally transfer a satellite between two circular orbits by using a propulsion system [9]. The satellite, whose mass is m, is in orbit around the Earth, whose mass is M. We let the number of revolutions of the Earth made by the satellite to be $p = 1$ or $p = 2$, during which time the craft is to be transferred between two coplanar circular orbits, the final having larger radius than the initial. We work in polar coordinates $\boldsymbol{q} = (r, \varphi)$, in which the Lagrangian is

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}m(\dot{r}^2 + r^2\dot{\varphi}^2) + \frac{\gamma M m}{r}, \tag{3.3.1}$$

with γ being the gravitational constant. The (Lagrangian) force f applied by the satellite's propulsion system is to act in the direction of motion, and so decomposes

as $f(t) = (0, r(t)u(t))^{\mathrm{T}}$ for some scalar function $u(t)$. As for the initial conditions, let $\boldsymbol{q}(0) = (r_0, 0)$. Then $\dot{\boldsymbol{q}}(0) = \big(0, \sqrt{\gamma M/r_0^3}\big)$ is the initial velocity [1]. Write T_0 and T_1 for the initial and final orbital periods of the craft, respectively. We wish for the satellite to reach the point $(r_1, 0)$ at time $T = p(T_0 + T_1)/2$, and then the propulsion system shuts off; the final velocity must therefore be $\dot{\boldsymbol{q}}(T) = \big(0, \sqrt{\gamma M/r_1^3}\big)$. The cost is

$$J(\boldsymbol{q}, u) = \int_0^T u(t)^2 \,\mathrm{d}t,$$

and the parameter values for the experiment are: $\gamma = 6.673 \times 10^{-26}$; $m = 100$ and $M = 6 \times 10^{24}$; $r_0 = 5$ and $r_1 = 6$; and $T_0 = 2\pi\sqrt{r_0^3/(\gamma M)}$ and $T_1 = 2\pi\sqrt{r_1^3/(\gamma M)}$.

We construct the variational formulation of the problem using the midpoint discretization, as in Section 3.2. The derivatives of L are

$$\frac{\partial L}{\partial \boldsymbol{q}} = (mr\dot{\varphi}^2 - \gamma Mm/r^2, 0)^{\mathrm{T}}, \qquad \frac{\partial L}{\partial \dot{\boldsymbol{q}}} = (m\dot{r}, mr^2\dot{\varphi})^{\mathrm{T}},$$

and the discrete forces are $f_k^- = f_k^+ = h(0, r_{k+1}u_{k+1} + r_k u_k)^{\mathrm{T}}/4$. For the explicit Euler discretization, formula (2.2.4) gives that the Hamiltonian is

$$H(\boldsymbol{q}, \boldsymbol{p}) = \frac{1}{2m}\left(p_r^2 + \frac{p_\varphi^2}{r^2}\right) - \frac{\gamma Mm}{r},$$

so that the forced Hamilton's equations are

$$\dot{r} = \frac{p_r}{m}, \quad \dot{\varphi} = \frac{p_\varphi}{mr^2}, \qquad \dot{p}_r = \frac{p_\varphi^2}{mr^3} - \frac{\gamma Mm}{r^2}, \quad \dot{p}_\varphi = ru,$$

with boundary conditions

$$\big(\boldsymbol{q}(0), \boldsymbol{p}(0)\big) = \big(r_0, 0, 0, \sqrt{\gamma Mm^2 r_0}\,\big),$$
$$\big(\boldsymbol{q}(T), \boldsymbol{p}(T)\big) = \big(r_1, 0, 0, \sqrt{\gamma Mm^2 r_1}\,\big).$$

The code for this experiment is more involved than for the pendulum, so the set-up in MATLAB is slightly different. This time our methods are as follows.

Euler Minimize discrete cost functional subject to discrete Hamilton's equations; rectangle rule discretization of integrals, explicit Euler discretization of DEs.

Mid. Minimize discrete cost functional subject to discrete Euler–Lagrange constraints, midpoint rule discretization for both.

The driver is called `orbbatch.m` (and its code is given in Appendix D.2.3). This initializes the physical data, and then solves the problem for both discretizations over the ranges $N = 10\colon 5\colon 50$ for $p = 1$ and $N = 10\colon 10\colon 100$ for $p = 2$. The function files are again common, but we anticipate that using this code structure for the image registration problem will not be worthwhile, owing to the M-files there being more complicated still. Gradients are coded for everything but the discrete Euler–Lagrange constraints. Initial guesses are linear, as before. Figures 3.3 and 3.4 show the computation times and final costs for $p = 1$ and $p = 2$.

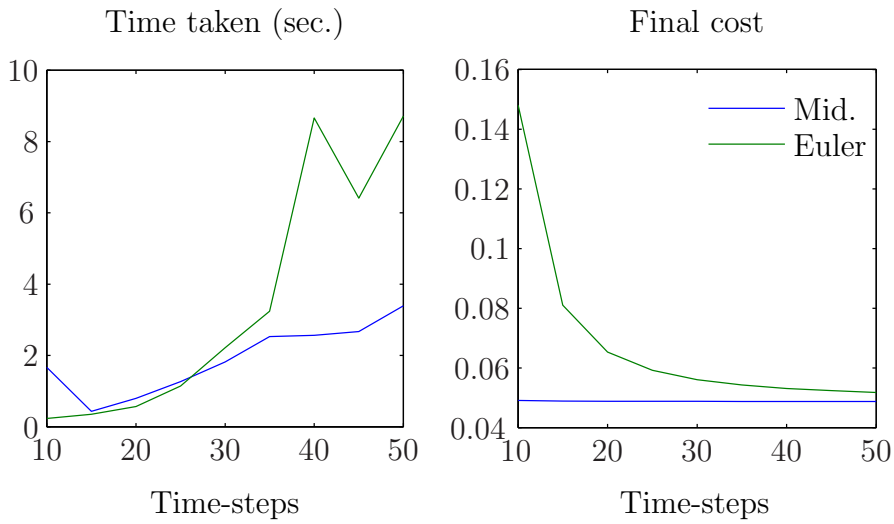


Figure 3.3: Orbital transfer: midpoint vs. explicit Euler (1 revolution)

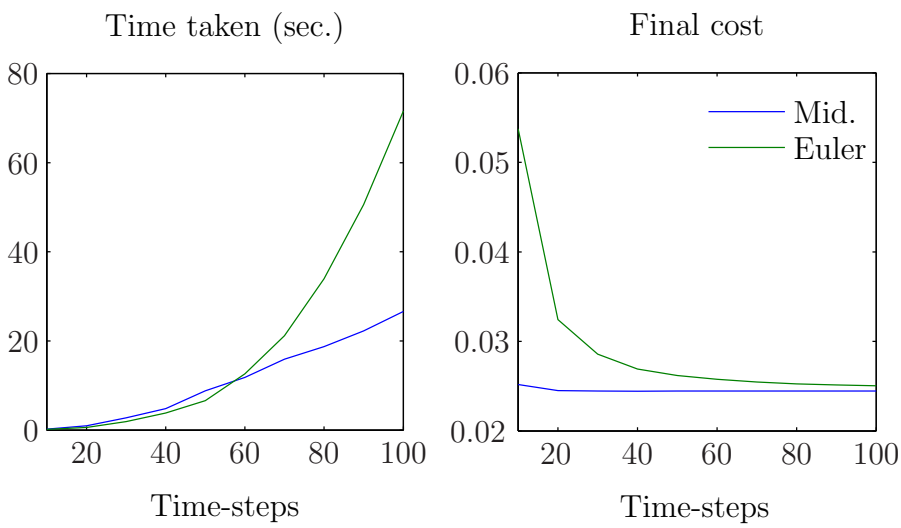


Figure 3.4: Orbital transfer: midpoint vs. explicit Euler (2 revolutions)

We see behaviour similar to that observed in the pendulum experiment, with the

midpoint discretization at least as fast as the Euler. As we did in Figure 3.2 for the pendulum, Junge, Marsden & Ober–Blöbaum compare variational with finite-difference midpoint constraints for the orbital transfer [9].

Instead of comparing our results with an exact solution, we integrate the forced Euler–Lagrange equations using `ode45`. This uses a Runge–Kutta integrator that is more accurate than the midpoint rule. We take as time-span $0\colon 10^{-3}\colon T$, with the force being interpolated from the discrete solution by a cubic spline over this range. Since the forced equations read

$$mr\dot{\varphi}^2 - \frac{\gamma M m}{r^2} - m\ddot{r} = 0,$$
$$mr^2\ddot{\varphi} + 2mr\dot{r}\dot{\varphi} - ru = 0,$$

writing $\dot{r} = Y$ and $\dot{\varphi} = Z$ these become

$$\dot{r} = Y, \quad \dot{\varphi} = Z, \qquad \dot{Y} = rZ^2 - \frac{\gamma M}{r^2}, \quad \dot{Z} = \frac{1}{mr}(u - 2mYZ).$$

The code in Appendix D.2.8 is the script file for solving this ODE system. We thus obtain a third method called 'R–K'.

R–K Interpolate optimal force from 'Mid.' by cubic spline, integrate forced equations using `ode45`.

Comparisons of the optimal trajectories for 'Mid.', 'Euler' and 'R–K' are given by the polar plots in Figures 3.5 and 3.6. The three orbits are practically indistinguishable, for either value of p.

Although the trajectories appear reasonable, we see from Figure 3.7 that there is noise in the corresponding optimal forces. Figures 3.7–3.9 suggest that more time-steps should be taken in order to resolve the forces correctly.

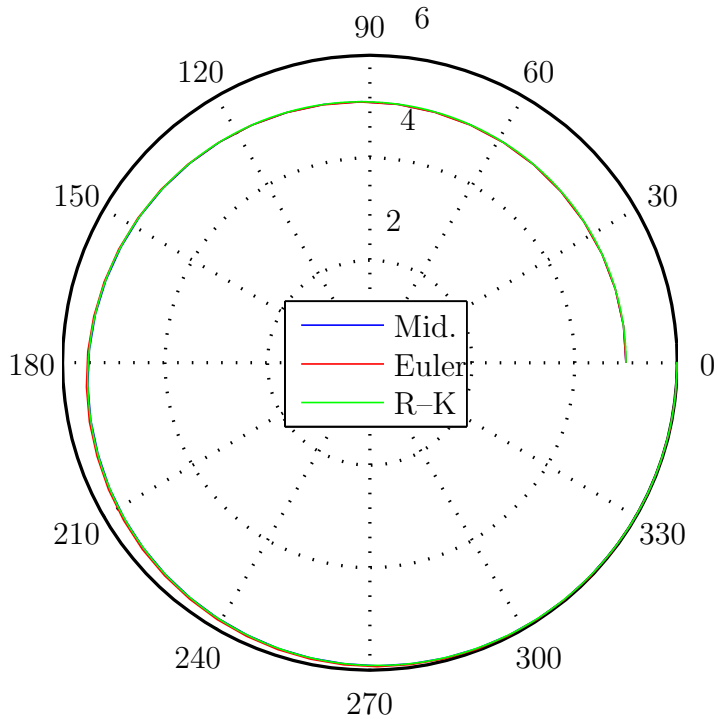


Figure 3.5: Orbital transfer: trajectory comparison ($p = 1$, $N = 50$, φ in degrees)

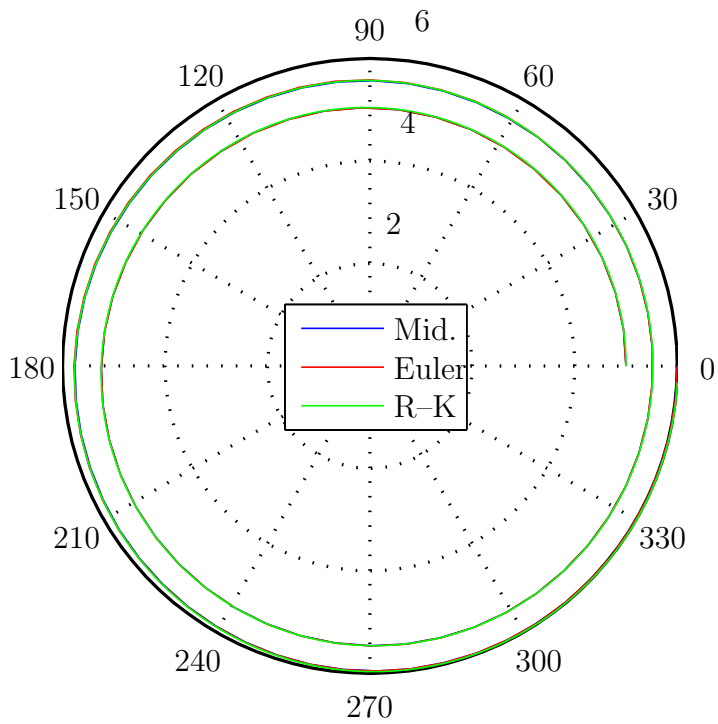


Figure 3.6: Orbital transfer: trajectory comparison ($p = 2$, $N = 100$, φ in degrees)

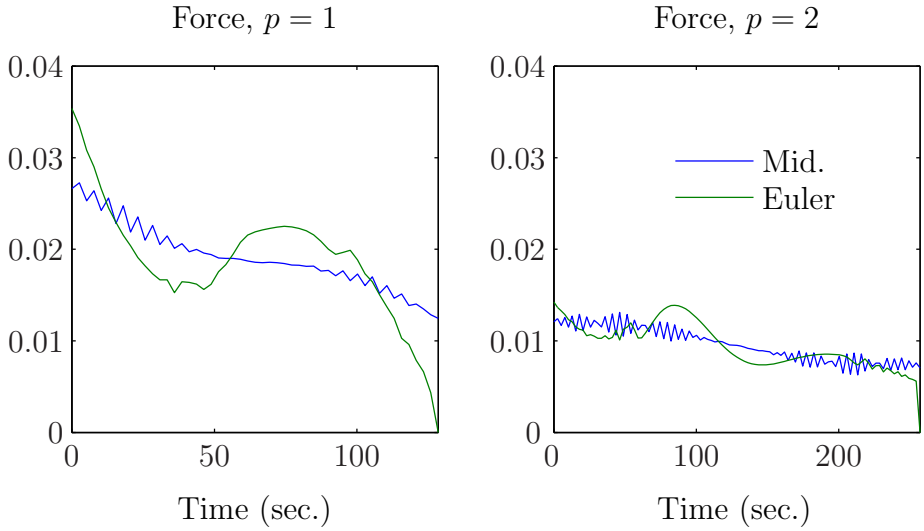


Figure 3.7: Orbital transfer: optimal forces ($N = 50$ (left), $N = 100$ (right))

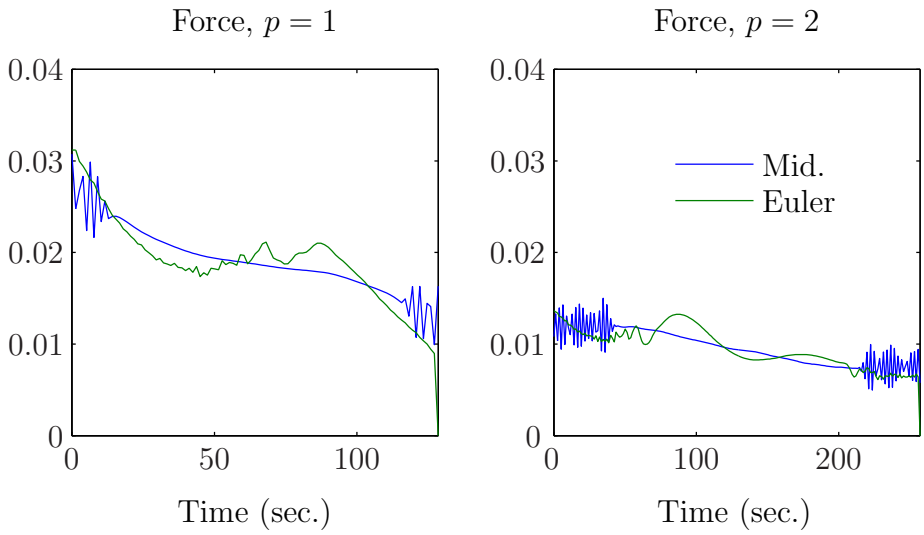


Figure 3.8: Orbital transfer: optimal forces ($N = 100$ (left), $N = 200$ (right))

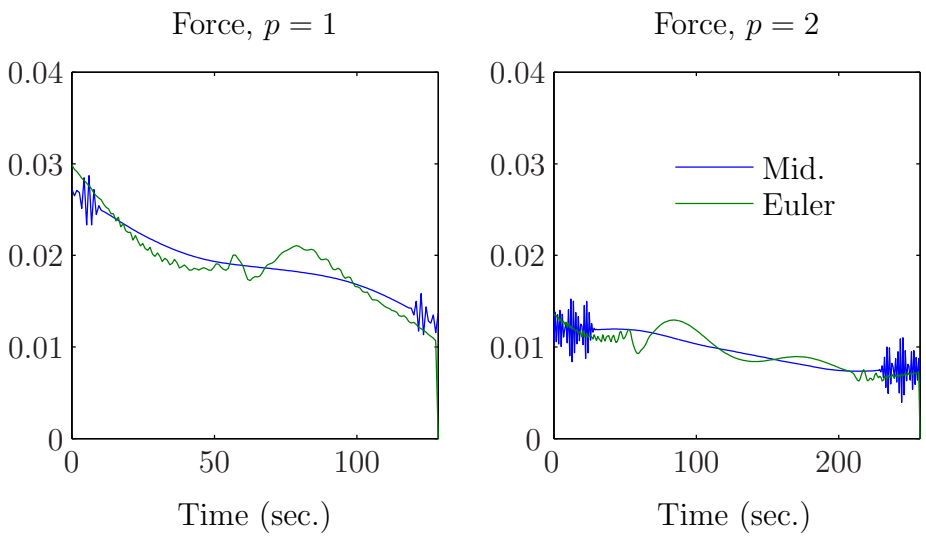


Figure 3.9: Orbital transfer: optimal forces ($N = 150$ (left), $N = 300$ (right))

Chapter 4

Image Registration

Let us apply some variational integrators to the problem of finding a warp between two given images. Our interest is in *control point registration*, and Section 4.1 discusses the techniques for doing this. We will be looking specifically at the method of *geodesic interpolating splines*, and we describe the MATLAB implementation of this in Section 4.2.

4.1 Geodesic Interpolating Splines

In control point registration we assume that we are given a finite set of n_c initial control points on one image, and we are to associate to each a corresponding final control point on a second image. Our notation will be such that the initial control points are $\{\boldsymbol{q}_i^0\}_{i=1}^{n_c}$ and the final are $\{\boldsymbol{q}_i^1\}_{i=1}^{n_c}$. We write the displacements as

$$\boldsymbol{v}_i \stackrel{\text{def}}{=} \boldsymbol{q}_i^1 - \boldsymbol{q}_i^0 \qquad \text{for } i = 1, \ldots, n_c.$$

The images (or regions of interest) should exist within a compact subset of the ambient space; for us this means that we will only be considering images lying within the unit disc $\mathbb{D} \subset \mathbb{R}^2$. We presuppose that all global warping has been carried out. Such warping would be necessary, for example, to orient or scale the two images to match more closely. In particular, all rotations are to have been already performed, and so we demand that $\mathbb{S}^1$ is fixed by the warp. We will denote by $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{q})$ the (full)

displacement field that the warp generates, and thus we assume that both $\boldsymbol{v}$ and its normal derivative are identically zero on $\mathbb{S}^1$. The local warp should interpolate, in a suitable way, the control point displacements by a diffeomorphic displacement field, where a map is called *diffeomorphic* if it is a differentiable bijection with differentiable inverse (with such a degree of smoothness we can expect no tearing or folding to occur). Unless we make clear what is meant by 'suitable', the problem as it stands is badly-formulated. As mentioned in Chapter 1, in practice this depends on the application. It is common to want to do the least work necessary, which in this case means that we should minimize an energy associated with the warp. We focus on the *bending energy* of the displacement field [4], which is

$$E(\boldsymbol{v}) = \int_{\mathbb{D}} (\Delta \boldsymbol{v}(\boldsymbol{q})) \cdot (\Delta \boldsymbol{v}(\boldsymbol{q}))\, \mathrm{d}\boldsymbol{q}, \tag{4.1.1}$$

with Δ denoting the Laplacian.

Before we discuss the current state-of-the-art for control-point registration, a reminder. Recall that, for $\boldsymbol{a} \in \mathbb{R}^2$, the *$\delta$-functional* $\delta_{\boldsymbol{a}} : C^0(\mathbb{R}^2) \to \mathbb{R}^2$ is defined by

$$\delta_{\boldsymbol{a}} : f \mapsto f(\boldsymbol{a}).$$

We shall follow the convention (in certain circles) of considering δ as a function rather than as a distribution, so that $\delta(\boldsymbol{x} - \boldsymbol{a}) \stackrel{\text{def}}{=} \delta_{\boldsymbol{a}}$ satisfies

$$\delta(\boldsymbol{x} - \boldsymbol{a}) = 0 \qquad \text{if } \|\boldsymbol{x} - \boldsymbol{a}\| \neq 0 \tag{4.1.2a}$$

and

$$\int_{\mathbb{R}^2} \delta(\boldsymbol{x} - \boldsymbol{a}) f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = f(\boldsymbol{a}), \qquad \text{for all } f. \tag{4.1.2b}$$

Twining, Marsland & Taylor show that to minimize the bending energy (4.1.1) subject to $\boldsymbol{v}(\boldsymbol{q}_i^0) = \boldsymbol{v}_i$, one must solve the boundary value problem [19]

$$\Delta^2 \boldsymbol{v}(\boldsymbol{q}) = \sum_{i=1}^{n_c} \delta(\boldsymbol{q} - \boldsymbol{q}_i^0)(\boldsymbol{v}(\boldsymbol{q}) - \boldsymbol{v}_i) \qquad \text{for } \boldsymbol{q} \in \mathbb{D}, \tag{4.1.3a}$$

$$\boldsymbol{v}(\boldsymbol{q}) = \frac{\partial \boldsymbol{v}}{\partial n} = 0 \qquad \text{for } \boldsymbol{q} \in \mathbb{S}^1. \tag{4.1.3b}$$

The *Green's function* associated with (4.1.3) is $G : \mathbb{D} \times \mathbb{D} \to \mathbb{R}$ such that $G(\boldsymbol{q}, \boldsymbol{y})$, as a function of $\boldsymbol{q}$, satisfies

$$\begin{aligned} \Delta^2 G(\boldsymbol{q}, \boldsymbol{y}) &= \delta(\boldsymbol{q} - \boldsymbol{y}) && \text{for } \boldsymbol{q} \in \mathbb{D}, \\ G(\boldsymbol{q}, \boldsymbol{y}) &= \frac{\partial G}{\partial n}(\boldsymbol{q}, \boldsymbol{y}) = 0 && \text{for } \boldsymbol{q} \in \mathbb{S}^1. \end{aligned} \tag{4.1.4}$$

A range of approaches to the control point registration problem is discussed by Mills and Twining, Marsland & Taylor [14], [19]. The ansatz for us is that the displacement field $\boldsymbol{v}$ is expanded in terms of the Green's function G as

$$\boldsymbol{v}(\boldsymbol{q}) = \sum_{i=1}^{n_c} \boldsymbol{\alpha}_i G\big(\boldsymbol{q}, \boldsymbol{q}_i^0\big), \tag{4.1.5}$$

where the coefficients $\boldsymbol{\alpha}_i \in \mathbb{R}^2$ are determined by the constraints $\boldsymbol{v}\big(\boldsymbol{q}_j^0\big) = \boldsymbol{v}_j$. In two dimensions, G is

$$G(\boldsymbol{q}, \boldsymbol{y}) = \|\boldsymbol{q} - \boldsymbol{y}\|^2 \left(\frac{A^2 - 1}{2} - \log A \right), \quad A(\boldsymbol{q}, \boldsymbol{y}) = \frac{\sqrt{\|\boldsymbol{q}\|^2 \|\boldsymbol{y}\|^2 - 2\boldsymbol{q} \cdot \boldsymbol{y} + 1}}{\|\boldsymbol{q} - \boldsymbol{y}\|}, \tag{4.1.6}$$

and is well-known [3]. It is not difficult to show that G satisfies the definition (4.1.4).

Once $\boldsymbol{v}$ is found, integration subject to the given initial control-point positions yields an interpolant for the warp that fixes everything outside and on the unit circle. Computing $\boldsymbol{v}$ in the manner just outlined is called the method of *clamped-plate splines*. Sadly, we only generate diffeomorphisms in the limit of small displacements [19].

We can generate diffeomorphic transformations by considering our full warp as a sequence of small clamped-plate spline deformations [19] (we will not attempt to verify that such warps generated by our experiments are diffeomorphisms). If we let the control points trace out paths $\{\boldsymbol{q}_i(t)\}$, with

$$\boldsymbol{q}_i(0) = \boldsymbol{q}_i^0 \quad \text{and} \quad \boldsymbol{q}_i(1) = \boldsymbol{q}_i^1, \tag{4.1.7}$$

this new formulation of the problem can be stated as

minimize

$$J\big(\boldsymbol{q}_i(t), \boldsymbol{v}(\boldsymbol{q}(t), t)\big) = \int_0^1 E(\boldsymbol{v})\, \mathrm{d}t \tag{4.1.8}$$

over fields $\boldsymbol{v}(\boldsymbol{q}(t), t) \in \mathbb{D}$ satisfying (4.1.3b) and paths $\boldsymbol{q}_i(t) \in \mathbb{D}$, subject to the *matching constraints*

$$\dot{\boldsymbol{q}}_i = \boldsymbol{v}(\boldsymbol{q}_i(t), t) \qquad \text{for } t \in [0, 1]$$

and the boundary constraints (4.1.7), for $i = 1, \ldots, n_c$.

The optimal warp generated is called a *geodesic interpolating spline* (GIS)[1]. By writing the problem as one of constrained optimization and substituting the Green's function expansion (4.1.5), it can be shown that to compute the spline we should solve the problem

minimize

$$J = \int_0^1 \sum_{i,j=1}^{n_c} \big(\boldsymbol{\alpha}_i(t) \cdot \boldsymbol{\alpha}_j(t)\big) G(\boldsymbol{q}_i(t), \boldsymbol{q}_j(t))\, \mathrm{d}t$$

over $\boldsymbol{\alpha}_i(t)$ and $\boldsymbol{q}_i(t)$, subject to

$$\dot{\boldsymbol{q}}_i = \sum_{j=1}^{n_c} \boldsymbol{\alpha}_j(t) G(\boldsymbol{q}_i(t), \boldsymbol{q}_j(t)) \qquad \text{for } t \in [0, 1]$$

and the boundary constraints (4.1.7), for $i = 1, \ldots, n_c$.

Discretizing in time then makes the state space finite-dimensional. This is where variational integrators enter the scene. The standard technique is to use an explicit Euler approximation for $\dot{\boldsymbol{q}}_i$ and a rectangle rule approximation for the cost integral. Doing so leads to the problem's final form

minimize

$$J_d = h \sum_{k=0}^{N-1} \sum_{i,j=1}^{n_c} \big(\boldsymbol{\alpha}_{i,k} \cdot \boldsymbol{\alpha}_{j,k}\big) G(\boldsymbol{q}_{i,k}, \boldsymbol{q}_{j,k}) \tag{4.1.10a}$$

over $\boldsymbol{\alpha}_{i,k}$ and $\boldsymbol{q}_{i,k}$, subject to

$$\boldsymbol{v}_{i,k+1/2} = \sum_{j=1}^{n_c} \boldsymbol{\alpha}_{j,k} G(\boldsymbol{q}_{i,k}, \boldsymbol{q}_{j,k}) \tag{4.1.10b}$$

and (4.1.7). We write $\boldsymbol{q}_{i,k} \stackrel{\text{def}}{=} \boldsymbol{q}_i(kh)$, likewise for the $\boldsymbol{\alpha}$s.

[1] A *geodesic* in a space Ω is the shortest path in Ω between two points in the space. Optimizing (4.1.8) corresponds to constructing geodesics in the space of diffeomorphisms [18].

To put the GIS construction in the language of this thesis, notice that what we are doing is computing the motion of n_c particles in $\mathrm{Diff}(\mathbb{D})$, the group of diffeomorphisms of the disc $\mathbb{D}$. (Holm & Marsden consider this problem in much more generality than space permits here [8].) The Lagrangian for the particle system is the kinetic energy

$$L(\boldsymbol{v}) = \frac{1}{2}\|\boldsymbol{v}\|^2 = \frac{1}{2}\int_{\mathbb{D}} \boldsymbol{v} \cdot \mathcal{A}\boldsymbol{v}\,\mathrm{d}\boldsymbol{q}, \tag{4.1.11}$$

with $\mathcal{A}$ being a given positive-definite, symmetric *inertia operator* $\mathcal{A} : T\,\mathrm{Diff}(\mathbb{D}) \to T^*\,\mathrm{Diff}(\mathbb{D})$ that determines the norm $\|\cdot\|$ in the sense just given. Write $\mathcal{A}\boldsymbol{v} = \boldsymbol{m}$; the mapping $\boldsymbol{v} \xrightarrow{\mathcal{A}} \boldsymbol{m}$ represents the Legendre transform for the system. Our inertia operator of interest is $\mathcal{A} = \Delta^2$, and this generates the norm

$$\|\boldsymbol{v}\|^2 = \int_{\mathbb{D}} (\Delta\boldsymbol{v}) \cdot (\Delta\boldsymbol{v})\,\mathrm{d}\boldsymbol{q}$$

upon integration by parts, since we assume that the velocity $\boldsymbol{v}$ contributes a zero boundary term to the integration. The Green's function G for this $\mathcal{A}$ is (4.1.6), and then, since $\mathcal{A}G(\boldsymbol{q},\boldsymbol{y}) = \delta(\boldsymbol{q}-\boldsymbol{y})$, the velocity $\boldsymbol{v}$ is given in terms of G and $\boldsymbol{m}$ by the convolution[2] $\boldsymbol{v} = G * \boldsymbol{m}$. Let $\boldsymbol{q}_1(t), \ldots, \boldsymbol{q}_{n_c}(t)$ be the positions of our control points and $\boldsymbol{p}_1(t), \ldots, \boldsymbol{p}_{n_c}(t)$ their momenta, and from now on let the t-dependence of the $\boldsymbol{q}$s and $\boldsymbol{p}$s be implicit, unless required. Under the particle ansatz (4.1.5) the total momentum $\boldsymbol{m}$ is

$$\boldsymbol{m}(\boldsymbol{q},t) = \sum_{i=1}^{n_c} \boldsymbol{p}_i \delta(\boldsymbol{q}-\boldsymbol{q}_i). \tag{4.1.12}$$

Appendix B outlines why this ansatz makes sense. We see from (4.1.11) that the Hamiltonian for the dynamics of the control points is

$$H = \frac{1}{2}\int_{\mathbb{D}} \mathcal{A}^{-1}\boldsymbol{m} \cdot \boldsymbol{m}\,\mathrm{d}\boldsymbol{q},$$

and so (4.1.12) yields the Hamiltonian

$$H(\boldsymbol{q},\boldsymbol{p}) = \frac{1}{2}\sum_{i,j=1}^{n_c} (\boldsymbol{p}_i \cdot \boldsymbol{p}_j) G(\boldsymbol{q}_i, \boldsymbol{q}_j), \tag{4.1.13}$$

[2]The *convolution*, $f * g$, of two functions f and g is defined by

$$(f * g)(t) = \int f(\tau) g(t-\tau)\,\mathrm{d}\tau.$$

and the expression

$$\boldsymbol{v}(\boldsymbol{q},t) = G * \boldsymbol{m} = \sum_{i=1}^{n_c} \boldsymbol{p}_i G(\boldsymbol{q},\boldsymbol{q}_i)$$

for $\boldsymbol{v}$. Each $\boldsymbol{\alpha}_i$ from (4.1.5) is exactly the momentum $\boldsymbol{p}_i$ [8], so we will use the latter notation from this point on. Hamilton's equations for (4.1.13) are

$$\dot{\boldsymbol{q}}_i = \sum_{j=1}^{n_c} \boldsymbol{p}_j G(\boldsymbol{q}_i,\boldsymbol{q}_j), \tag{4.1.14a}$$

$$\dot{\boldsymbol{p}}_i = -\frac{1}{2}(\boldsymbol{p}_i\cdot\boldsymbol{p}_i)\frac{\partial}{\partial \boldsymbol{q}_i}G(\boldsymbol{q}_i,\boldsymbol{q}_i) - \sum_{j=1,\, j\neq i}^{n_c} (\boldsymbol{p}_i\cdot\boldsymbol{p}_j)\frac{\partial}{\partial \boldsymbol{q}_i}G(\boldsymbol{q}_i,\boldsymbol{q}_j). \tag{4.1.14b}$$

Finally, the Lagrangian for the system can now be written

$$L(\boldsymbol{q},\dot{\boldsymbol{q}}) = \frac{1}{2}\sum_{i,j=1}^{n_c} (\boldsymbol{p}_i\cdot\boldsymbol{p}_j)G(\boldsymbol{q}_i,\boldsymbol{q}_j),$$

with the $\boldsymbol{p}_i$ being given by (4.1.14a), for $i = 1,\ldots,n_c$.

The optimization problem (4.1.10) can be rephrased as

> extremize, over all paths between the initial and final control points, the following discretization of the action functional (2.1.2) for the system whose Hamiltonian is (4.1.13): approximate the integral in the action by the rectangle rule and approximate $\dot{\boldsymbol{q}}$, through the Legendre transform (4.1.14a), by the explicit Euler method (cf. Example 2.1.2).

The hierarchy of methods for solving (4.1.10) can be described as follows. We can solve the underlying discrete variational equations of motion, as we commented following Example 2.1.1, or we can discretize the variational principle and then solve the associated optimization problem. The formulation (4.1.10) does the latter, using a non-symplectic discretization. A solve for the equations of motion can be implemented as a *full solve* (nonlinear) for all $\boldsymbol{q}$ and $\boldsymbol{p}$ in one go, or as a *stepping solve*. More precisely, the 'full solve' finds the $\boldsymbol{q}_{i,k}$ and $\boldsymbol{p}_{i,k}$ that satisfy the time-discretized version of (4.1.14) by solving the resulting system of $4Nn_c$ nonlinear equations. On the other hand, the 'stepping solve' uses the two-step update (2.2.17) to generate $\{\boldsymbol{q}_i(T)\}$ as a function of the initial momenta $\{\boldsymbol{p}_i(0)\}$. We can then minimize, as a function of these momenta, the deviation of $\big\{\boldsymbol{q}_i(T)(\{\boldsymbol{p}_i(0)\})\big\}$ from the fixed final control-point positions. Of course, in exact arithmetic the full solve and stepping solve would produce equal results.

4.2 Implementation

First we make some general comments.

As we mentioned in Section 3.3, the complexity of the GIS problem means that the code for each formulation we are testing stands alone—there are no common driver files. We are interested primarily in comparisons of final costs and of times taken, but for reference purposes we also record the output structure from the relevant solver used. More precisely, all of our solvers return the optimal path in phase space, the initial guess, the output structure, and the time taken to perform the main solve (that is, we do not time any of the initialization subroutines that may be present). Tolerances of 10^{-6} are used throughout.

The initial guess $\boldsymbol{z}_0$ is built in stages. The initial and final control points are generated randomly inside the unit disc, with points that are too close to the boundary being moved inwards a little. An initial guess for the path between each pair $(\boldsymbol{q}_i^0, \boldsymbol{q}_i^1)$ is generated using code (Appendix E.2.2) of Twining, Marsland & Taylor [19]. This generates a curved path by using an analytic expression derived from the case $n_c = 1$ (the authors claim a smaller convergence time compared with constant velocity, straight-line paths). The initial momenta are generated by using the relevant discretization of the Legendre transform (4.1.14a). The specific methods we use are discussed following (4.2.1), below.

The full *costate vector* $\boldsymbol{z}$ consists of the positions and momenta at each time step for every control point, so it is important to make a decision about the ordering of these values within the vector, and to stick with it. We have chosen to structure $\boldsymbol{z}$ as

$$\boldsymbol{z}^{\mathrm{T}} = (\boldsymbol{q}_{1,1}^{\mathrm{T}}, \ldots, \boldsymbol{q}_{1,N+1}^{\mathrm{T}}, \ldots, \boldsymbol{q}_{n_c,1}^{\mathrm{T}}, \ldots, \boldsymbol{q}_{n_c,N+1}^{\mathrm{T}}, \boldsymbol{p}_{1,1}^{\mathrm{T}}, \ldots, \boldsymbol{p}_{1,N+1}^{\mathrm{T}}, \ldots, \boldsymbol{p}_{n_c,1}^{\mathrm{T}}, \ldots, \boldsymbol{p}_{n_c,N+1}^{\mathrm{T}}),$$

remembering that we must index from 1 in MATLAB. In fact, we will follow this indexing convention from now on. Sometimes, for brevity, we will write

$$\boldsymbol{q}_k \stackrel{\text{def}}{=} (\boldsymbol{q}_{1,k}, \ldots, \boldsymbol{q}_{n_c,k})$$

(mutatis mutandis for $\boldsymbol{p}_k$), whenever this causes no confusion.

We saw above that the cost function for the optimization problem is

$$J_d = 2h \sum_{k=1}^{N} H_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}, \boldsymbol{p}_k, \boldsymbol{p}_{k+1}),$$

where H_d is some discretization of the Hamiltonian (4.1.13); for example,

$$H_d(\boldsymbol{q}_k, \boldsymbol{q}_{k+1}, \boldsymbol{p}_k, \boldsymbol{p}_{k+1}) = H\big(\boldsymbol{q}_{k+1/2}, \boldsymbol{p}_{k+1/2}\big)$$

for the midpoint rule. Once the partials (4.1.14) of H are coded, it is easy to include the gradient of J_d in the cost routine.

In order to code analytically-computed partial derivatives of the Hamiltonian, we need first to study the Green's function G. By considering Taylor series it can be shown that

$$\lim_{\boldsymbol{x}\to\boldsymbol{y}} G(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}(\|\boldsymbol{x}\|^2 - 1)^2.$$

Since G is ubiquitous in the equations for the GIS problem, it is important to vectorize the function's M-file as much as possible. Our file `Green.m` (Appendix E.2.5) accepts as input two matrices $\boldsymbol{x}$ and $\boldsymbol{y}$ of size $2 \times m$, and it returns a $1 \times m$ vector whose entries are the values of the Green's function evaluated columnwise on $\boldsymbol{x}$ and $\boldsymbol{y}$. The code in Appendix E.2.6 for the derivative $\partial/\partial\boldsymbol{x}(G(\boldsymbol{x}, \boldsymbol{y}))$ is vectorized too, for the same reason. This derivative is given analytically by

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{x}} G(\boldsymbol{x}, \boldsymbol{y}) &= \big(\boldsymbol{x}\|\boldsymbol{y}\|^2 - \boldsymbol{y}\big)\left(1 - \frac{1}{A^2}\right) - 2(\boldsymbol{x} - \boldsymbol{y})\log(A), \\
\frac{\partial}{\partial \boldsymbol{x}} G(\boldsymbol{x}, \boldsymbol{x}) &= 2\boldsymbol{x}\big(\|\boldsymbol{x}\|^2 - 1\big),
\end{aligned}$$

with A as in (4.1.6). To ensure these computations are correct, simple test scripts compare the analytic derivatives with their definitions as limits. The vectors $\partial H/\partial \boldsymbol{q}_k$ and $\partial H/\partial \boldsymbol{p}_k$ are coded in a way that avoids `for` loops by using the built-in `sum` function and some matrix-masking (Appendix E.2.10, respectively E.2.8). Again, test scripts verify the integrity of the code.

Once the optimal paths are computed, in the manner that we discuss in the following paragraph, the function `GISpaths.m` of Appendix E.2.4 may be used to plot the control points and the paths between.

We now make some comments about the individual methods. We will be looking primarily at the midpoint rule and the explicit Euler method. The optimal control versions of the GIS problem minimize the discretized cost using `fmincon` with its medium-scale algorithm. This is because the large-scale algorithm does not accept nonlinear constraints, these constraints being the discretized Legendre transform. For the full nonlinear solve formulations we use `fsolve`, with initial experiments suggesting that the default medium-scale dogleg algorithm performs slightly better than the other available methods. The function `fsolve` does not accept constraints, so the control-point positions are passed as parameters, with the solver computing only the 'inner' values of $\boldsymbol{q}$ (as well as all the $\boldsymbol{p}$). The stepping nonlinear solves work differently to the other two methods, as we noted at the end of Section 4.1. We update using (2.2.17): for the midpoint rule this entails a nonlinear solve for $\{\boldsymbol{q}_{i,k}\}$ and then a matrix inversion for $\{\boldsymbol{p}_{i,k}\}$; for the explicit Euler method the $\boldsymbol{q}$s and $\boldsymbol{p}$s are updated explicitly. The iterative process repeats until we reach $\{\boldsymbol{q}_{i,N+1}\}$. There are then two options; either we can solve the equations

$$\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1 = 0, \qquad \text{for } i = 1, \ldots, n_c, \tag{4.2.1}$$

using `fsolve`, or we can minimize the norm of the difference (4.2.1) using `fminsearch`. There follows a summary of the approaches being taken. Recall that we discretize using rectangle rule for integrals with explicit Euler for DEs, or midpoint rule for integrals with midpoint rule for DEs.

Full nonlin. Solve discretized variational equations of motion (4.1.14) as a full system, over all $\boldsymbol{q}$ and all $\boldsymbol{p}$ at once.

Opt. control Minimize discretized Lagrangian integral subject to velocity constraints (cf. (4.1.10)).

Step. (`fmin`) Minimize the deviation $\big\|\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1\big\|$, for $i = 1, \ldots, n_c$, as a function of $\{\boldsymbol{p}_i(0)\}$.

Step. (`fsolve`) Solve the system $\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1 = 0$, where $i = 1, \ldots, n_c$, for $\{\boldsymbol{p}_i(0)\}$.

In order that errors do not propagate through the one-step solves inside the stepping methods, for each intermediate `fsolve` we use the default medium-scale algorithm with tolerances of 10^{-8}.

We expect the `fsolve` formulation to be better than `fminsearch` for the stepping solve. The `fsolve` method finds the $2n_c$ components of $\{\boldsymbol{q}_{i,N+1}\}$ using the $2n_c$ equations (4.2.1). In contrast, the `fminsearch` method has to find the optimal components of $\boldsymbol{q}_{i,N+1}$ using only the scalar measure of deviation provided by the norm: in a sense the minimum-search system has less information encoded into it with which the solver can work.

Once the stepping solver finds the optimal initial momenta, the optimal paths are generated by outputting the intermediate $\boldsymbol{q}$s and $\boldsymbol{p}$s generated by $\{\boldsymbol{p}_i(0)\}$.

Chapter 5

Numerical Experiments II

Here are the initial and final control-point data and interpolating paths for $n_c = 3, 4, 5$; see Figure 5.1.

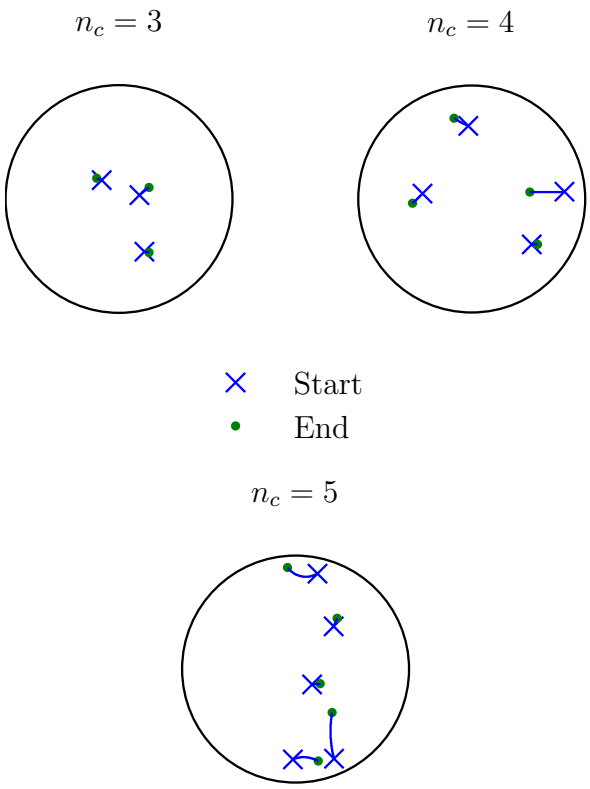


Figure 5.1: GIS: initial control points and paths

5.1 Explicit Euler

We compare the solution to the optimal control explicit Euler optimization problem (4.1.10) with a full nonlinear solve and a stepping nonlinear solve. For the latter we show the results of using `fsolve` with the dogleg algorithm and of using `fminsearch`. The plots in Figures 5.2–5.4 show time taken and final cost for $n_c = 3, 4, 5$, respectively.

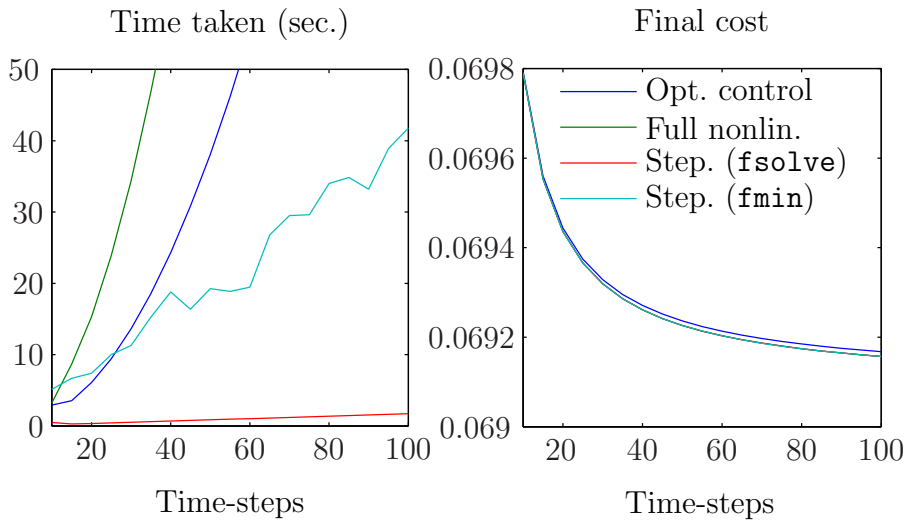


Figure 5.2: GIS: explicit Euler, optimal control vs. nonlinear solve ($n_c = 3$)

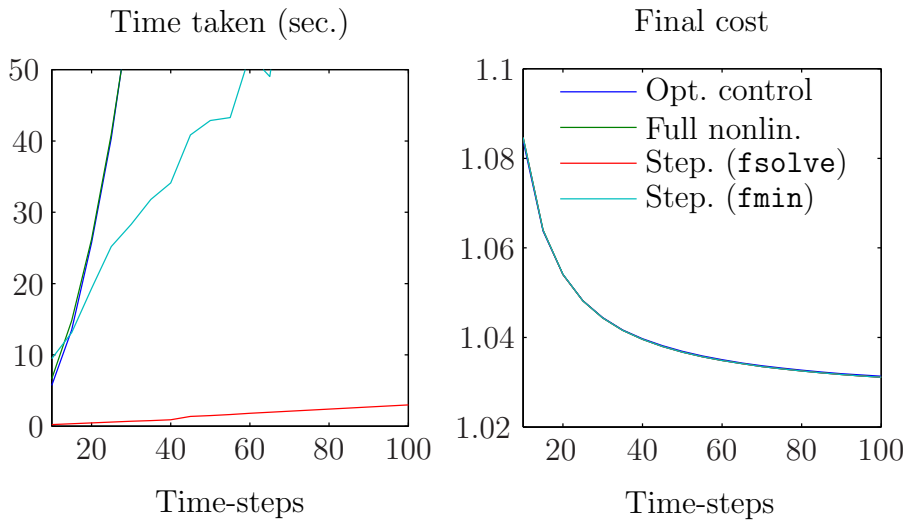


Figure 5.3: GIS: explicit Euler, optimal control vs. nonlinear solve ($n_c = 4$)

The costs for all methods agree closely, with the `fsolve` stepper being undoubtedly the best. For the record, the times (seconds) for $N = 100$ are given in Table 5.1.

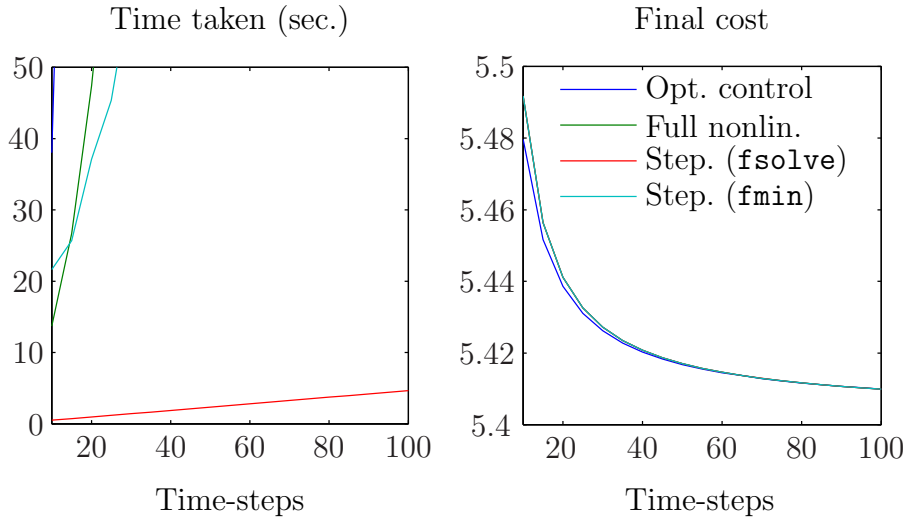


Figure 5.4: GIS: explicit Euler, optimal control vs. nonlinear solve ($n_c = 5$)

The behaviour shown in the figures is certainly to be expected. The explicit updates

n_c	Opt. ctrl	Full nonlin.	Step. (`fsolve`)	Step. (`fmin`)
3	157.5	381.0	1.705	41.80
4	831.6	786.6	2.974	79.81
5	7345	1543	4.663	205.4

Table 5.1: GIS: explicit Euler computation times ($N = 100$)

for the stepping Euler mean that it takes very little time to compute the final positions as a function of the initial momenta. The resulting `fsolve` for this small system then completes much more rapidly than the other methods, and especially more than for the optimization, over *all* positions and momenta, of the cost integral. When the cost is not so small as in Figure 5.2, we do not find the optimal control method to be any faster than the others, as we did in Section 3.2, since the optimal control system is the same size as the others. The time taken by the optimal control method for $n_c = 5$ is linked to the final cost being relatively high compared with that for the other values of n_c. Shardlow notes that optimal control methods can experience difficulties when the cost is high [17], which itself is related to the distribution of the control points (see Figure 5.1).

Having coded an `fsolve` and `fminsearch` stepping nonlinear solve for the other

discretizations in this chapter, we noticed continued poor performance while using `fminsearch`. From this point we will disregard the `fminsearch` formulation and consider only `fsolve` stepping solves.

Finally for the Euler comparisons, the optimal paths are shown in Figures 5.5–5.7.

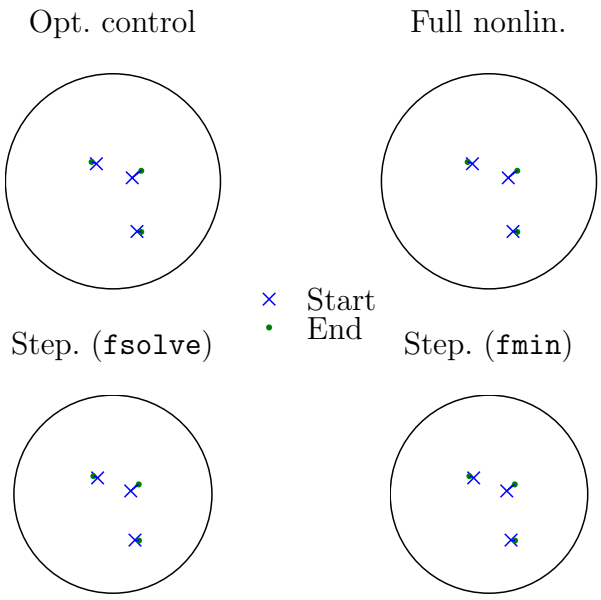


Figure 5.5: GIS: explicit Euler, optimal paths ($n_c = 3$)

We have coded symplectic Euler versions of the four explicit Euler solvers ('Opt. control', 'Full nonlin.', 'Step. (`fsolve`)' and 'Step. (`fmin`)'). We omit our findings for these. For each, the cost generated is in close agreement with that of the corresponding explicit Euler solver. The reason for this is that the GIS problem has a fixed final state, so trajectories generated by the explicit Euler methods are not as bad as they could be for an initial value problem (cf. Example 1.0.1), and they are closer to the trajectories of the symplectic Euler methods than they would be in the IVP case. However, none of the symplectic Eulers can match the rapidity of the `fsolve` stepping Euler method.

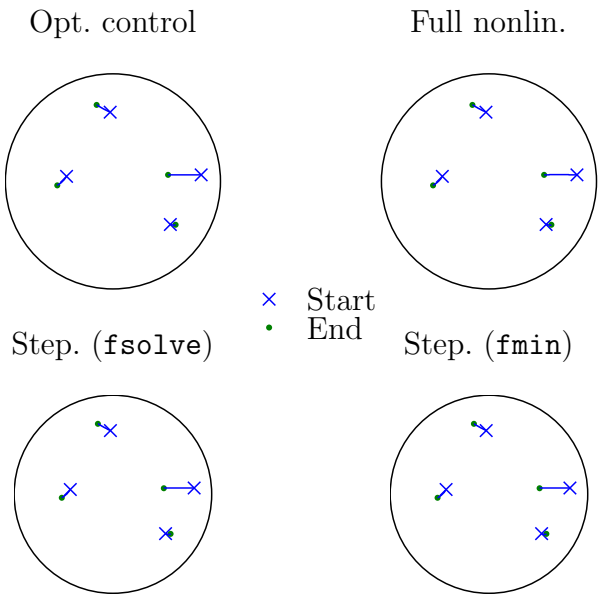


Figure 5.6: GIS: explicit Euler, optimal paths ($n_c = 4$)

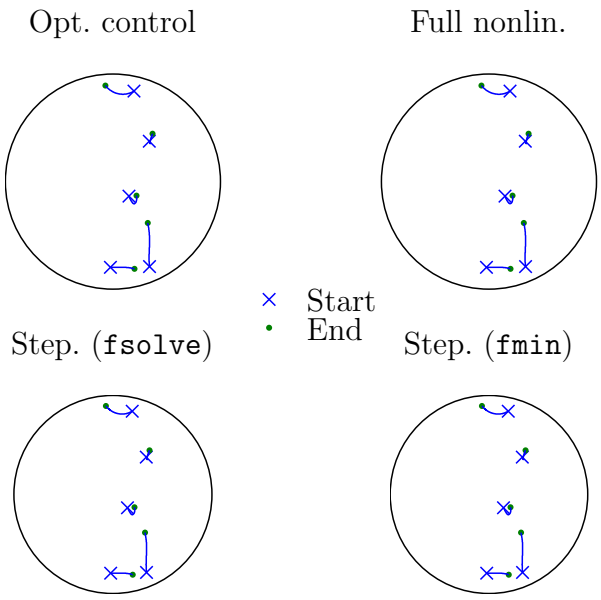


Figure 5.7: GIS: explicit Euler, optimal paths ($n_c = 5$)

5.2 Midpoint Rule

We now wish to compare the three methods that use the midpoint rule: optimal control; a full nonlinear solve for all positions and momenta; and an `fsolve` stepping nonlinear solve for the initial momenta. See Figures 5.8–5.10.

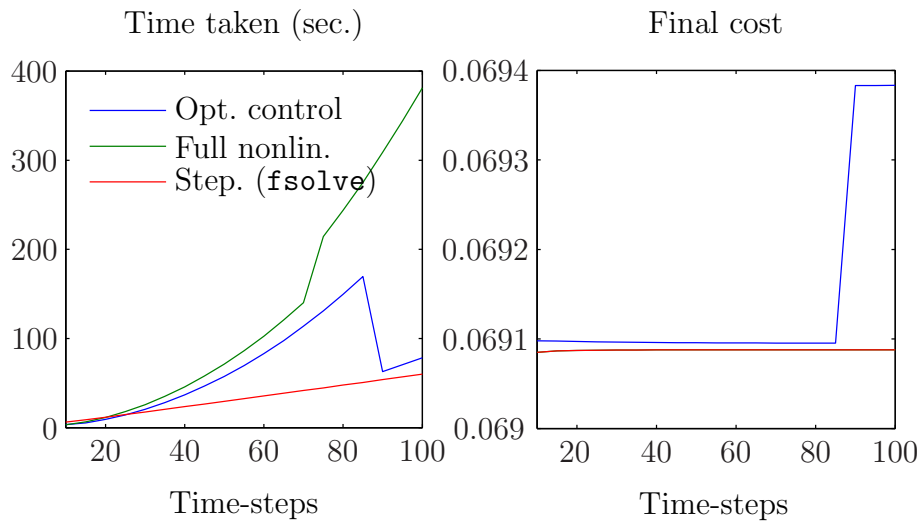


Figure 5.8: GIS: midpoint rule, optimal control vs. nonlinear solve ($n_c = 3$)

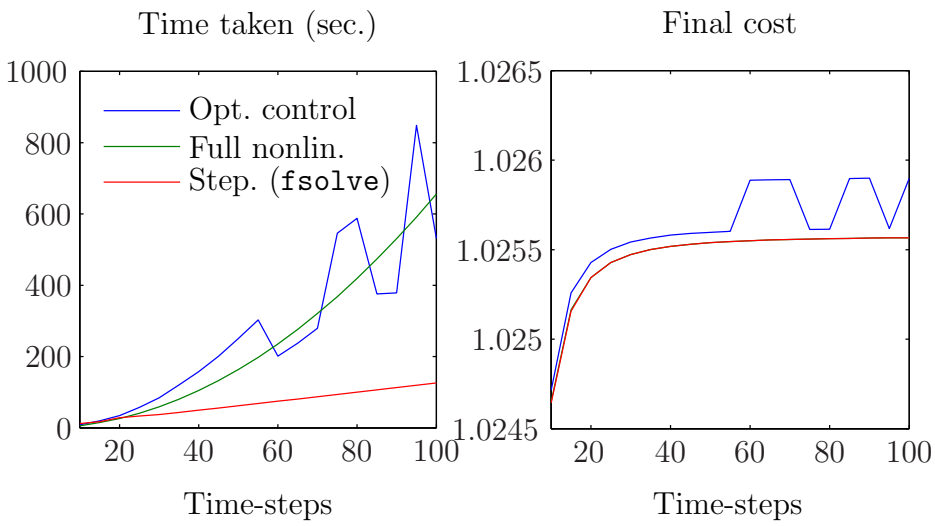


Figure 5.9: GIS: midpoint rule, optimal control vs. nonlinear solve ($n_c = 4$)

It is interesting to observe that for larger N and the first two sets of control points, the optimal control method is suddenly able to solve the problem more quickly, although less accurately. This could be something to do with the tolerances used in `fmincon`'s `options` structure. Away from the jumps in the optimal control costs, the

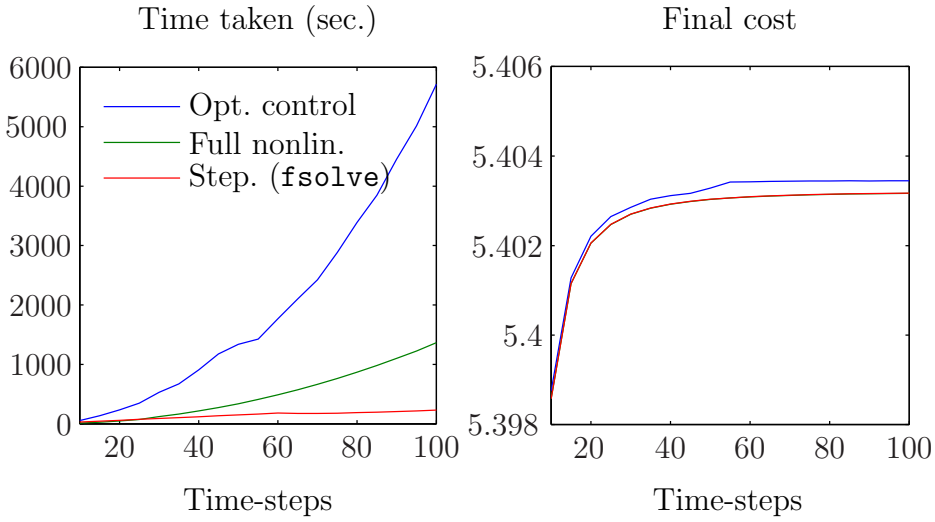


Figure 5.10: GIS: midpoint rule, optimal control vs. nonlinear solve ($n_c = 5$)

slight difference between the optimal control cost curve and the nonlinear solve curves is minor, once the vertical scale is taken into account. We also see, as in Figures 5.2–5.4, that the optimal control method performs quite well when the final cost is small, but less well when the cost is greater. The stepping solver again out-performs the others.

The optimal paths for the midpoint solvers are given in Figures 5.11–5.13.

5.3 Other Tests

Selecting the best methods, based on execution time and accuracy, for the explicit Euler and midpoint discretizations we obtain the following.

Step. exp. Euler Solve $\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1 = 0$, for $i = 1, \ldots, n_c$. Explicit Euler discretization of DEs.

Step. midpoint Solve $\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1 = 0$, for $i = 1, \ldots, n_c$. Midpoint rule discretization of DEs.

As a comparison, we include the results from a stepping integrator that uses `ode113`[1] to evaluate the map $\{\boldsymbol{p}_i(0)\} \mapsto \boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big)$. This solver is initialized with data

[1] `ode113` appears to perform better than `ode45` on this problem, but the difference is small. Neither method is symplectic [2].

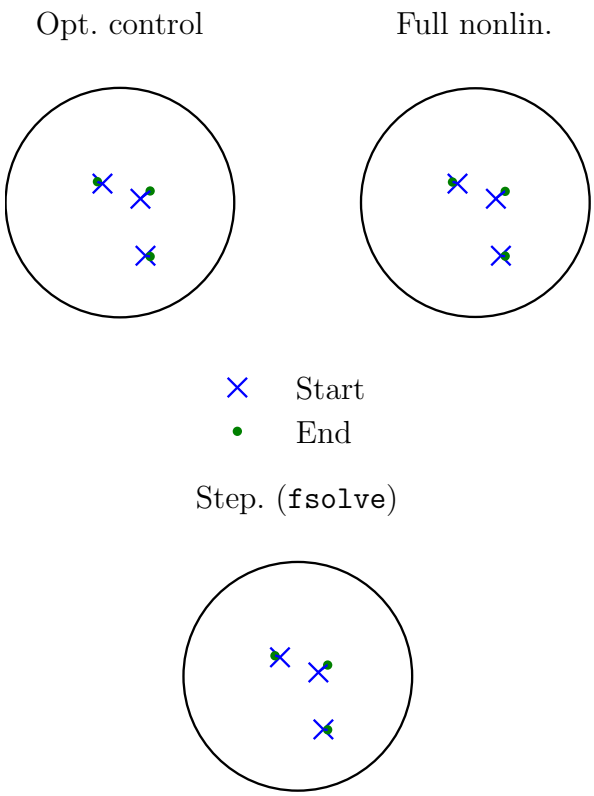


Figure 5.11: GIS: midpoint, optimal paths ($n_c = 3$)

from the midpoint-rule initializer of Appendix E.4.5, leading to what we call 'Step. `ode113`'.

Step. `ode113` Given $\{\boldsymbol{p}_i(0)\}$, integrate equations of motion using `ode113` with initial conditions $\{\boldsymbol{q}_i^0\}$ and $\{\boldsymbol{p}_i(0)\}$. Use result to form function $\{\boldsymbol{p}_i(0)\} \mapsto \boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big)$. Solve $\boldsymbol{q}_{i,N+1}\big(\{\boldsymbol{p}_i(0)\}\big) - \boldsymbol{q}_i^1 = 0$, for $i = 1, \ldots, n_c$.

Figures 5.14–5.16 compare the times taken and costs for 'Step. midpoint', 'Step. exp. Euler' and 'Step. `ode113`' (the midpoint time-curve for $n_c = 5$ is not visible with the scale used). These cost results are consistent with our earlier experience and expectations. Although the midpoint stepping solver out-performed the other midpoint solvers, it still cannot match the explicit Euler stepper and its explicit updates. The `ode113` solver combines the speed of the explicit Euler with the accuracy of the midpoint solver.

A good test of 'Step. exp. Euler', 'Step. midpoint' and 'Step. `ode113`' is to give them data from a real experiment. Small changes to our codes enable the solvers to

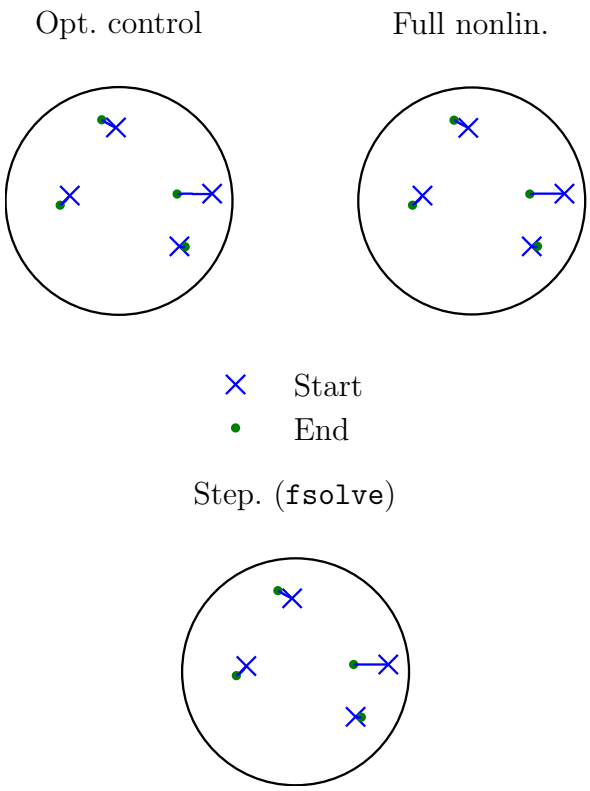


Figure 5.12: GIS: midpoint, optimal paths ($n_c = 4$)

load a MAT-file containing 40 control points, which was provided by Shardlow [17]. Since the number of control points is quite large, we need to find a value for the tolerance in `ode113` that gives the best balance between accuracy and efficiency on the data. As range of tolerances we take

$$(10^{-6}, 2.5 \times 10^{-6}, 5 \times 10^{-6}, 7.5 \times 10^{-6}, 10^{-5}, 2.5 \times 10^{-5}, \ldots, 10^{-2}).$$

We compute the absolute error in the final cost for each tolerance-value by comparing with a benchmark cost, which is generated from a tolerance of 2.24×10^{-14}. See Figure 5.17. The absolute error never exceeds 4×10^{-6} over the tolerance range, so in the comparison that follows we will use the tolerance that corresponds to the least time-taken; that is, a tolerance of 10^{-2}.

On the real data, we found that the midpoint `fsolve` stepping method was not able to handle the larger system well; Table 5.2 displays the results. We again compute the absolute error using the earlier benchmark cost value, which is 1.0089 to 5 significant figures. The costs agree to 3 significant figures, but it seems that the

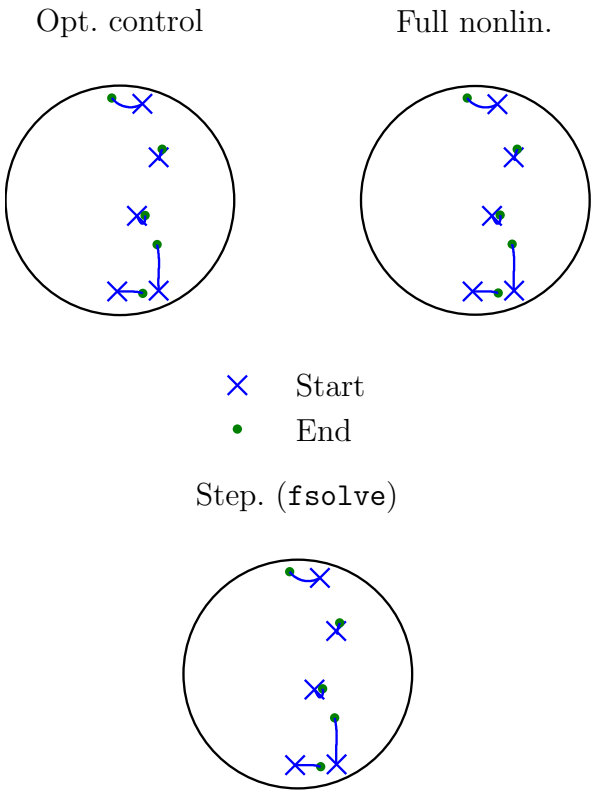


Figure 5.13: GIS: midpoint, optimal paths ($n_c = 5$)

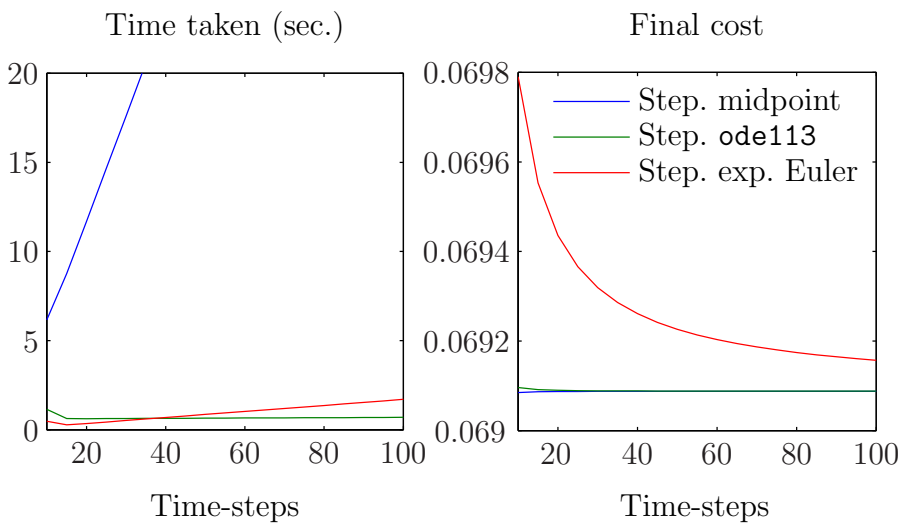


Figure 5.14: GIS: midpoint rule vs. `ode113` vs. explicit Euler ($n_c = 3$)

	Mid.	`ode113`	Exp. Euler
Time (sec.)	9399	63.30	19.14
Abs. error in cost	1.729×10^{-3}	2.803×10^{-6}	3.751×10^{-3}

Table 5.2: GIS: performances on real data ($N = 10$)

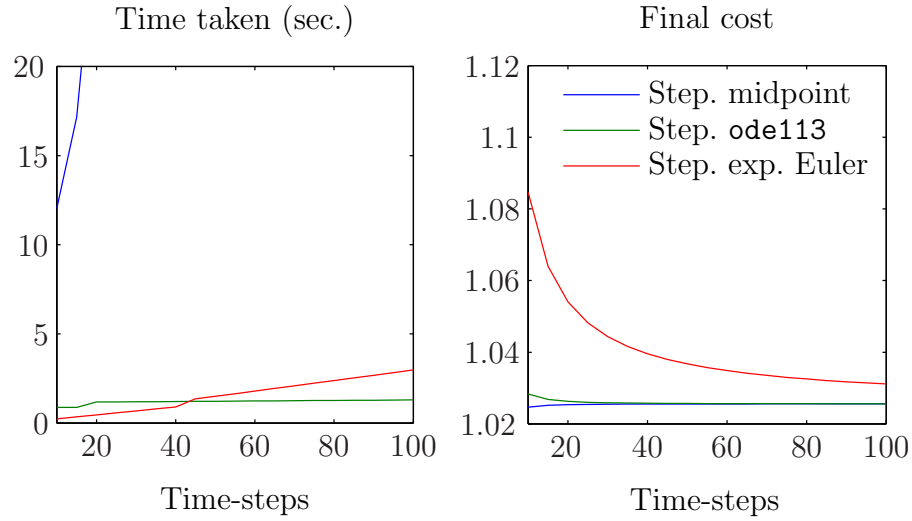


Figure 5.15: GIS: midpoint rule vs. `ode113` vs. explicit Euler ($n_c = 4$)

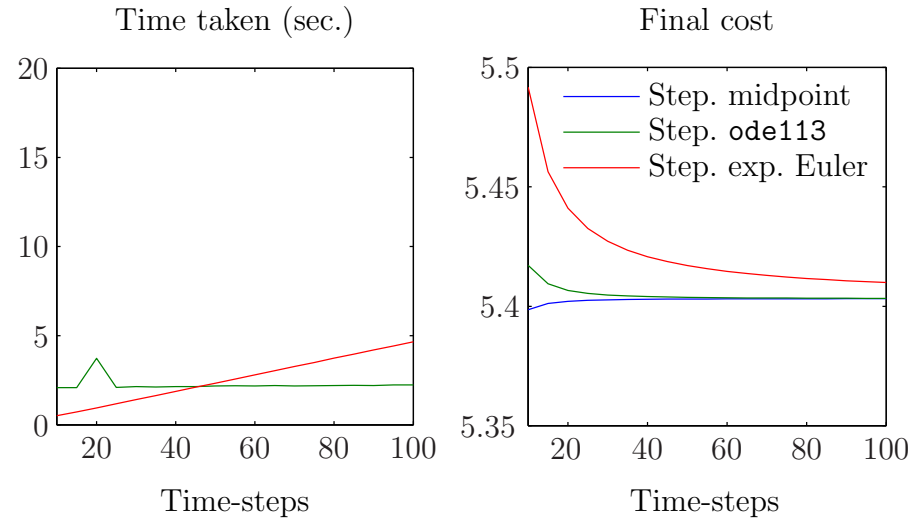


Figure 5.16: GIS: midpoint rule vs. `ode113` vs. explicit Euler ($n_c = 5$)

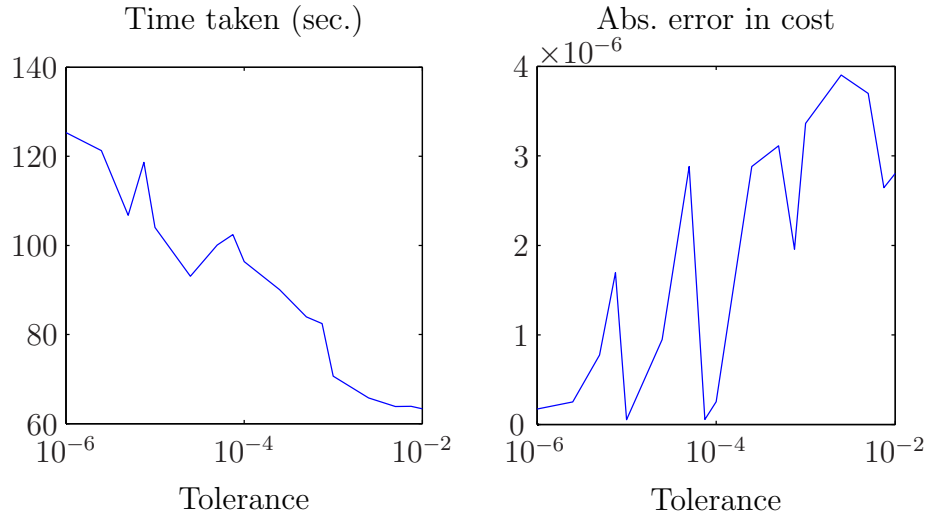


Figure 5.17: GIS: changing tolerance in `ode113` (real data)

stepping Euler solver is the only realistic option for dealing with real data. Figure 5.18 shows the optimal paths from '`ode113`'.

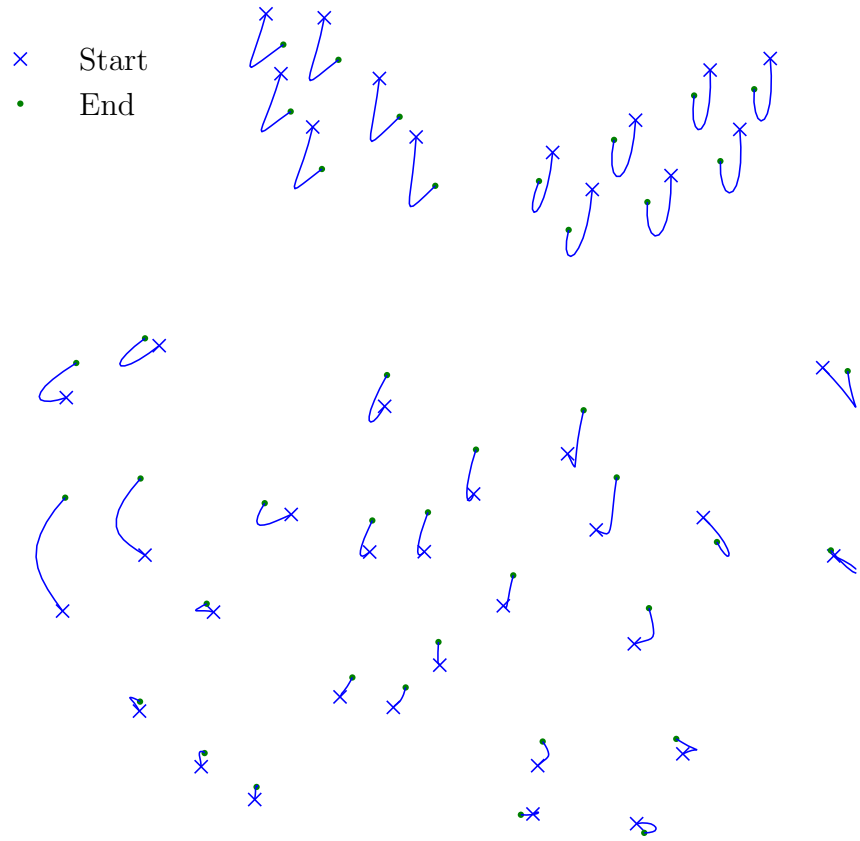


Figure 5.18: GIS: `ode113`, optimal paths (real data, magnified)

Finally, we have coded a practical user-interface for warping a chosen image. Using `ginput`, MATLAB collects the positions of the control points from the user's mouse-clicks. The optimal trajectories are computed using the stepping explicit Euler solver, and the resulting paths are used to generate the full warp using code adapted from Mills [14], the warp then being applied to the image. See Figure 1.1.

Chapter 6

Conclusions

Symplectic integrators preserve important geometric properties of the flow of Hamiltonian systems. Compared with the (non-symplectic) explicit Euler integrator, the (symplectic) implicit midpoint rule gives better performance and accuracy when applied to optimal control problems (where a mechanical system is to be moved under the influence of a force f in such a way that a cost functional J is minimized).

The distinction between symplectic and non-symplectic methods is less well-defined in the context of the more complicated GIS system in two dimensions. Since this is not, strictly speaking, an optimal control system, the problem is better solved by directly computing a solution to the discrete Hamilton's equations. This is best done by finding the initial momenta $\{\boldsymbol{p}_i(0)\}$ that generate final points $\{\boldsymbol{q}_{i,N+1}\}$ such that

$$\boldsymbol{q}_{i,N+1} = \boldsymbol{q}_i^1, \qquad \text{for } i = 1, \ldots, n_c \tag{6.0.1}$$

to some required tolerance, with $\{\boldsymbol{q}_i^1\}$ being the fixed final control-point positions. Doing this using an explicit Euler discretization of the dynamics results in a faster, less accurate method compared with an implicit midpoint discretization. The integrator `ode113` from MATLAB's ODE suite appears to have the speed of the explicit Euler method and the accuracy of the implicit midpoint rule. Symplectic Euler discretizations show little improvement over those using explicit Euler.

For a number of control points more akin to that in a real application of the GIS problem, the midpoint solver becomes impractical. The explicit Euler solver continues

to perform quite well, owing to it taking inexpensive explicit one-step updates, as opposed to the more costly implicit updates of the midpoint rule. The `ode113` solver is not faster than explicit Euler on the real data. More investigation into tweaking the ODE method's settings may reveal how its performance can be improved.

Care has gone into optimizing our code, and it would be interesting to see the performance of a further-optimized FORTRAN or C version. The major limitation for the best (that is, the explicit Euler stepping) method seems only to be availability of computing power.

Other topics of investigation following from the experiments herein are: can better initial paths than those analytically derived from the single particle example be generated practically; exactly how do the methods we have considered scale with the number of spatial dimensions; how many time-steps are needed to generate a warp having a given degree of smoothness; what is happening inside `fmincon` that causes the behaviour seen in Figures 5.8 and 5.9; can the midpoint stepping solve be made more efficient, so that we may benefit more from its accuracy; and if the midpoint stepper cannot be improved, how do higher order (explicit, symplectic) methods compare with the methods we used, a question we considered briefly with `ode113`.

Appendix A

Configuring a Group of Hovercraft

We demonstrate qualitatively how discrete mechanics can be used to solve a demanding control problem: we wish to optimally configure a group of hovercraft in the plane. This experiment is similar to one presented by Junge, Marsden & Ober–Blöbaum [9]. We consider only a midpoint rule discretization of the dynamics. The material is presented here because it is spiritually different from the experiments in Chapter 3, and so with this appendix the flow of that chapter is uninterrupted.

A.1 The Experiment

We have H identical craft, $H > 2$, which are to be moved from an initial configuration to a given final state with minimal effort. Each craft is described by the position in $\mathbb{R}^2$ of its centre of mass and by its orientation $\theta \in \mathbb{S}^1$, so that the configuration space is $Q = \mathbb{R}^2 \times \mathbb{S}^1$. Each is controlled by two forces applied at a distance r from the centre of mass. See Figure A.1 for the set-up. The particular initial data for us are: the separation r equals 0.2; the mass of each craft is $m = 1$; the moment of inertia of each craft is $I = 0.1$; the initial position in Q of craft i is $(1 + 2(i-1)/(H-1), 0, 0)$; and the initial velocities are uniformly zero.

The system is governed by the (kinetic energy) Lagrangian

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\dot{\boldsymbol{q}}^{\mathrm{T}} M \dot{\boldsymbol{q}},$$

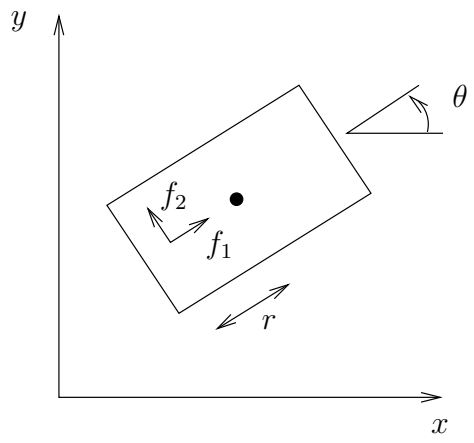


Figure A.1: Configuring hovercraft: a hovercraft

where M is the matrix $\mathrm{diag}(m, m, I)$. The forces f_1 and f_2 resolve in the x-, y- and θ-directions into

$$\boldsymbol{f}(t) = \begin{pmatrix} \cos\theta(t) f_1(t) - \sin\theta(t) f_2(t) \\ \sin\theta(t) f_1(t) + \cos\theta(t) f_2(t) \\ -r f_2(t) \end{pmatrix}.$$

If we write $(\boldsymbol{q}_i, f^i) = (x_i, y_i, \theta_i, f_1^i, f_2^i)$ for the full configuration of the ith craft, then as cost for this craft we use the expression

$$J_i(\boldsymbol{q}_i, f^i) = \int_0^1 (f_1^i(t))^2 + (f_2^i(t))^2 \,\mathrm{d}t.$$

The total cost for the group is the sum of the J_is.

For the dynamical constraints, using the midpoint rule we know that the discrete Euler–Lagrange equations for craft i read

$$M\left(\frac{\boldsymbol{q}_{i,k+1} - 2\boldsymbol{q}_{i,k} + \boldsymbol{q}_{i,k-1}}{h^2}\right) + \frac{1}{4}(\boldsymbol{f}_{i,k-1} + 2\boldsymbol{f}_{i,k} + \boldsymbol{f}_{i,k+1}) = 0, \qquad \text{for } k = 1, \ldots, N-1,$$

where $\boldsymbol{f}_{i,k} \stackrel{\text{def}}{=} \boldsymbol{f}_i(kh)$ is the force $\boldsymbol{f}$ for the ith craft at time kh.

We wish the final configuration to be a regular H-gon, as follows: each craft is $R = 1$ units away from the others, so that

$$(x_i(1) - x_{i+1}(1))^2 + (y_i(1) - y_{i+1}(1))^2 = R^2, \qquad \text{for } i = 1, \ldots, H-1,$$
$$(x_1(1) - x_H(1))^2 + (y_1(1) - y_H(1))^2 = R^2;$$

the centre of the final configuration is $(M_x, M_y) = (2, 2)$, so

$$\frac{1}{H}\sum_{i=1}^{H}\big(x_i(1), y_i(1)\big) = (M_x, M_y);$$

to eliminate the final rotational degree of freedom, we constrain the position of craft 1 by

$$y_1(1) = M_y;$$

and when $H > 3$ we have constraints on the interior angles of the H-gon

$$\text{interior angle at vertex } i = \frac{\pi(H-2)}{H}, \qquad \text{for } i = 1, \ldots, H.$$

As final conditions for each individual craft, the final orientation is fixed at $\pi/2$ and the final velocity is zero.

The intention is to minimize the discretized total cost, subject to the discrete equations of motion and to attaining the final configuration. Since we use the midpoint rule, this is similar to what we did with 'Mid.' in Section 3.3. In terms of our implementation, the most important thing to point out is how we compute the analytic derivatives. Upon coding the cost and constraints, we create modified versions of these functions that take symbolic input, and then we use `jac` to evaluate their analytic Jacobians. By invoking the function `sym2fun` (Appendix A.2.8) we convert these symbolic arrays into function M-files, which can then be called whenever the derivatives are required.

One other thing to highlight is the generation of an initial guess. A little elementary geometry gives that the radius of the circle inscribed by the final H-gon is $R/(2\sin(\pi/H))$. We set the guess for $y_1(1)$ to be the point π radians around this circle, and then we add $H-1$ additional evenly spaced points on the perimeter. Then we linearly interpolate between the initial and final points in Q for each craft, and we set the initial forces uniformly equal to 1. We show the initial paths for $H = 3$ and 5 in Figures A.2 and A.3, respectively. The final circle on which the craft are to lie is given in black and the final positions by black squares, while the red arrows serve to show the orientations of the craft along the way.

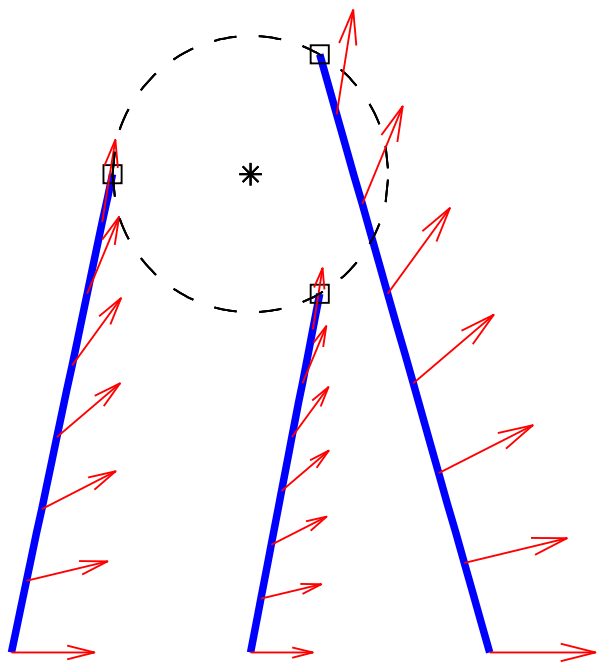


Figure A.2: Configuring hovercraft: initial guess (3 craft)

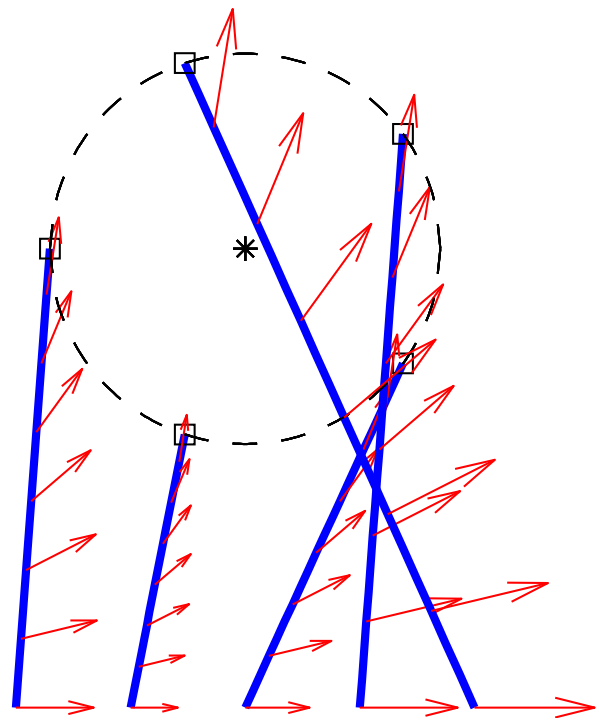


Figure A.3: Configuring hovercraft: initial guess (5 craft)

Running the code for 20 time-steps, we obtain the trajectories in Figures A.4 and A.5.

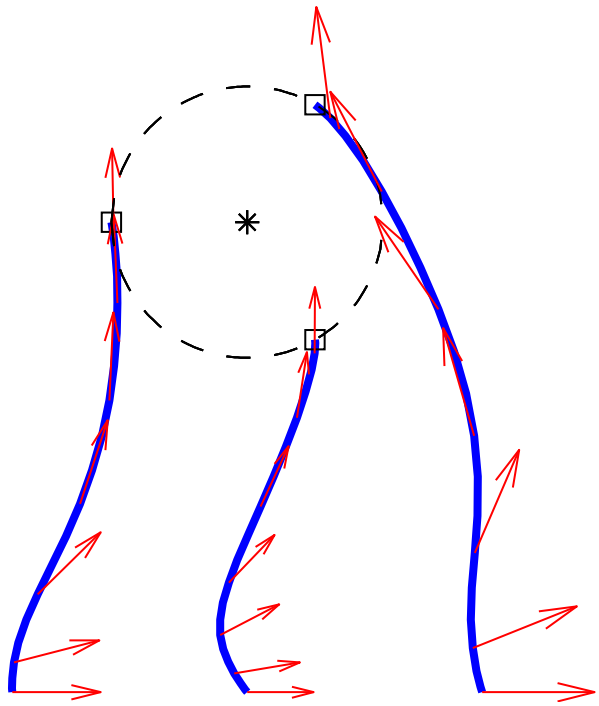


Figure A.4: Configuring hovercraft: optimal configuration (3 craft)

It is important to note that, if we allow the rotational degree of freedom in the initial guess and constraints, the problem becomes one of global optimization. This is because a solution then depends on its initial guess [9].

A.2 Code

A.2.1 Dependency Tree

```
                          ┌─ hovcost ─── Gradfun
              ┌─ mkhovjac ─┤
              │            ├─ symjac
              │            └─ sym2fun
hovbatch ─────┤
              ├─ hover
              ├─ hovcost ─── Gradfun
              │
              └─ hovnlcon ─── Jacfun
```

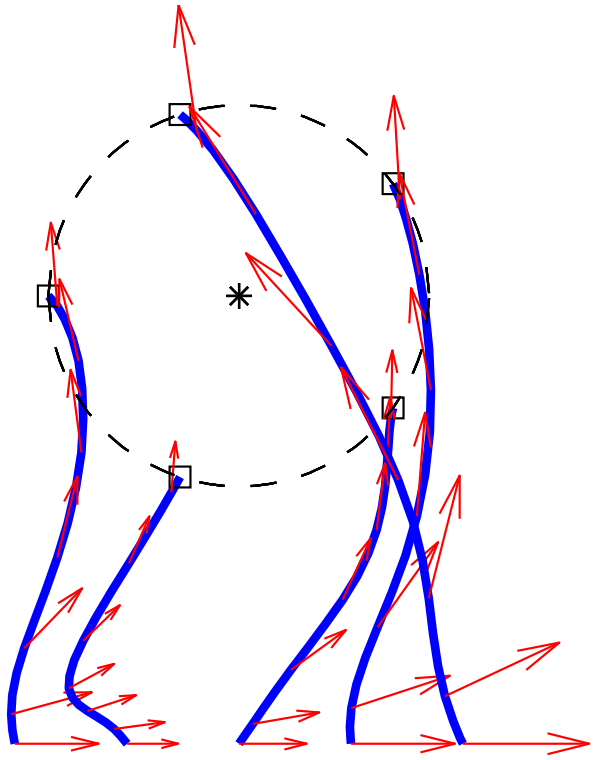


Figure A.5: Configuring hovercraft: optimal configuration (5 craft)

```
hovplot
```

(The files `Gradfun.m` and `Jacfun.m` are dynamically generated at runtime, so are not listed).

A.2.2 `hovbatch.m`

```
%HOVBATCH Batch file for hovercraft problem.
% 15/08/05.

datestring = datestr(now);
datestring(datestring == ' ') = [];
datestring(datestring == '-') = [];
datestring(datestring == ':') = [];

hovrange = [3,5];
save thedate datestring hovrange
```

```
for hov = hovrange
    N = 20; % Time-steps.
    argstr = mkhovjac(hov,N); % Make M-files for Jacobians.

    % Initial data:
    h = 1/N;
    r = repmat(0.2,hov,1); % Distance of forces from COM.
    R = 1; % Final separation.
    [Mx, My] = deal(2); % Final centre.
    m = 1; % Craft mass.
    I = 0.1; % Craft moment of inertia.
    M = diag([m, m, I]); % Mass matrix.
    % Initial configs. (straight line):
    X0 = linspace(1,3,hov).'; Y0 = zeros(hov,1);
    theta0 = zeros(hov,1);
    theta1 = repmat(pi/2,hov,1); % Fixed final orientations.

    % Fixed boundary velocities.
    for i = 1:hov
        qdot0(:,i) = zeros(3,1); qdot1(:,i) = zeros(3,1);
    end

    [swapstr,Aeq,beq,x0] = ...
        hover(argstr,hov,Mx,My,N,R,X0,Y0,theta0,theta1);

    options = optimset('Disp', 'iter', ...
        'GradConstr', 'on', ...
        'GradObj', 'on', ...
        'LargeScale', 'off');

    x = fmincon(@(x)hovcost(x,argstr,hov,N,swapstr), ...
        x0,[],[],Aeq,beq,[],[], ...
        @(x)hovnlcon(x,argstr,hov,M,Mx,My,N,qdot0,qdot1,r,R,swapstr),...
        options);

    save(sprintf('hovvars%d%s',hov,datestring))
```

```
end
exit
```

A.2.3 hovcost.m

```
function [J,G] = hovcost(x,argstr,hov,N,swapstr)
%HOVCOST Cost function for hovercraft problem.
% INPUT
%   X Vector [q1 f1 q2 f2 ...].
%   ARGSTR comma-separated argument-list string.
%   HOV Number of craft.
%   N Number of time-steps.
%   SWAPSTR Assigns x(i) to xi.
% OUTPUT
%   J Cost at X.
%   G Gradient of cost at X.
% 12/08/05.

J = 0; h = 1/N;

for i = 1:hov
    J = J + h/4*sum((x((5*(i-1)+3)*(N+1)+(1:N)) ...
        + x((5*(i-1)+3)*(N+1)+(2:N+1))).^2) ...
        +h/4*sum((x((5*(i-1)+4)*(N+1)+(1:N)) ...
        + x((5*(i-1)+4)*(N+1)+(2:N+1))).^2);
end

if nargout > 1
   % For compatibility with Gradfun, put x(j) -> xj:
   eval(swapstr)

   % Evaluate Jacobian:
   G = zeros(5*hov*(N+1),1);
   eval(sprintf('G = Gradfun(%s)'';', argstr));
end
```

A.2.4 hover.m

```
function [swapstr,Aeq,beq,x0] = ...
    hover(argstr,hov,Mx,My,N,R,X0,Y0,theta0,theta1)
%HOVER Initializes variables for hovercraft optimization problem.
% INPUT
%   ARGSTR comma-separated argument-list string.
%   HOV Number of craft.
%   MX x-coordinate of centre of final configuration.
%   MY y-coordinate of centre of final configuration.
%   N Number of time-steps.
%   R Final separation.
%   X0 Craft initial xs.
%   Y0 Craft initial ys.
%   THETA0 Craft initial thetas.
%   THETA1 Craft final thetas.
% OUTPUT
%   SWAPSTR String for later evaluation, assigns x(i) to xi,
%           particularly for use in evaluation of Jacobians.
%   AEQ Linear equality constraints matrix.
%   BEQ Equality constraints vector: AEQ*X0 = BEQ.
%   X0 Initial guess for FMINCON.
% 02/09/05.

Aeq = zeros(4*hov,5*hov*(N+1));
% Pick out correct indices. Concatenation of 4 arrays:
% 1: ith entry is x_0^i.
% 2: ith entry is y_0^i.
% 3: ith entry is theta_0^i.
% 4: ith entry is theta_1^i.
Aeq([4*hov*(0:5*(N+1):5*(hov-1)*(N+1)) + (1:3:3*hov-2), ...
        4*hov*(N+1:5*(N+1):(5*(hov-1)+1)*(N+1)) + (2:3:3*hov-1), ...
        4*hov*(2*(N+1):5*(N+1):(5*(hov-1)+2)*(N+1)) + (3:3:3*hov), ...
        4*hov*(3*(N+1)-1:5*(N+1):(5*(hov-1)+3)*(N+1)-1) ...
        + (3*hov+1:4*hov)]) = 1;
```

```
beq = zeros(4*hov,1);
beq(1:3:3*hov-2) = X0; beq(2:3:3*hov-1) = Y0; beq(3:3:3*hov) = theta0;
beq(3*hov+1:3*hov+hov) = theta1;

% Initial guess for fmincon.
x0 = zeros(5*hov*(N+1),1);
x0(1:5*(N+1):5*(hov-1)*(N+1)+1) = X0;
x0(N+1+1:5*(N+1):(5*(hov-1)+1)*(N+1)+1) = Y0;
x0(2*(N+1)+1:5*(N+1):(5*(hov-1)+2)*(N+1)+1) = theta0;

% Final config. Circumradius of final hov-gon is R/(2*sin(pi/hov)):
rad = R/(2*sin(pi/hov)); r = repmat(rad,hov,1);
th = pi; theta = [];
for i = 1:hov
    theta = [theta; th+(i-1)*2*pi/hov];
end

q1 = zeros(2,hov);
[q1(1,:), q1(2,:)] = pol2cart(theta,r);
finalx = repmat(Mx,hov,1)+q1(1,:).';
finaly = repmat(My,hov,1)+q1(2,:).';

xstep = (finalx - X0)/N; ystep = (finaly - Y0)/N;
thetastep = (theta1 - theta0)/N;

for i = 1:hov
    x0(5*(i-1)*(N+1)+(1:N+1)) = ...
        X0(i) + cumsum([0 repmat(xstep(i),1,N)]);
    x0((5*(i-1)+1)*(N+1)+(1:N+1)) = ...
        Y0(i) + cumsum([0 repmat(ystep(i),1,N)]);
    x0((5*(i-1)+2)*(N+1)+(1:N+1)) = ...
        theta0(i) + cumsum([0 repmat(thetastep(i),1,N)]);
    x0((5*(i-1)+3)*(N+1)+(1:N+1)) = repmat(1,N+1,1);
    x0((5*(i-1)+4)*(N+1)+(1:N+1)) = repmat(1,N+1,1);
end
```

```
% swapstr is used in HOVERNLCON to assign arguments to Jacobian
% function file.
swapstr = sprintf('x(%d),', 1:5*hov*(N+1)); swapstr(end) = [];
swapstr = sprintf('[%s] = deal(%s);', argstr, swapstr);
```

A.2.5 hovnlcon.m

```
function [C,Ceq,GC,GCeq] = ...
    hovnlcon(x,argstr,hov,M,Mx,My,N,qdot0,qdot1,r,R,swapstr)
%HOVNLCON Nonlinear constraints for hovercraft optimization problem.
% INPUT:
%   X Vector [q1 f1 q2 f2 ...].
%   ARGSTR comma-separated argument-list string.
%   HOV Number of craft.
%   M Mass matrix.
%   MX x-coordinate of centre of final configuration.
%   MY y-coordinate of centre of final configuration.
%   N Number of time-steps.
%   QDOT0 Initial velocities.
%   QDOT1 Final velocities.
%   r Distance of forces from COM.
%   R Final separation.
%   SWAPSTR Assigns x(i) to xi.
% 02/09/05.

% Unpack x:
[ex,y,theta,f1,f2] = unpack(x,hov,N);

C = [];
if hov == 3
    Ceq = zeros(3*hov*(N+1)+hov+2,1);
else
    Ceq = zeros(3*hov*(N+1)+2*hov+3,1);
end
h = 1/N;
```

```
for i = 1:hov
    q = [ex(i,1:N+1); y(i,1:N+1); theta(i,1:N+1)];
    f = h*[cos(theta(i,2:N+1)).*f1(i,2:N+1) ...
        - sin(theta(i,2:N+1)).*f2(i,2:N+1); ...
        sin(theta(i,2:N+1)).*f1(i,2:N+1) ...
        - cos(theta(i,2:N+1)).*f2(i,2:N+1); - r(i)*f2(i,2:N+1)]/4 ...
        + h*[cos(theta(i,1:N)).*f1(i,1:N) ...
        - sin(theta(i,1:N)).*f2(i,1:N); ...
        sin(theta(i,1:N)).*f1(i,1:N) ...
        - cos(theta(i,1:N)).*f2(i,1:N); - r(i)*f2(i,1:N)]/4;

    Ceq(3*(N+1)*(i-1) + (1:3)) = ...
        M*qdot0(:,i) - M*(q(:,2) - q(:,1))/h + f(:,1);

    % Discrete E-L eq'ns:
    % D2L(q,qdot) = M*qdot,
    % D1Ld = -D2Ld(qk,qkp) = M*(qkp - qk)/h:
    Ceq(3*(N+1)*(i-1) + 3 + (1:3*(N-1))) = ...
        reshape(M*(q(:,2:N) - q(:,1:N-1))/h - ...
        M*(q(:,3:N+1) - q(:,2:N))/h + f(:,1:N-1) + f(:,2:N),3*(N-1),1);

    Ceq(3*(N+1)*(i-1) + 3 + 3*(N-1) + (1:3)) = ...
        -M*qdot1(:,i) + M*(q(:,N+1) - q(:,N))/h + f(:,N);
end

% Craft final positions:
for i = 1:hov-1
    Ceq(3*hov*(N+1)+i) = ...
        (ex(i,N+1) - ex(i+1,N+1))^2 + (y(i,N+1) - y(i+1,N+1))^2 - R^2;
end
Ceq(3*hov*(N+1)+hov) = ...
    (ex(1,N+1) - ex(hov,N+1))^2 + (y(1,N+1) - y(hov,N+1))^2 - R^2;

% Craft final centre:
Ceq(3*hov*(N+1)+hov+1) = sum(ex(:,N+1))/hov - Mx;
Ceq(3*hov*(N+1)+hov+2) = sum(y(:,N+1))/hov - My;
```

```
% Final orientation of hov-gon:
Ceq(3*hov*(N+1)+hov+3) = y(1,N+1) - My;

% Constraints on interior angles (N/A for triangle):
if hov ~= 3
    v1 = [ex(2,N+1); y(2,N+1)] - [ex(1,N+1); y(1,N+1)];
    v2 = [ex(hov,N+1); y(hov,N+1)] - [ex(1,N+1); y(1,N+1)];
    n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
    Ceq(3*hov*(N+1)+hov+4) = acos(d/(n1*n2)) - (pi*(hov-2)/hov);
    for i = 2:hov-1
        v1 = [ex(i-1,N+1); y(i-1,N+1)] - [ex(i,N+1); y(i,N+1)];
        v2 = [ex(i+1,N+1); y(i+1,N+1)] - [ex(i,N+1); y(i,N+1)];
        n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
        Ceq(3*hov*(N+1)+hov+3+i) = acos(d/(n1*n2)) - (pi*(hov-2)/hov);
    end
    v1 = [ex(hov-1,N+1); y(hov-1,N+1)] - [ex(hov,N+1); y(hov,N+1)];
    v2 = [ex(1,N+1); y(1,N+1)] - [ex(hov,N+1); y(hov,N+1)];
    n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
    Ceq(3*hov*(N+1)+2*hov+3) = acos(d/(n1*n2)) - (pi*(hov-2)/hov);
end

if nargout > 2
    GC = [];

    % For compatibility with Jacfun, put x(j) -> xj:
    eval(swapstr)

    % Evaluate Jacobian:
    if hov == 3
        GCeq = zeros(5*hov*(N+1),3*hov*(N+1)+hov+3);
    else
        GCeq = zeros(5*hov*(N+1),3*hov*(N+1)+2*hov+3);
    end
    eval(sprintf('GCeq = Jacfun(%s)'';', argstr));
end
```

```
%==========================================================================
% Unpacking subfunction.
function [ex,y,theta,f1,f2] = unpack(x,hov,N)
%UNPACK Splits X into component parts.
%   Number of pairs (q_i,f_i) is equal to HOV.

ex= zeros(hov,N+1); y = zeros(hov,N+1); theta = zeros(hov,N+1);
f1 = zeros(hov,N+1); f2 = zeros(hov,N+1);

for i = 1:hov
        ex(i,:) = x(5*(i-1)*(N+1)+(1:N+1));
        y(i,:) = x((5*(i-1)+1)*(N+1)+(1:N+1));
        theta(i,:) = x((5*(i-1)+2)*(N+1)+(1:N+1));
        f1(i,:) = x((5*(i-1)+3)*(N+1)+(1:N+1));
        f2(i,:) = x((5*(i-1)+4)*(N+1)+(1:N+1));
end
```

A.2.6 hovplot.m

```
function hovplot(lap)
%HOVPLOT Plots for hovercraft results.
% INPUT
%   LAP String 'on' or 'off' for exporting the figures using LAPRINT[1].
% 15/08/05.

if nargin < 1, lap = 'off'; end

load thedate, dirstr = 'C:\';

for hov = hovrange
    load(sprintf('hovvars%d%s',hov,datestring))
    rad = R/(2*sin(pi/hov));
    % Initial trajectories: ==========================================
    figure
```

[1] `laprint.m` is available from the MATLAB Central File Exchange.

```
for i = 1:hov
    % Trajectories:
    plot(x0(5*(i-1)*(N+1)+1:5*(i-1)*(N+1)+N+1), ...
        x0((5*(i-1)+1)*(N+1)+1:(5*(i-1)+1)*(N+1)+N+1), ...
        'b','LineWidth',2)
    xlim([0.5 3.5]), xlabel('$x$'), ylabel('$y$')
    axis equal, axis off
    box off, hold on

    % Centre of final config.:
    plot(Mx,My,'k*')

    % Final circle:
    X = linspace(Mx-rad,Mx+rad);
    plot(X,My+sqrt(rad^2-(X-Mx).^2),'k--')
    plot(X,My-sqrt(rad^2-(X-Mx).^2),'k--')

    % Final positions:
    plot(x0(5*(i-1)*(N+1)+N+1),x0((5*(i-1)+1)*(N+1)+N+1),'ks')

    % Quiver plot showing final orientations:
    quiver(x0(5*(i-1)*(N+1)+1:3:5*(i-1)*(N+1)+N+1), ...
        x0((5*(i-1)+1)*(N+1)+1:3:(5*(i-1)+1)*(N+1)+N+1), ...
        cos(x0((5*(i-1)+2)*(N+1)+1:3:(5*(i-1)+2)*(N+1)+N+1)), ...
        sin(x0((5*(i-1)+2)*(N+1)+1:3:(5*(i-1)+2)*(N+1)+N+1)), ...
        0.5,'r');
    hold on
end
hold off

if strcmp(lap,'on'),
    laprint(gcf,sprintf('%s%dhovercraftinit',dirstr,hov), ...
        'scalefonts','off','width',8);
end

% Optimal trajectories: ==========================================
```

```
    figure
    for i = 1:hov
        % Trajectories:
        plot(x(5*(i-1)*(N+1)+1:5*(i-1)*(N+1)+N+1), ...
            x((5*(i-1)+1)*(N+1)+1:(5*(i-1)+1)*(N+1)+N+1), ...
            'b','LineWidth',2)
        xlim([0.5 3.5]), xlabel('$x$'), ylabel('$y$')
        axis equal, axis off
        box off, hold on

        % Centre of final config.:
        plot(Mx,My,'k*')

        % Final circle:
        X = linspace(Mx-rad,Mx+rad);
        plot(X,My+sqrt(rad^2-(X-Mx).^2),'k--')
        plot(X,My-sqrt(rad^2-(X-Mx).^2),'k--')

        % Final positions:
        plot(x(5*(i-1)*(N+1)+N+1),x((5*(i-1)+1)*(N+1)+N+1),'ks')

        % Quiver plot showing final orientations:
        quiver(x(5*(i-1)*(N+1)+1:3:5*(i-1)*(N+1)+N+1), ...
            x((5*(i-1)+1)*(N+1)+1:3:(5*(i-1)+1)*(N+1)+N+1), ...
            cos(x((5*(i-1)+2)*(N+1)+1:3:(5*(i-1)+2)*(N+1)+N+1)), ...
            sin(x((5*(i-1)+2)*(N+1)+1:3:(5*(i-1)+2)*(N+1)+N+1)), ...
            0.5,'r');
        hold on
    end
    hold off

    if strcmp(lap,'on'),
        laprint(gcf,sprintf('%s%dhovercraftfin',dirstr,hov), ...
            'scalefonts','off','width',8);
    end
end
```

A.2.7 mkhovjac.m

```
function argstr = mkhovjac(hov,N)
%MKHOVJAC Constructs function M-files for hovercraft Jacobians.
% ARGSTR = MKHOVJAC(HOV,N) uses SYM2FUN to generate function
% files for Jacobian of hovercraft cost function and of constraints.
% INPUT
%   HOV Number of craft. Defaults to HOV = 3.
%   N Number of time-steps. Defaults to N = 20.
% OUTPUT
%   ARGSTR Argument-list string
%         ARGSTR = 'x1,x2,...'
%         Required for Jacobian functions.
% 02/09/05.

if nargin < 1
    hov = 3; N = 20;
end

% Initial data:
r = repmat(0.2,hov,1); % Distance of forces from COM.
R = 1; % Final separation.
[Mx, My] = deal(2); % Final centre.
m = 1; % Craft mass.
I = 0.1; % Craft moment of inertia.
M = diag([m, m, I]); % Mass matrix.

% Fixed boundary velocities.
for i = 1:hov
    qdot0(:,i) = zeros(3,1); qdot1(:,i) = zeros(3,1);
end

% Argument-list string:
argstr = sprintf('x%d,', 1:5*hov*(N+1));
% Remove trailing comma:
argstr(end) = [];
```

```
% String for symbolic initialization: varstr = 'x1 x2 ... '.
varstr = argstr; varstr(varstr == ',') = ' ';

eval(sprintf('syms %s', varstr));

% swapstr is used in HOVNLCON to assign arguments to Jacobian
% function file.
swapstr = sprintf('x(%d),', 1:5*hov*(N+1)); swapstr(end) = [];
swapstr = sprintf('[%s] = deal(%s);', argstr, swapstr);

% Symbolic Jacobians:
x = eval(sprintf('[%s];', varstr));
J = hovcost(x,argstr,hov,N,swapstr);
v = eval(sprintf('[%s];', varstr));
Grad = jacobian(J,v);
Jac = symjac(hov,M,Mx,My,N,qdot0,qdot1,r,R);
% Convert to function M-files:
sym2fun(Grad,'Gradfun',argstr); sym2fun(Jac,'Jacfun',argstr);
rehash % Update path to reveal new M-files.
```

A.2.8 sym2fun.m

```
function sym2fun(A,str,argstr)
%SYM2FUN Converts symbolic array A into function M-file STR.M
% INPUT
%   A Symbolic array.
%   STR Output filename string.
%   ARGSTR comma-separated argument-list string.
% 12/08/05.

% Create a file str.m:
fid = fopen([str '.m'], 'w');

% Create the function header:
fprintf(fid, 'function out = %s(%s)\n\n', str, argstr);
```

```
% Create the output:
fprintf(fid, 'out = [');
sA = size(A);
for row = 1:sA(1)
    rowstr = char(A(row,:));

    % Remove unwanted additions:
    rowstr(1:9) = []; rowstr(end-2:end) = [];

    fprintf(fid, '%s; ...\n ', rowstr);
end
fprintf(fid, ']; '); fclose(fid);
```

A.2.9 symjac.m

```
function Jac = symjac(hov,M,Mx,My,N,qdot0,qdot1,r,R)
%SYMJAC Symbolic Jacobian of hovercraft constraints.
% INPUT
%   HOV Number of craft.
%   M Mass matrix.
%   MX x-coordinate of centre of final configuration.
%   MY y-coordinate of centre of final configuration.
%   N Number of time-steps.
%   QDOT0 Initial velocities.
%   QDOT1 Final velocities.
%   r Distance of forces from COM.
%   R Final separation.
% OUTPUT
%   JAC Jacobian of constraints.
% 15/08/05.

% String for symbolic initialization: varstr = 'x1 x2 ... '.
varstr = sprintf('x%d ', 1:5*hov*(N+1)); varstr(end) = [];
eval(sprintf('syms %s', varstr));
```

```
h = 1/N;

for i = 1:hov
    eval(sprintf('qoldold = [x%d; x%d; x%d]; ',5*(i-1)*(N+1) + 1, ...
        (5*(i-1)+1)*(N+1) + 1,(5*(i-1)+2)*(N+1) + 1));
    eval(sprintf('qold = [x%d; x%d; x%d]; ',5*(i-1)*(N+1) + 2, ...
        (5*(i-1)+1)*(N+1) + 2, (5*(i-1)+2)*(N+1) + 2));
    eval(sprintf(['fold = h*[cos(x%d)*x%d - sin(x%d)*x%d; ', ...
        'sin(x%d)*x%d - cos(x%d)*x%d; ','-r(%d)*x%d]/4 ', ...
        '+ h*[cos(x%d)*x%d - sin(x%d)*x%d; ', ...
        'sin(x%d)*x%d - cos(x%d)*x%d; ','-r(%d)*x%d]/4; '], ...
        (5*(i-1)+2)*(N+1) + 2, (5*(i-1)+3)*(N+1) + 2, ...
        (5*(i-1)+2)*(N+1) + 2, (5*(i-1)+4)*(N+1) + 2, ...
        (5*(i-1)+2)*(N+1) + 2, (5*(i-1)+3)*(N+1) + 2, ...
        (5*(i-1)+2)*(N+1) + 2, (5*(i-1)+4)*(N+1) + 2, ...
        i, (5*(i-1)+4)*(N+1) + 2,  ...
        (5*(i-1)+2)*(N+1) + 1, (5*(i-1)+3)*(N+1) + 1, ...
        (5*(i-1)+2)*(N+1) + 1, (5*(i-1)+4)*(N+1) + 1, ...
        (5*(i-1)+2)*(N+1) + 1, (5*(i-1)+3)*(N+1) + 1, ...
        (5*(i-1)+2)*(N+1) + 1, (5*(i-1)+4)*(N+1) + 1, ...
        i, (5*(i-1)+4)*(N+1) + 1));

    Ceq(3*(N+1)*(i-1) + (1:3)) = ...
        M*qdot0(:,i) - M*(qold - qoldold)/h + fold;

    for k = 2:N
        eval(sprintf('qnew = [x%d; x%d; x%d]; ',5*(i-1)*(N+1) + k+1, ...
            (5*(i-1)+1)*(N+1) + k+1, (5*(i-1)+2)*(N+1) + k+1));
        eval(sprintf(['fnew = h*[cos(x%d)*x%d - sin(x%d)*x%d; ', ...
        'sin(x%d)*x%d - cos(x%d)*x%d; ','-r(%d)*x%d]/4 ', ...
        '+ h*[cos(x%d)*x%d - sin(x%d)*x%d; ', ...
        'sin(x%d)*x%d - cos(x%d)*x%d; ','-r(%d)*x%d]/4; '], ...
        (5*(i-1)+2)*(N+1) + k+1, (5*(i-1)+3)*(N+1) + k+1, ...
        (5*(i-1)+2)*(N+1) + k+1, (5*(i-1)+4)*(N+1) + k+1, ...
        (5*(i-1)+2)*(N+1) + k+1, (5*(i-1)+3)*(N+1) + k+1, ...
        (5*(i-1)+2)*(N+1) + k+1, (5*(i-1)+4)*(N+1) + k+1, ...
```

```
        i, (5*(i-1)+4)*(N+1) + k+1,   ...
        (5*(i-1)+2)*(N+1) + k, (5*(i-1)+3)*(N+1) + k, ...
        (5*(i-1)+2)*(N+1) + k, (5*(i-1)+4)*(N+1) + k, ...
        (5*(i-1)+2)*(N+1) + k, (5*(i-1)+3)*(N+1) + k, ...
        (5*(i-1)+2)*(N+1) + k, (5*(i-1)+4)*(N+1) + k, ...
        i, (5*(i-1)+4)*(N+1) + k));

        % Discrete E-L eq'ns:
        Ceq(3*(N+1)*(i-1) + 3*(k-1) + (1:3)) ...
            = M*(qold - qoldold)/h - M*(qnew - qold)/h + fold + fnew;

        qoldold = qold; qold = qnew; fold = fnew;
    end
    Ceq(3*(N+1)*(i-1)+3+3*(N-1)+(1:3)) = ...
        -M*qdot1(:,i) + M*(qold - qoldold)/h + fold;
end

% Craft final positions:
for i = 1:hov-1
    eval(sprintf( ...
    'Ceq(3*hov*(N+1)+%d) = (x%d - x%d)^2 + (x%d - x%d)^2 - R^2;', ...
    i,(5*(i-1)+1)*(N+1),(5*i+1)*(N+1),(5*(i-1)+2)*(N+1),(5*i+2)*(N+1)));
end
eval(sprintf( ...
    'Ceq(3*hov*(N+1)+hov) = (x%d - x%d)^2 + (x%d - x%d)^2 - R^2;', ...
    (N+1), (5*(hov-1)+1)*(N+1), 2*(N+1), (5*(hov-1)+2)*(N+1)));

% Craft final centre:
str1 = sprintf('x%d +',N+1); str2 = sprintf('x%d +',2*(N+1));
for i = 2:hov
    str1 = sprintf('%s x%d +',str1,(5*(i-1)+1)*(N+1));
    str2 = sprintf('%s x%d +',str2,(5*(i-1)+2)*(N+1));
end
str1(end) = []; str2(end) = []; str1(end) = []; str2(end) = [];
eval(sprintf('Ceq(3*hov*(N+1)+hov+1) = (%s)/hov - Mx;',str1));
eval(sprintf('Ceq(3*hov*(N+1)+hov+2) = (%s)/hov - My;',str2));
```

```
eval(sprintf('Ceq(3*hov*(N+1)+hov+3) = x%d - My;',2*(N+1)));

if hov ~= 3
    eval(sprintf( ...
        'v1 = [x%d - x%d; x%d - x%d];', ...
        6*(N+1),(N+1),7*(N+1),2*(N+1)));
    eval(sprintf( ...
        'v2 = [x%d - x%d; x%d - x%d];', ...
        (5*(hov-1)+1)*(N+1),(N+1),(5*(hov-1)+2)*(N+1),2*(N+1)));
    n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
    eval(sprintf( ...
        'Ceq(3*hov*(N+1)+hov+4) = acos(d/(n1*n2))-(pi*(hov-2)/hov);'));
    for i = 2:hov-1
        eval(sprintf( ...
            'v1 = [x%d - x%d; x%d - x%d];', ...
            (5*(i-2)+1)*(N+1),(5*(i-1)+1)*(N+1), ...
            (5*(i-2)+2)*(N+1),(5*(i-1)+2)*(N+1)));
        eval(sprintf( ...
            'v2 = [x%d - x%d; x%d - x%d];', ...
            (5*i+1)*(N+1),(5*(i-1)+1)*(N+1), ...
            (5*i+2)*(N+1),(5*(i-1)+2)*(N+1)));
        n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
        eval(sprintf( ...
            ['Ceq(3*hov*(N+1)+hov+3+%d) = ' ...
            'acos(d/(n1*n2)) - (pi*(hov-2)/hov);'],i));
    end
    eval(sprintf( ...
        'v1 = [x%d - x%d; x%d - x%d];', ...
        (5*(hov-2)+1)*(N+1),(5*(hov-1)+1)*(N+1), ...
        (5*(hov-2)+2)*(N+1),(5*(hov-1)+2)*(N+1)));
    eval(sprintf( ...
        'v2 = [x%d - x%d; x%d - x%d];', ...
        (N+1),(5*(hov-1)+1)*(N+1),2*(N+1),(5*(hov-1)+2)*(N+1)));
    n1 = sum(v1.^2); n2 = sum(v2.^2); d = sum(v1.*v2);
    eval(sprintf( ...
```

```
        ['Ceq(3*hov*(N+1)+2*hov+3) = ' ...
        'acos(d/(n1*n2)) - (pi*(hov-2)/hov);']));
end

% Variables to differentiate with respect to:
v = zeros(5*hov*(N+1),1); eval(sprintf('v = [%s];', varstr));

Jac = jacobian(Ceq,v);
```

Appendix B

The Euler–Poincaré Equations

We saw in Chapter 4 that control point registration can be thought of as a particle-motion problem on the diffeomorphism group $\mathrm{Diff}(\mathbb{D})$. The material in this appendix serves to explain this a little further. The reason that this discussion appears here and not in the main body is its dependence on some concepts in differential geometry. There is no space in this thesis to adequately introduce these topics, but a good reference for notation and the theory is Lee's *Introduction to Smooth Manifolds* [10]. Section B.1 concerns the derivation of a PDE that governs our n_c-particle dynamics. This section is completely optional in the context of the rest of the thesis, and the presentation is inspired by the article [7] of Holm. In Section B.2, where the material is less dependent on external theory, we use the PDE from Section B.1 to justify the particle ansatz (4.1.5).

B.1 Derivation

The Euler–Lagrange equations of motion on $\mathcal{D} \stackrel{\text{def}}{=} \mathrm{Diff}(\mathbb{D})$ are equivalent to the *Euler–Poincaré equations* [11], which are (B.1.1), below. Let $L : T\mathcal{D} \to \mathbb{R}$ be a left-invariant Lagrangian and $\ell : \mathfrak{g}(\mathcal{D}) \to \mathbb{R}$ its restriction to the Lie algebra $\mathfrak{g}(\mathcal{D})$. As a vector space, $\mathfrak{g}(\mathcal{D})$ is the space of vector fields on $\mathcal{D}$ [11]. Its dual space is the space of *one-form densities* $\mathfrak{g}(\mathcal{D})^* = \Lambda^1(\mathcal{D}) \otimes \mathrm{Den}(\mathcal{D})$, in the sense that if $\boldsymbol{m} = \boldsymbol{\omega} \otimes \mu \in \mathfrak{g}(\mathcal{D})^*$

then, for $\boldsymbol{v} \in \mathfrak{g}(\mathcal{D})$,

$$\langle \boldsymbol{\omega} \otimes \mu, \boldsymbol{v} \rangle_{\mathfrak{g}(\mathcal{D})} = \int_{\mathcal{D}} \boldsymbol{v} \lrcorner \boldsymbol{\omega}\, \mu,$$

where $\lrcorner$ is contraction of a one-form with a vector field $\big(\boldsymbol{v} \lrcorner \boldsymbol{\omega}(p) = \boldsymbol{\omega}(\boldsymbol{v})(p)$ for $p \in \mathcal{D}\big)$. With the dual of the adjoint representation, $\mathrm{ad}^*_{\boldsymbol{v}} : \mathfrak{g}(\mathcal{D})^* \to \mathfrak{g}(\mathcal{D})^*$, defined by

$$\langle \mathrm{ad}^*_{\boldsymbol{v}} Y, Z \rangle_{\mathfrak{g}(\mathcal{D})} = \langle Y, \mathrm{ad}_{\boldsymbol{v}} Z \rangle_{\mathfrak{g}(\mathcal{D})},$$

the Euler–Poincaré equations on $\mathcal{D}$ are

$$\frac{\mathrm{d}}{\mathrm{d}t} \frac{\delta \ell}{\delta \boldsymbol{v}} + \mathrm{ad}^*_{\boldsymbol{v}} \frac{\delta \ell}{\delta \boldsymbol{v}} = 0, \tag{B.1.1}$$

where δ denotes the functional derivative. (These equations are valid in a more general Lie group setting). In our specific case, equations (B.1.1) are the (weak) system

$$\frac{\partial}{\partial t} \boldsymbol{m} + \boldsymbol{v} \cdot \nabla \boldsymbol{m} + \nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{m} + (\operatorname{div} \boldsymbol{v}) \boldsymbol{m} = 0, \qquad \boldsymbol{m} = \mathcal{A} \boldsymbol{v},$$

with $\mathcal{A} = \Delta^2$, as follows.

We compute a formula for ad^*. First,

$$\langle \mathrm{ad}^*_{\boldsymbol{v}} (\boldsymbol{\omega} \otimes \mu), \boldsymbol{u} \rangle_{\mathfrak{g}(\mathcal{D})} = \int_{\mathcal{D}} \big([\boldsymbol{v}, \boldsymbol{u}]_{\mathfrak{g}(\mathcal{D})} \big) \lrcorner \boldsymbol{\omega}\, \mu,$$

since $\mathrm{ad}_{\boldsymbol{v}}(\boldsymbol{u}) = [\boldsymbol{v}, \boldsymbol{u}]_{\mathfrak{g}(\mathcal{D})}$. But the Lie bracket on $\mathfrak{g}(\mathcal{D})$ is minus the commutator bracket of vector fields [11], so

$$\langle \mathrm{ad}^*_{\boldsymbol{v}} (\boldsymbol{\omega} \otimes \mu), \boldsymbol{u} \rangle_{\mathfrak{g}(\mathcal{D})} = - \int_{\mathcal{D}} [\boldsymbol{v}, \boldsymbol{u}] \lrcorner \boldsymbol{\omega}\, \mu. \tag{B.1.2}$$

Recall that the divergence of a smooth vector field X relative to a volume measure μ is the smooth real-valued function $\mathrm{div}_\mu X$ defined by

$$(\mathrm{div}_\mu X)\mu = \mathrm{d}(X \lrcorner \mu).$$

Since $\mathrm{d}\mu$ is zero, Cartan's formula for the Lie derivative of a smooth differential form ν in the direction of a smooth vector field X,

$$\mathcal{L}_X \nu = X \lrcorner (\mathrm{d}\nu) + \mathrm{d}(X \lrcorner \nu), \tag{B.1.3}$$

gives the expression $\mathcal{L}_X \mu = (\mathrm{div}_\mu X)\mu$.

Continuing from Equation (B.1.2),

$$-\int_{\mathcal{D}} [\boldsymbol{v}, \boldsymbol{u}] \lrcorner \boldsymbol{\omega}\ \mu = \int_{\mathcal{D}} \boldsymbol{u} \lrcorner (\mathcal{L}_{\boldsymbol{v}} \boldsymbol{\omega}) \mu - \int_{\mathcal{D}} \mathcal{L}_{\boldsymbol{v}} (\boldsymbol{u} \lrcorner \boldsymbol{\omega}) \mu$$

(product rule for $\mathcal{L}$ and $\lrcorner$). The second term on the right-hand side gives

$$\begin{aligned}
-\mathcal{L}_{\boldsymbol{v}}(\boldsymbol{u} \lrcorner \boldsymbol{\omega})\mu &= (\boldsymbol{u} \lrcorner \boldsymbol{\omega})\mathcal{L}_{\boldsymbol{v}}\mu - \mathcal{L}_{\boldsymbol{v}}(\boldsymbol{u} \lrcorner \boldsymbol{\omega}\ \mu) \\
&= (\boldsymbol{u} \lrcorner \boldsymbol{\omega})\mathcal{L}_{\boldsymbol{v}}\mu - \mathrm{d}\big(\boldsymbol{v} \lrcorner (\boldsymbol{u} \lrcorner \boldsymbol{\omega}\ \mu)\big) - \boldsymbol{v} \lrcorner \mathrm{d}(\boldsymbol{u} \lrcorner \boldsymbol{\omega}\ \mu) \qquad \text{by (B.1.3)} \\
&= (\boldsymbol{u} \lrcorner \boldsymbol{\omega})\mathcal{L}_{\boldsymbol{v}}\mu \qquad \text{since contraction of a contraction is zero} \\
&= (\boldsymbol{u} \lrcorner \boldsymbol{\omega})(\mathrm{div}_\mu \boldsymbol{v})\mu.
\end{aligned}$$

Thus

$$-\int_{\mathcal{D}} [\boldsymbol{v}, \boldsymbol{u}] \lrcorner \boldsymbol{\omega}\ \mu = \int_{\mathcal{D}} \boldsymbol{u} \lrcorner \big(\mathcal{L}_{\boldsymbol{v}} \boldsymbol{\omega} + (\mathrm{div}_\mu \boldsymbol{v})\boldsymbol{\omega}\big)\ \mu,$$

and hence

$$\mathrm{ad}^*_{\boldsymbol{v}}(\boldsymbol{\omega} \otimes \mu) = \big(\mathcal{L}_{\boldsymbol{v}} \boldsymbol{\omega} + (\mathrm{div}_\mu \boldsymbol{v})\boldsymbol{\omega}\big) \otimes \mu.$$

Let us compute the expression for ad^* in Euclidean coordinates $\boldsymbol{q} = (q_1, q_2)$ on $\mathbb{R}^2$. First, observe that $\mathcal{L}_{\boldsymbol{v}}\, \mathrm{d}q_k = \mathrm{d}(\mathcal{L}_{\boldsymbol{v}} q_k) = \mathrm{d}(\boldsymbol{v} q_k) = \mathrm{d}v_k$. Hence (summation convention)

$$\begin{aligned}
\mathcal{L}_{\boldsymbol{v}} \boldsymbol{\omega} &= \mathcal{L}_{\boldsymbol{v}}(\omega_k\, \mathrm{d}q_k) \\
&= \mathcal{L}_{\boldsymbol{v}}(\omega_k)\, \mathrm{d}q_k + \omega_k(\mathcal{L}_{\boldsymbol{v}}\, \mathrm{d}q_k) \\
&= \boldsymbol{v}\omega_k\, \mathrm{d}q_k + \omega_k\, \mathrm{d}v_k \\
&= \left(v_\ell \frac{\partial \omega_k}{\partial q_\ell} + \omega_\ell \frac{\partial v_\ell}{\partial q_k} \right) \mathrm{d}q_k.
\end{aligned}$$

Since div in Euclidean coordinates is just the regular divergence,

$$\mathrm{ad}^*_{\boldsymbol{v}}(\boldsymbol{\omega} \otimes \mathrm{d}\boldsymbol{q}) = \big(\boldsymbol{v} \cdot \nabla \boldsymbol{\omega} + \nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{\omega} + (\mathrm{div}\, \boldsymbol{v})\boldsymbol{\omega}\big) \otimes \mathrm{d}\boldsymbol{q}.$$

Our n_c-particle dynamics in $\mathbb{R}^2$ is described by the kinetic energy Lagrangian (4.1.11), which is

$$\ell\big(\boldsymbol{v}(\boldsymbol{q})\big) = \frac{1}{2}\|\boldsymbol{v}\|^2 = \frac{1}{2}\int \boldsymbol{v} \lrcorner (\mathcal{A}\boldsymbol{v})\, \mathrm{d}\boldsymbol{q},$$

with $\mathcal{A} : \mathfrak{g}(\mathcal{D}) \to \mathfrak{g}(\mathcal{D})^*$ being $\mathcal{A} = \Delta^2$. Write $\mathcal{A}\boldsymbol{v} = \boldsymbol{m}$, as well as using $\boldsymbol{m}$ to denote the covector components of this one-form density.

The functional derivative of $\ell(\boldsymbol{v})$ with respect to $\boldsymbol{v}$ satisfies

$$\begin{aligned}\left\langle \frac{\delta \ell(\boldsymbol{v})}{\delta \boldsymbol{v}}, \delta \boldsymbol{v} \right\rangle_{\mathfrak{g}(\mathcal{D})} &= \lim_{\epsilon \to 0} \frac{1}{\epsilon}\big(\ell\big(\boldsymbol{v} + \epsilon \delta \boldsymbol{v}\big) - \ell(\boldsymbol{v})\big) \\ &= \int_{\mathcal{D}} (\delta \boldsymbol{v}) \lrcorner \boldsymbol{m} \, \mathrm{d}\boldsymbol{q} \\ &= \langle \boldsymbol{m}, \delta \boldsymbol{v} \rangle_{\mathfrak{g}(\mathcal{D})},\end{aligned}$$

so that $\delta \ell(\boldsymbol{v})/\delta \boldsymbol{v} = \boldsymbol{m}$. Therefore (B.1.1) is equivalent on $\mathbb{R}^2$ to the Euler-Poincaré equations

$$\frac{\partial}{\partial t}\boldsymbol{m} + \boldsymbol{v} \cdot \nabla \boldsymbol{m} + \nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{m} + (\operatorname{div} \boldsymbol{v})\boldsymbol{m} = 0, \qquad \boldsymbol{m} = \mathcal{A}\boldsymbol{v}, \tag{B.1.4}$$

as promised.

B.2 Why the particle ansatz?

We now show that

$$\boldsymbol{m}(\boldsymbol{q}, t) = \sum_{i=1}^{n_c} \boldsymbol{p}_i \delta\big(\boldsymbol{q} - \boldsymbol{q}_i\big)$$

is a weak solution of the Euler–Poincaré equations (B.1.4).

We know that, for the above $\boldsymbol{m}$, the velocity $\boldsymbol{v}$ is

$$\boldsymbol{v}(\boldsymbol{q}, t) = \sum_{i=1}^{n_c} \boldsymbol{p}_i G(\boldsymbol{q}, \boldsymbol{q}_i).$$

Now, the PDE system (B.1.4) in coordinates is

$$\frac{\partial m_k}{\partial t} + v_\ell \frac{\partial m_k}{\partial q_\ell} + m_\ell \frac{\partial v_\ell}{\partial q_k} + m_k \frac{\partial v_\ell}{\partial q_\ell} = 0, \qquad \text{for } k = 1, 2, \tag{B.2.1}$$

so to verify that $\boldsymbol{m}$ weakly solves (B.1.4) it suffices to show that m_k weakly solves (B.2.1) for $k = 1$.

Since $\boldsymbol{q}$ and $\boldsymbol{p}$ satisfy Hamilton's equations (see (4.1.14)),

$$\begin{aligned}\frac{\partial m_1}{\partial t} &= \sum_i \dot{p}_{i,1} \delta(\boldsymbol{q} - \boldsymbol{q}_i) - p_{i,1}\big(\nabla_{\boldsymbol{q}} \delta(\boldsymbol{q} - \boldsymbol{q}_i)\big) \cdot \dot{\boldsymbol{q}}_i \\ &= -\sum_{i,j} \delta(\boldsymbol{q} - \boldsymbol{q}_i) \frac{\partial}{\partial q_{i,1}} G(\boldsymbol{q}_i, \boldsymbol{q}_j) \boldsymbol{p}_i \cdot \boldsymbol{p}_j \\ &\quad - \sum_{i,j} p_{i,1} G(\boldsymbol{q}_i, \boldsymbol{q}_j)\big(\nabla_{\boldsymbol{q}} \delta(\boldsymbol{q} - \boldsymbol{q}_i)\big) \cdot \boldsymbol{p}_j.\end{aligned}$$

Let $\psi(\boldsymbol{q})$ be a test function satisfying the boundary conditions (4.1.3b). Then, using integration by parts and the property (4.1.2b) of δ,

$$\int_{\mathbb{D}} \frac{\partial m_1}{\partial t}\psi \,\mathrm{d}\boldsymbol{q} = -\sum_{i,j} \frac{\partial}{\partial q_{i,1}} G(\boldsymbol{q}_i, \boldsymbol{q}_j)\psi(\boldsymbol{q}_i)\boldsymbol{p}_i \cdot \boldsymbol{p}_j + \sum_{i,j} p_{i,1} G(\boldsymbol{q}_i, \boldsymbol{q}_j)\big(\nabla\psi(\boldsymbol{q}_i)\big) \cdot \boldsymbol{p}_j.$$

In a similar way, short exercises in manipulation yield

$$\begin{aligned}
\boldsymbol{v} \cdot \nabla m_1 &= \sum_{i,j} p_{i,1} G(\boldsymbol{q}, \boldsymbol{q}_j)\big(\nabla_{\boldsymbol{q}}\delta(\boldsymbol{q} - \boldsymbol{q}_i)\big) \cdot \boldsymbol{p}_j, \\
(\nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{m})_1 &= \sum_{i,j} \delta(\boldsymbol{q} - \boldsymbol{q}_i) \frac{\partial}{\partial q_1} G(\boldsymbol{q}, \boldsymbol{q}_j)\boldsymbol{p}_i \cdot \boldsymbol{p}_j, \\
(\operatorname{div} \boldsymbol{v}) m_1 &= \sum_{i,j} p_{i,1}\delta(\boldsymbol{q} - \boldsymbol{q}_i)\big(\nabla_{\boldsymbol{q}} G(\boldsymbol{q}, \boldsymbol{q}_j)\big) \cdot \boldsymbol{p}_j.
\end{aligned}$$

Continuing as above,

$$\begin{aligned}
\int_{\mathbb{D}} (\boldsymbol{v} \cdot \nabla m_1)\psi \,\mathrm{d}\boldsymbol{q} &= -\sum_{i,j} p_{i,1}\big(\nabla_{\boldsymbol{q}_i}\big(G(\boldsymbol{q}_i, \boldsymbol{q}_j)\psi(\boldsymbol{q}_i)\big)\big) \cdot \boldsymbol{p}_j, \\
\int_{\mathbb{D}} (\nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{m})_1 \psi \,\mathrm{d}\boldsymbol{q} &= \sum_{i,j} \frac{\partial}{\partial q_{i,1}} G(\boldsymbol{q}_i, \boldsymbol{q}_j)\psi(\boldsymbol{q}_i)\boldsymbol{p}_i \cdot \boldsymbol{p}_j, \\
\int_{\mathbb{D}} (\operatorname{div} \boldsymbol{v}) m_1 \psi \,\mathrm{d}\boldsymbol{q} &= \sum_{i,j} p_{i,1}\psi(\boldsymbol{q}_i)\big(\nabla_{\boldsymbol{q}_i} G(\boldsymbol{q}_i, \boldsymbol{q}_j)\big) \cdot \boldsymbol{p}_j.
\end{aligned}$$

Then, upon invoking the product rule for ∇ it is clear that

$$\int_{\mathbb{D}} \left(\frac{\partial m_1}{\partial t} + \boldsymbol{v} \cdot \nabla m_1 + (\nabla \boldsymbol{v}^{\mathrm{T}} \cdot \boldsymbol{m})_1 + (\operatorname{div} \boldsymbol{v}) m_1 \right) \psi \,\mathrm{d}\boldsymbol{q} = 0,$$

for all ψ satisfying (4.1.3b).

Appendix C

Introduction Code

```
function pendexample(lap)
%PENDEXAMPLE Plots for introductory pendulum example.
% INPUT
%   LAP String 'on' or 'off' for exporting the figures using LAPRINT.
% 11/08/05.

if nargin < 1, lap = 'off'; end

dirstr = 'C:\';
h = 0.2; N = 125;

% Explicit Euler: ==================================================
q = zeros(N,1); p = zeros(N,1);
q(1) = 0.5; p(1) = 0;

for k = 1:N-1
    q(k+1) = q(k) + h*p(k); p(k+1) = p(k) - h*sin(q(k));
end

figure, subplot(2,2,1)
plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5), hold on
x = -3:0.2:6; y  = -2:0.2:2.5;
[X,Y] = meshgrid(x,y);
Z = 0.5*Y.^2-cos(X);
```

```
contour(x,y,Z,25); hold off
title('Explicit Euler'), xlim([-3 6]), ylim([-2 2.5])
axis square, axis off

% Implicit Euler: ==================================================
q = zeros(N,1); p = zeros(N,1);
q(1) = 1.5; p(1) = 0;
options = optimset('Disp', 'off');

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)impEuler(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

subplot(2,2,2)
plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5), hold on
x = -3:0.2:6; y = -2:0.2:2.5;
[X,Y] = meshgrid(x,y);
Z = 0.5*Y.^2-cos(X);
contour(x,y,Z,25); hold off
title('Implicit Euler'), xlim([-3 6]), ylim([-2 2.5])
axis square, axis off

% Symplectic Euler: ================================================
h = 0.3; N = 30;
q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 0.7;

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)sympEuler(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

subplot(2,2,3)
```

```
plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5), hold on

q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 1.4;

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)sympEuler(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5)

q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 2.1;

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)sympEuler(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5)
x = -3:0.2:6; y = -2:0.2:2.5;
[X,Y] = meshgrid(x,y);
Z = 0.5*Y.^2-cos(X);
contour(x,y,Z,25); hold off
title('Symplectic Euler'), xlim([-3 6]), ylim([-2 2.5])
axis square, axis off

% Implicit midpoint rule: ==================================================
q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 0.7;

for k = 1:N-1
    x0 = [q(k); p(k)];
```

```
    x = fsolve(@(x)impmid(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

subplot(2,2,4)
plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5), hold on

q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 1.4;

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)impmid(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5)

q = zeros(N,1); p = zeros(N,1);
q(1) = 0; p(1) = 2.1;

for k = 1:N-1
    x0 = [q(k); p(k)];
    x = fsolve(@(x)impmid(x,h,q(k),p(k)),x0,options);
    q(k+1) = x(1); p(k+1) = x(2);
end

plot(q(1),p(1),'x',q(end),p(end),'o',q,p,'b-','LineWidth',1.5)
x = -3:0.2:6; y = -2:0.2:2.5;
[X,Y] = meshgrid(x,y);
Z = 0.5*Y.^2-cos(X);
contour(x,y,Z,25); hold off
title('Implicit Midpoint'), xlim([-3 6]), ylim([-2 2.5])
axis square, axis off

if strcmp(lap,'on')
```

```
    laprint(gcf,[dirstr 'PendIntro'],'scalefonts','off');
end

%==========================================================================
function y = impEuler(x,h,qk,pk)

y = [x(1) - qk - h*x(2); x(2) - pk + h*sin(x(1))];
%==========================================================================
function y = sympEuler(x,h,qk,pk)

y = [x(1) - qk - h*x(2); x(2) - pk + h*sin(qk)];
%==========================================================================
function y = impmid(x,h,qk,pk)

y = [x(1) - qk - h/2*(pk+x(2)); x(2) - pk + h*sin((qk+x(1))/2)];
```

Appendix D

Numerical Experiments I Code

D.1 Linearized Pendulum

D.1.1 Dependency Tree

```
                                  ┌─ linpend
linpendbatch ── linpendsol ──┼─ linpendcost
                                  └─ linpendcon

linpendplot ── linpendexact
```

D.1.2 `linpend.m`

```
function [x0,Aeq,beq] = linpend(discr,N,q0,q1,p0,p1)
%LINPEND Initializes variables for linear pendulum problem.
% INPUT
%   DISCR String: 'dEL', 'eE' or 'mid', depending on user's choice of
%           discretization of the problem.
%   N Number of time-steps.
%   Q0 Initial angle.
%   Q1 Final angle.
%   P0 Initial angular velocity (if DISCR = 'dEL').
%   P1 Final angular velocities (if DISCR = 'dEL').
%   T Final time.
% OUTPUT
```

```
%   X0 Initial guess for (q,f) if 'eE' / 'mid' or (q,p,f) if 'dEL'.
%   AEQ Linear equality constraints matrix.
%   BEQ Equality constraints vector: AEQ*X0 = BEQ.
% 22/07/05.

qstep = (q1 - q0)/N;

if strcmp(discr, 'dEL')
    % Initial guess for fmincon:
    x0 = zeros(2*(N+1),1);
    x0(1:N+1) = q0 + cumsum([0, repmat(qstep,1,N)]);
    % BCs for q:
    Aeq = zeros(2,2*(N+1));
    Aeq(2*[0, N]+[1, 2]) = 1;
    beq = [q0; q1];
elseif (strcmp(discr, 'eE') || strcmp(discr, 'mid'))
    % Initial guess for fmincon:
    x0 = zeros(3*(N+1),1);
    x0(1:N+1) = q0 + cumsum([0, repmat(qstep,1,N)]);
    x0([N+2, 2*(N+1)]) = [p0, p1];
    % BCs for q and qdot:
    Aeq = zeros(4,3*(N+1));
    Aeq(4*([1, N+1, N+2, 2*(N+1)]-1)+(1:4)) = 1;
    beq = [q0; q1; p0; p1];
else
    error('linpend:IllegalArgString','Illegal argument string.')
end
```

D.1.3 linpendbatch.m

```
%LINPENDBATCH Batch file for linearized pendulum problem.
% 21/07/05.

Nrange = (10:5:200);

for N = Nrange
```

```
    ind = (N - Nrange(1))/5 + 1;

    [objdEL(ind),outputdEL{ind},timedEL(ind)] = linpendsol(N,'dEL');
    fcountdEL(ind) = outputdEL{ind}.funcCount;
    itersdEL(ind) = outputdEL{ind}.iterations;

    [objeE(ind),outputeE{ind},timeeE(ind)] = linpendsol(N,'eE');
    fcounteE(ind) = outputeE{ind}.funcCount;
    iterseE(ind) = outputeE{ind}.iterations;

    [objmid(ind),outputmid{ind},timemid(ind)] = linpendsol(N,'mid');
    fcountmid(ind) = outputmid{ind}.funcCount;
    itersmid(ind) = outputmid{ind}.iterations;
end

datestring = datestr(now);
datestring(datestring == ' ') = [];
datestring(datestring == '-') = [];
datestring(datestring == ':') = [];
save thedate datestring
save(['linpendvars' datestring])
exit
```

D.1.4 linpendcon.m

```
function [C,Ceq,GC,GCeq] = linpendcon(x,discr,h,N,qdot0,qdot1)
%LINPENDCON Constraints for linearized pendulum problem.
% INPUT
%   X Vector (q,p,f) for explicit Euler / midpoint formulation, or (q,f)
%       for variational.
%   DISCR String: 'dEL', 'eE' or 'mid', depending on user's choice of
%           discretization of the problem.
%   H Step-size.
%   N Number of time-steps.
%   QDOT0 Velocity at time t = 0.
%   QDOT1 Velocity at time t = T.
```

```
% 02/09/05.

C = [];

if strcmp(discr, 'dEL')
    Ceq = zeros(N+1,1);
    Ceq(1) = qdot0 + D1Ld(x(1),x(2),h) + (h/4)*(x((N+1)+2)+x((N+1)+1));
    % Discrete Euler-Lagrange terms:
    % Ceq(k) = D2Ld(q(k-1),q(k)) + D1Ld(q(k),q(k+1))
    %   + h/4*(f(k-1) + 2f(k) + f(k+1))
    Ceq(2:N) = D2Ld(x(1:N-1),x(2:N),h) + D1Ld(x(2:N),x(3:N+1),h) ...
        + (h/4)*(x((N+1)+(1:N-1)) ...
        + 2*x((N+1)+(2:N)) + x((N+1)+(3:N+1)));
    Ceq(N+1) = -qdot1 + D2Ld(x(N),x(N+1),h) + (h/4)*(x(2*(N+1)) ...
        + x((N+1)+N));
elseif strcmp(discr, 'eE')
    [q,p,f] = unpack(x,N);
    Ceq = zeros(2*N,1);
    Ceq(1:N) = q(2:N+1) - q(1:N) - h*p(1:N);
    Ceq(N + (1:N)) = p(2:N+1) - p(1:N) - h*(f(1:N) - q(1:N));
elseif strcmp(discr, 'mid')
    [q,p,f] = unpack(x,N);
    Ceq = zeros(2*N,1);
    Ceq(1:N) = q(2:N+1) - q(1:N) - h/2*(p(1:N)+p(2:N+1));
    Ceq(N+(1:N)) = p(2:N+1) - p(1:N) ...
        - h/2*(f(1:N) + f(2:N+1) - q(1:N) - q(2:N+1));
else
    error('linpendcon:IllegalArgString','Illegal argument string.')
end

if nargout > 2
    if strcmp(discr, 'dEL')
        GC = [];
        % GCeq(i,j) = dCeq(j) / dx(i)
        % GCeq = [GCeq_q; GCeq_f].
        GCeq_q = zeros(N+1); GCeq_f = zeros(N+1);
```

```
    % Main diagonal, GCeq_q:
    GCeq_q([1 (N+1)^2]) = -h/4 + 1/h;
    GCeq_q((N+1)+2:(N+1)+1:(N+1)^2-(N+1)-1) = -h/2 + 2/h;
    % Leading subdiagonal, GCeq_q:
    GCeq_q(2:(N+1)+1:(N+1)^2-N) = -h/4 - 1/h;
    % Leading superdiagonal, GCeq_q:
    GCeq_q((N+1)+1:(N+1)+1:(N+1)^2-1) = -h/4 - 1/h;
    % Main diagonal, GCeq_f:
    GCeq_f([1 (N+1)^2]) = h/4;
    GCeq_f((N+1)+2:(N+1)+1:(N+1)^2-(N+1)-1) = h/2;
    % Leading subdiagonal, GCeq_f:
    GCeq_f(2:(N+1)+1:(N+1)^2-N) = h/4;
    % Leading superdiagonal, GCeq_f:
    GCeq_f((N+1)+1:(N+1)+1:(N+1)^2-1) = h/4;
    GCeq = [GCeq_q; GCeq_f];
elseif strcmp(discr, 'eE')
    GC = [];
    % GCeq(i,j) = dCeq(j) / dx(i)
    % GCeq = [GCeq_q; GCeq_p; GCeq_f].
    GCeq_q = zeros(N+1,2*N); GCeq_p = zeros(N+1,2*N);
    GCeq_f = zeros(N+1,2*N);

    diagon = 1:((N+1)+1):(N*(N+1)-1);

    GCeq_q(diagon) = -1; % Main diagonal, GCeq_q.
    GCeq_q(N*(N+1)+diagon) = h; % Second diagonal, GCeq_q.
    GCeq_q(1+diagon) = 1; % Leading subdiagonal, GCeq_q.
    GCeq_p(diagon) = -h; % Main diagonal, GCeq_p.
    GCeq_p(N*(N+1)+diagon) = -1; % Second diagonal, GCeq_p.
    GCeq_p(N*(N+1)+1+diagon) = 1; % Second subdiagonal, GCeq_p.
    GCeq_f(N*(N+1)+diagon) = -h; % Second diagonal, GCeq_f.
    GCeq = [GCeq_q; GCeq_p; GCeq_f];
else
    GC = [];
    % GCeq(i,j) = dCeq(j) / dx(i)
    % GCeq = [GCeq_q; GCeq_p; GCeq_f].
```

```
        GCeq_q = zeros(N+1,2*N); GCeq_p = zeros(N+1,2*N);
        GCeq_f = zeros(N+1,2*N);

        diagon = 1:((N+1)+1):(N*(N+1)-1);

        GCeq_q(diagon) = -1; % Main diagonal, GCeq_q.
        GCeq_q(N*(N+1)+diagon) = h/2; % Second diagonal, GCeq_q.
        GCeq_q(1+diagon) = 1; % Leading subdiagonal, GCeq_q.
        GCeq_q(N*(N+1)+1+diagon) = h/2; % Second subdiagonal, GCeq_q.
        GCeq_p(diagon) = -h/2; % Main diagonal, GCeq_p.
        GCeq_p(N*(N+1)+diagon) = -1; % Second diagonal, GCeq_p.
        GCeq_p(1+diagon) = -h/2; % Leading subdiagonal, GCeq_p.
        GCeq_p(N*(N+1)+1+diagon) = 1; % Second subdiagonal, GCeq_p.
        GCeq_f(N*(N+1)+diagon) = -h/2; % Second diagonal, GCeq_f.
        GCeq_f(N*(N+1)+1+diagon) = -h/2; % Second subdiagonal, GCeq_f.
        GCeq = [GCeq_q; GCeq_p; GCeq_f];
    end
end

%==========================================================================
function D = D1Ld(qk,qkp,h)

D = -h/4*(qkp + qk) - (qkp - qk)/h;
%==========================================================================
function D = D2Ld(qk,qkp,h)

D = -h/4*(qkp + qk) + (qkp - qk)/h;
%==========================================================================
function [q,p,f] = unpack(x,N)
%UNPACK Splits X into component parts.

q = x(1:N+1); p = x(N+1+(1:N+1)); f = x(2*(N+1)+(1:N+1));
```

D.1.5 linpendcost.m

```
function [J,G] = linpendcost(x,discr,h,N)
```

```
%LINPENDCOST Discrete cost for linearized pendulum problem.
% INPUT
%   X Vector (q,p,f) for explicit Euler / midpoint formulation, or (q,f)
%       for variational.
%   DISCR String: 'dEL', 'eE' or 'mid', depending on user's choice of
%           discretization of the problem.
%   H Step-size.
%   N Number of time-steps.
% OUTPUT
%   J Cost at X.
%   G Gradient of cost function.
% 02/09/05.

f = x(end-(N+1)+(1:N+1)); % Unpack forces.

if (strcmp(discr,'dEL') || strcmp(discr,'mid'))
    J = h/4*sum((f(1:N) + f(2:N+1)).^2);
elseif strcmp(discr,'eE')
    J = h*norm(f(1:N))^2;
else
    error('linpendcost:IllegalArgString', 'Illegal argument string.')
end

if nargout > 1
    if strcmp(discr,'dEL')
        G = zeros(2*(N+1),1);
        G(1*(N+1)+1) = h/2*(f(1) + f(2));
        G(1*(N+1)+(2:N)) = h/2*(f(1:N-1) + 2*f(2:N) + f(3:N+1));
        G(2*(N+1)) = h/2*(f(N) + f(N+1));
    elseif strcmp(discr, 'eE')
        G = zeros(3*(N+1),1);
        G(2*(N+1)+(1:N)) = 2*h*f(1:N);
    else
        G = zeros(3*(N+1),1);
        G(2*(N+1)+1) = h/2*(f(1) + f(2));
        G(2*(N+1)+(2:N)) = h/2*(f(1:N-1) + 2*f(2:N) + f(3:N+1));
```

```
        G(3*(N+1)) = h/2*(f(N) + f(N+1));
    end
end
```

D.1.6 linpendexact.m

```
function J = linpendexact
%LINPENDEXACT Exact cost for linear pendulum problem.
% 21/07/05.

% Initial conditions:
q0 = pi/2 + pi/10; q1 = pi/2 - pi/10;
p0 = 0; p1 = 0;

c(2) = (8*q1*pi - 16*p1 - 8*p0*pi - 16*q0)/(pi^2 - 4);
c(1) = 4*p0 - 4*q1 + c(2)*pi/2; c = c.';
k = [q0; p0 - c(1)/4];

J = pi/16*(c(1)^2 + c(2)^2) - c(1)*c(2)/4;
```

D.1.7 linpendplot.m

```
function linpendplot(lap)
%LINPENDPLOT Plots for linearized pendulum results.
% INPUT
%   LAP String 'on' or 'off' for exporting the figures using LAPRINT.
% 11/08/05.

if nargin < 1, lap = 'off'; end

load thedate datestring
load(['linpendvars' datestring])
dirstr = 'C:\';

J = linpendexact; obj = repmat(J,1,39);

% Variational vs. eE: ==================================================
```

```
figure, subplot(1,2,1)
pN1 = plot(Nrange,timedEL,Nrange,timeeE);
xlabel('Time-steps'), title('Time taken (sec.)'), axis square
xlim([Nrange(1) Nrange(end)])

subplot(1,2,2)
pN2 = plot(Nrange,objdEL,Nrange,objeE,Nrange,obj);
xlabel('Time-steps'), title('Final cost'), axis square
xlim([Nrange(1) Nrange(end)])

leg = legend(pN2,'Var.','Euler','Exact');
legpos = get(leg,'position');
legpos(1) = 0.72; legpos(2) = 0.568;
set(leg,'position',legpos), legend boxoff

if strcmp(lap,'on')
    laprint(gcf,[dirstr 'linpendvareE'],'scalefonts','off');
end

% Variational vs. finite diff.: ==================================
figure, subplot(1,2,1)
pmid1 = plot(Nrange,timedEL,Nrange,timemid);
xlabel('Time-steps'), title('Time taken (sec.)'), axis square
xlim([Nrange(1) Nrange(end)])

subplot(1,2,2)
pmid2 = plot(Nrange,objdEL,Nrange,objmid);
xlabel('Time-steps'), title('Final cost'), axis square
xlim([Nrange(1) Nrange(end)])

leg = legend(pmid2,'Var.','Ham.\ Eq''ns');
legpos = get(leg,'position');
legpos(1) = 0.622; legpos(2) = 0.508;
set(leg,'position',legpos), legend boxoff

if strcmp(lap,'on')
```

```
    laprint(gcf,[dirstr 'linpendvarctrl'],'scalefonts','off');
end
```

D.1.8 linpendsol.m

```
function [J,output,time] = linpendsol(N,discr)
%LINPENDSOL Solves linearized pendulum problem.
% INPUT
%   N Number of time-steps.
%   DISCR String: 'dEL', 'eE' or 'mid', depending on user's choice of
%           discretization of the problem.
% OUTPUT
%   J Optimal cost as computed by FMINCON.
%   OUTPUT Output structure from FMINCON.
%   TIME Time taken by the minimization.
% 21/07/05.

if nargin < 2, discr = 'dEL'; end

q0 = pi/2 + pi/10; q1 = pi/2 - pi/10;
p0 = 0; p1 = 0;
T = pi/2; h = T/N;

options = optimset('Display', 'off', ...
    'GradConstr', 'on', ...
    'GradObj', 'on', ...
    'LargeScale', 'off');

% Initial guess and equality constraints:
[x0,Aeq,beq] = linpend(discr,N,q0,q1,p0,p1);

tic
[x,J,exitflag,output] = ...
    fmincon(@(x)linpendcost(x,discr,h,N),x0,[],[],Aeq,beq,[],[], ...
    @(x)linpendcon(x,discr,h,N,p0,p1),options);
time = toc;
```

D.2 Orbital Transfer

D.2.1 Dependency Tree

```
                ┌── orb
orbbatch ───────┼── orbcost
                └── orbnlcon

orbRKbatch ─── orbRK ─── orbsyst

orbplot
```

D.2.2 orb.m

```
function [x0,Aeq,beq,options] = orb(discr,N,r0,r1,phi1,varargin)
%ORB Initializes variables for orbital transfer problem.
% INPUT
%   DISCR String: 'eE' or 'dEL', depending on user's choice of
%           discretization of the problem.
%   N Number of time-steps.
%   R0 Initial radius.
%   R1 Final radius.
%   PHI1 Final angle.
%   T Final time.
%   OPTIONAL:
%       If DISCR = 'eE', user must also input, in this order,
%       PR0 Initial p_r.
%       PR1 Final p_r.
%       PPHI0 Initial p_phi.
%       PPHI1 Final p_phi.
% OUTPUT
%   X0 Initial guess for (r,phi,u) if eE or (r,phi,pr,pphi,u) if dEL.
%   AEQ Linear equality constraints matrix.
%   BEQ Equality constraints vector: AEQ*X0 = BEQ.
%   OPTIONS Options structure for FMINCON.
% 22/07/05.
```

```
% Steps:
rstep = (r1 - r0)/N; phistep = phi1/N;

options = optimset('Display','iter', ...
    'GradObj','on', ...
    'LargeScale', 'off');

if strcmp(discr,'dEL')
    x0 = zeros(3*(N+1),1);

    x0(1:N+1) = r0 + cumsum([0 repmat(rstep,1,N)]);
    x0((N+1)+(1:N+1)) = 0 + cumsum([0 repmat(phistep,1,N)]);

    % BCs for q: q(0) = q0, q(T) = qN:
    Aeq = zeros(4,3*(N+1));
    Aeq([1, 4*N+2, 4*(N+1)+3, 8*(N+1)]) = 1;
    beq = Aeq*x0;
elseif strcmp(discr,'eE')
    pr0 = varargin{1}; pr1 = varargin{2};
    pphi0 = varargin{3}; pphi1 = varargin{4};

    % Initial guess for fmincon:
    pphistep = (pphi1 - pphi0)/N;

    x0 = zeros(5*(N+1),1);
    x0([N+2, 2*(N+1)]) = [pr0, pr1];
    x0(1:N+1) = r0 + cumsum([0 repmat(rstep,1,N)]);
    x0(2*(N+1)+(1:(N+1))) = 0 + cumsum([0 repmat(phistep,1,N)]);
    x0(3*(N+1)+(1:N+1)) = pphi0 + cumsum([0 repmat(pphistep,1,N)]);

    % BCs for q and p:
    Aeq = zeros(8,5*(N+1));

    % Put 1 in BC positions:
    Aeq([1, 8*((N+1:(N+1):4*(N+1)) - 1) + (2:2:8), ...
        8*(N+1:(N+1):3*(N+1)) + (3:2:7)]) = 1;
```

```
    beq = Aeq*x0;

    options = optimset(options,'GradConstr','on');
else
    error('orb:IllegalArgString','Illegal argument string.')
end
```

D.2.3 orbbatch.m

```
%ORBBATCH Batch file for orbital transfer problem.
% 11/08/05.

m = 100; M = 6e24; gamma = 6.673e-26; r0 = 5; r1 = 6; rd0 = 0; rd1 = 0;
T0 = 2*pi*sqrt(r0^3/(gamma*M)); T1 = 2*pi*sqrt(r1^3/(gamma*M));

% Boundary velocities.
phid0 = sqrt((gamma*M)/r0^3); phid1 = sqrt((gamma*M)/r1^3);

% Boundary momenta.
pr0 = m*rd0; pr1 = m*rd1;
pphi0 = m*sqrt(gamma*M*r0); pphi1 = m*sqrt(gamma*M*r1);

Nrange{1} = 10:5:50; Nrange{2} = 10:10:100;
Nrange2{1} = 100; Nrange2{2} = 200;
Nrange3{1} = 150; Nrange3{2} = 300;

for p = 1:2
    % Final time and angle:
    T = p*(T0+T1)/2; phi1 = 2*pi*p;

    for N = Nrange{p}
        h = T/N; ind = (N-Nrange{p}(1))/(Nrange{p}(2)-Nrange{p}(1)) + 1;

        % Midpoint:
        [x0,Aeq,beq,options] = orb('dEL',N,r0,r1,phi1);
```

```
        tic;
        [xmid{p,ind},objfinmid{p}(ind),exitflag,outputmid{p,ind}] = ...
            fmincon(@(x)orbcost(x,'dEL',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'dEL',h,m,M,N,gamma, ...
                rd0,rd1,phid0,phid1),options);
        timemid{p}(ind) = toc;

        % Euler:
        [x0,Aeq,beq,options] = ...
            orb('eE',N,r0,r1,phi1,pr0,pr1,pphi0,pphi1);
        tic;
        [xeE{p,ind},objfineE{p}(ind),exitflag,outputeE{p,ind}] = ...
            fmincon(@(x)orbcost(x,'eE',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'eE',h,m,M,N,gamma),options);
        timeeE{p}(ind) = toc;
    end

    % Halved step:
    for N = Nrange2{p}
        h = T/N; ind = 1;

        % Midpoint:
        [x0,Aeq,beq,options] = orb('dEL',N,r0,r1,phi1);
        tic;
        [xmid2{p,ind},objfinmid2{p}(ind),exitflag,outputmid2{p,ind}] =
            ...
            fmincon(@(x)orbcost(x,'dEL',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'dEL',h,m,M,N,gamma, ...
                rd0,rd1,phid0,phid1),options);
        timemid2{p}(ind) = toc;

        % Euler:
        [x0,Aeq,beq,options] = ...
            orb('eE',N,r0,r1,phi1,pr0,pr1,pphi0,pphi1);
        tic;
        [xeE2{p,ind},objfineE2{p}(ind),exitflag,outputeE2{p,ind}] = ...
```

```
            fmincon(@(x)orbcost(x,'eE',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'eE',h,m,M,N,gamma),options);
        timeeE2{p}(ind) = toc;
    end

    % Step/3:
    for N = Nrange3{p}
        h = T/N; ind = 1;

        % Midpoint:
        [x0,Aeq,beq,options] = orb('dEL',N,r0,r1,phi1);
        tic;
        [xmid3{p,ind},objfinmid3{p}(ind),exitflag,outputmid3{p,ind}] =
            ...
            fmincon(@(x)orbcost(x,'dEL',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'dEL',h,m,M,N,gamma, ...
                rd0,rd1,phid0,phid1),options);
        timemid3{p}(ind) = toc;

       % Euler:
        [x0,Aeq,beq,options] = ...
            orb('eE',N,r0,r1,phi1,pr0,pr1,pphi0,pphi1);
        tic;
        [xeE3{p,ind},objfineE3{p}(ind),exitflag,outputeE3{p,ind}] = ...
            fmincon(@(x)orbcost(x,'eE',h,N),x0,[],[],Aeq,beq,[],[], ...
            @(x)orbnlcon(x,'eE',h,m,M,N,gamma),options);
        timeeE3{p}(ind) = toc;
    end
end

datestring = datestr(now);
datestring(datestring == ' ') = [];
datestring(datestring == '-') = [];
datestring(datestring == ':') = [];
save thedate datestring
save(['orbvarsmid' datestring],'xmid','Nrange', ...
```

```
    'objfinmid','outputmid','timemid')
save(['orbvarseE' datestring],'xeE','Nrange', ...
    'objfineE','outputeE','timeeE')
save(['orbvarsmid2' datestring],'xmid2','Nrange2', ...
    'objfinmid2','outputmid2','timemid2')
save(['orbvarseE2' datestring],'xeE2','Nrange2', ...
    'objfineE2','outputeE2','timeeE2')
save(['orbvarsmid3' datestring],'xmid3','Nrange3', ...
    'objfinmid3','outputmid3','timemid3')
save(['orbvarseE3' datestring],'xeE3','Nrange3', ...
    'objfineE3','outputeE3','timeeE3')
exit
```

D.2.4 orbcost.m

```
function [J,G] = orbcost(x,discr,h,N)
%ORBCOST Control effort for orbital transfer problem.
% INPUT
%   X Vector (r,phi,u) for explicit Euler formulation, or
%       (r,phi,pr,pphi,u) for variational.
%   DISCR String: 'eE' or 'dEL', depending on user's choice of
%       discretization of the problem.
%   H Step-size.
%   N Number of time-steps.
% OUTPUT
%   J Cost at X.
%   G Gradient of cost function.
% 22/07/05.

if strcmp(discr,'dEL')
    J = h/4*sum((x(2*(N+1)+(1:N)) + x(2*(N+1)+(2:N+1))).^2);
elseif strcmp(discr,'eE')
    J = h*norm(x(4*(N+1)+(1:N)))^2;
else
    error('orbcost:IllegalArgString','Illegal argument string.')
end
```

```
if nargout > 1
    if strcmp(discr,'dEL')
        G = zeros(3*(N+1),1);
        G(2*(N+1)+1) = h/2*(x(2*(N+1)+1) + x(2*(N+1)+2));
        G(2*(N+1)+2:2*(N+1)+N) = ...
            h/2*(x(2*(N+1)+1:2*(N+1)+N-1) ...
            + 2*x(2*(N+1)+2:2*(N+1)+N) ...
            + x(2*(N+1)+3:2*(N+1)+N+1));
        G(3*(N+1)) = h/2*(x(3*(N+1)-1) + x(3*(N+1)));
    else
        G = zeros(5*(N+1),1);
        G(4*(N+1)+(1:N)) = 2*h*x(4*(N+1)+(1:N));
    end
end
```

D.2.5 orbnlcon.m

```
function [C,Ceq,GC,GCeq] = orbnlcon(x,discr,h,m,M,N,gamma,varargin)
%ORBNLCON Constraints for orbital transfer problem.
% INPUT
%   X Vector (r,phi,u) for explicit Euler formulation, or
%       (r,phi,pr,pphi,u) for variational.
%   DISCR String: 'eE' or 'dEL', depending on user's choice of
%           discretization of the problem.
%   H Step-size.
%   m Satellite's mass.
%   M Earth's mass.
%   N Number of time-steps.
%   GAMMA Gravitational constant.
%   OPTIONAL:
%       If DISCR = 'dEL', user must also input, in this order,
%       RD0 Initial radial velocity.
%       RD1 Final radial velocity.
%       PHID0 Initial angular velocity.
%       PHID1 Final angular velocity.
```

```
% 22/07/05.

C = [];

if strcmp(discr,'dEL')
    rd0 = varargin{1}; rd1 = varargin{2};
    phid0 = varargin{3}; phid1 = varargin{4};

    % Unpack x:
    r = x(1:N+1); phi = x(N+1+(1:N+1));
    u = x(2*(N+1)+(1:N+1));

    Ceq = zeros(2*(N+1),1);
    Ceq(1:2) = D2L(r(1),phi(1),rd0,phid0,m) ...
        + D1Ld(r(1),phi(1),r(2),phi(2),h,m,M,gamma) ...
        + (h/4)*[0; r(2)*u(2)+r(1)*u(1)];

    for k = 2:N
        Ceq(2*(k-1)+(1:2)) ...
            = D2Ld(r(k-1),phi(k-1),r(k),phi(k),h,m,M,gamma) ...
            + D1Ld(r(k),phi(k),r(k+1),phi(k+1),h,m,M,gamma) ...
            + (h/4)*[0; r(k-1)*u(k-1)+2*r(k)*u(k) + r(k+1)*u(k+1)];
    end

    Ceq(2*N+(1:2)) = -D2L(r(N+1),phi(N+1),rd1,phid1,m) ...
        + D2Ld(r(N),phi(N),r(N+1),phi(N+1),h,m,M,gamma) ...
        + (h/4)*[0; r(N)*u(N) + r(N+1)*u(N+1)];
elseif strcmp(discr,'eE')
    % Unpack x:
    r = x(1:N+1); pr = x(N+1+(1:N+1));
    phi = x(2*(N+1)+(1:N+1)); pphi = x(3*(N+1)+(1:N+1));
    u = x(4*(N+1)+(1:N+1));

    Ceq = zeros(4*N,1);
    for k = 1:N
        Ceq(k) = r(k+1) - r(k) - h*pr(k)/m;
```

```
        Ceq(N+k) = pr(k+1) - pr(k) - h*pphi(k)^2/(m*r(k)^3) ...
            + h*gamma*M*m/(r(k)^2);
        Ceq(2*N+k) = phi(k+1) - phi(k) - h*pphi(k)/(m*r(k)^2);
        Ceq(3*N+k) = pphi(k+1) - pphi(k) - h*r(k)*u(k);
    end
else
    error('orbnlcon:IllegalArgString','Illegal argument string.')
end

if nargout > 2
    if strcmp(discr,'dEL')
        error('orbnlcon:NodELJacobian','Jacobian not coded.')
    else
        GC = [];
        GCeq = zeros(5*(N+1),4*N);
        for k = 1:N
            GCeq(k,N+k) = (h*3*pphi(k)^2)/(m*r(k)^4) ...
                - h*2*gamma*M*m/r(k)^3;
            GCeq(3*(N+1)+k,N+k) = -h*2*pphi(k)/(m*r(k)^3);
            GCeq(k,2*N+k) = h*2*pphi(k)/(m*r(k)^3);
            GCeq(3*(N+1)+k,2*N+k) = -h/(m*r(k)^2);
            GCeq(k,3*N+k) = -h*u(k);
            GCeq(4*(N+1)+k,3*N+k) = -h*r(k);
        end
        GCeq([5*(N+1)*(0:N-1) + (1:N), ...
            5*(N+1)*(N:2*N-1) + (N+2:2*N+1), ...
            5*(N+1)*(2*N:3*N-1) + (2*(N+1)+1:2*(N+1)+N), ...
            5*(N+1)*(3*N:4*N-1) + (3*(N+1)+1:3*(N+1)+N)]) = -1;
        GCeq([5*(N+1)*(0:N-1) + (2:N+1), ...
            5*(N+1)*(N:2*N-1) + (N+3:2*N+2), ...
            5*(N+1)*(2*N:3*N-1) + (2*(N+1)+2:3*(N+1)), ...
            5*(N+1)*(3*N:4*N-1) + (3*(N+1)+2:4*(N+1))]) = 1;
        GCeq(5*(N+1)*(0:N-1) + (N+2:2*N+1)) = -h/m;
    end
end
```

```
%==========================================================================
function D = D1Ld(rk,phik,rkp,phikp,h,m,M,gamma)

D = [-m*(rkp-rk)/h + m*(phikp-phik)^2*(rkp+rk)/(4*h) ...
    - (2*gamma*M*m*h)/(rkp+rk)^2;
    -m*(rkp+rk)^2*(phikp-phik)/(4*h)];
%==========================================================================
function D = D2L(r,phi,rdot,phidot,m)

D = [m*rdot; m*r^2*phidot];
%==========================================================================
function D = D2Ld(rk,phik,rkp,phikp,h,m,M,gamma)

D = [m*(rkp-rk)/h + m*(phikp-phik)^2*(rkp+rk)/(4*h) ...
    - (2*gamma*M*m*h)/(rkp+rk)^2;
    m*(rkp+rk)^2*(phikp-phik)/(4*h)];
```

D.2.6 orbplot.m

```
function orbplot(lap)
%ORBPLOT Plots for orbital transfer results.
% INPUT
%   LAP String 'on' or 'off' for exporting the figures using LAPRINT.
% 11/08/05.

if nargin < 1, lap = 'off'; end

load thedate datestring
load(['orbvarsmid' datestring]), load(['orbvarseE' datestring])
load(['orbvarsmid2' datestring]), load(['orbvarseE2' datestring])
load(['orbvarsmid3' datestring]), load(['orbvarseE3' datestring])
load(['orbvarsRK' datestring])
dirstr = 'C:\';

% Time / cost comparisons: ==========================================
for p = 1:2
```

```
    figure, subplot(1,2,1)
    pl{1} = plot(Nrange{p},timemid{p},Nrange{p},timeeE{p});
    xlabel('Time-steps'), title('Time taken (sec.)'), axis square
    xlim([Nrange{p}(1) Nrange{p}(end)])

    subplot(1,2,2)
    pl{2} = plot(Nrange{p},objfinmid{p},Nrange{p},objfineE{p});
    xlabel('Time-steps'), title('Final cost'), axis square
    xlim([Nrange{p}(1) Nrange{p}(end)])

    leg = legend(pl{2},'Mid.','Euler');
    legpos = get(leg,'position');
    legpos(1) = 0.734; legpos(2) = 0.62;
    set(leg,'position',legpos), legend boxoff

    if strcmp(lap,'on'),
        laprint(gcf,sprintf('%sorb%d',dirstr,p),'scalefonts','off');
    end
end

% Trajectories: ==================================================
for p = 1:2
    N = 50*p; leng = length(Nrange{p});
    rmid = xmid{p,leng}(1:N+1);
    phimid = xmid{p,leng}(N+2:2*(N+1));
    reE = xeE{p,leng}(1:N+1);
    phieE = xeE{p,leng}(2*(N+1)+1:3*(N+1));
    rRK = q{p}(:,1);
    phiRK = q{p}(:,2);
    figure
    polar(phimid,rmid,'b'), hold on
    polar(phieE,reE,'r'), hold on
    polar(phiRK,rRK,'g'); hold off

    leg = legend('Mid.','Euler','R-K');
    legpos = get(leg,'position');
```

```
    legpos(1) = 0.444; legpos(2) = 0.444;
    set(leg,'position',legpos)

    if strcmp(lap,'on'),
        laprint(gcf,sprintf('%sorbtraj%d',dirstr,p), ...
            'scalefonts','off','width',10);
    end
end

% Forces: ==================================================================
M = 6e24; gamma = 6.673e-26; r0 = 5; r1 = 6;
T0 = 2*pi*sqrt(r0^3/(gamma*M)); T1 = 2*pi*sqrt(r1^3/(gamma*M));

% N = 50, 100:
figure
for p = 1:2
    T = p*(T0+T1)/2;
    N = 50*p; leng = length(Nrange{p});
    fmid = xmid{p,leng}(2*(N+1)+(1:N+1));
    feE = xeE{p,leng}(4*(N+1)+(1:N+1));
    subplot(1,2,p)
    pl{p} = plot(linspace(0,T,N+1),fmid,linspace(0,T,N+1),feE);
    xlim([0 T]), ylim([0 0.04])
    xlabel('Time (sec.)'), axis square
    title(sprintf('Force, $p = %d$',p))
end
leg = legend(pl{2},'Mid.','Euler');
legpos = get(leg,'position');
legpos(1) = 0.667; legpos(2) = 0.553;
set(leg,'position',legpos), legend boxoff

if strcmp(lap,'on'),
    laprint(gcf,sprintf('%sorbforces',dirstr),'scalefonts','off');
end

% N = 100, 200:
```

```
figure
for p = 1:2
    T = p*(T0+T1)/2;
    N = 100*p; leng = length(Nrange2{p});
    fmid2 = xmid2{p,leng}(2*(N+1)+(1:N+1));
    feE2 = xeE2{p,leng}(4*(N+1)+(1:N+1));
    subplot(1,2,p)
    pl2{p} = plot(linspace(0,T,N+1),fmid2,linspace(0,T,N+1),feE2);
    xlim([0 T]), ylim([0 0.04])
    xlabel('Time (sec.)'), axis square
    title(sprintf('Force, $p = %d$',p))
end
leg = legend(pl2{2},'Mid.','Euler');
legpos = get(leg,'position');
legpos(1) = 0.667; legpos(2) = 0.553;
set(leg,'position',legpos), legend boxoff

if strcmp(lap,'on'),
    laprint(gcf,sprintf('%sorbforces2',dirstr),'scalefonts','off');
end

% N = 150, 300:
figure
for p = 1:2
    T = p*(T0+T1)/2;
    N = 150*p; leng = length(Nrange3{p});
    fmid3 = xmid3{p,leng}(2*(N+1)+(1:N+1));
    feE3 = xeE3{p,leng}(4*(N+1)+(1:N+1));
    subplot(1,2,p)
    pl3{p} = plot(linspace(0,T,N+1),fmid3,linspace(0,T,N+1),feE3);
    xlim([0 T]), ylim([0 0.04])
    xlabel('Time (sec.)'), axis square
    title(sprintf('Force, $p = %d$',p))
end
leg = legend(pl3{2},'Mid.','Euler');
legpos = get(leg,'position');
```

```
legpos(1) = 0.667; legpos(2) = 0.553;
set(leg,'position',legpos), legend boxoff

if strcmp(lap,'on'),
    laprint(gcf,sprintf('%sorbforces3',dirstr),'scalefonts','off');
end
```

D.2.7 orbRK.m

```
function q = orbRK(x,m,M,N,r0,T,gamma,phid0)
%ORBRK Calls ODE45 on orbital transfer problem.
% 18/07/05.

u = x(2*(N+1)+(1:N+1));
pp = spline(linspace(0,T,N+1),u);
t_int = 0:(1e-3):T;
IC = [r0 0 0 phid0];
[t q] = ode45(@orbsyst,t_int,IC,[],pp,m,M,gamma);
```

D.2.8 orbRKbatch.m

```
%ORBRKBATCH Batch file for ODE45 solved orbit problem.
% 21/07/05.

load thedate datestring, load(['orbvarsmid' datestring]);

m = 100; M = 6e24; gamma = 6.673e-26; r0 = 5; r1 = 6;
phid0 = sqrt((gamma*M)/r0^3);
T0 = 2*pi*sqrt(r0^3/(gamma*M)); T1 = 2*pi*sqrt(r1^3/(gamma*M));

N = [Nrange{1}(end); Nrange{2}(end)];

for p = 1:2
    T = p*(T0+T1)/2; leng = length(Nrange{p});
    q{p} = orbRK(xmid{p,leng},m,M,N(p),r0,T,gamma,phid0);
end
```

```
save(['orbvarsRK' datestring],'q'), exit
```

D.2.9 orbsyst.m

```
function f = orbsyst(t,q,pp,m,M,gamma)
%ORBSYST ODE system for orbital transfer problem.
% See also: DOC ODE45
% 15/07/05.

r = q(1); Y = q(3); Z = q(4);
f =  [Y; Z; r*Z^2 - (gamma*M)/r^2; (ppval(pp,t)/m - 2*Y*Z)/r];
```

Appendix E

Numerical Experiments II Code

The methods for each discretization have structure in common. In the dependency trees of Appendix E.1 the wildcard `*` denotes either `mid` or `eE`. Only the stepping solvers are formed differently, corresponding to explicit versus implicit updates. We list these in their respective sections.

Certain routines are called by every type of solver we have coded. These are listed in Appendix E.1.1. We give the explicit Euler and midpoint rule codes in Appendices E.3 and E.4, respectively; we omit any symplectic Euler codes, since they do not play a significant role in Chapter 5.

E.1 Dependency Trees

E.1.1 Common Files

```
curvedpath

GISpaths

pHpp ——— BigGreen ——— Green


pHpq ——— pGreenpx

                  ┌── pGreenpx
pGreenpxTest ──┤
                  └── Green
```

```
                 ┌─ pHpp ─────── BigGreen ── Green
pHppTest ────┤
                 └─ GISHam ── BigGreen ── Green

                 ┌─ pHpq ─────── BigGreen ── Green
pHpqTest ────┤
                 └─ GISHam ── BigGreen ── Green
```

E.1.2 Optimal Control Solver

```
                 ┌─ GIS*init ── curvedpath
                 │
                 │                 ┌─ GISHam
                 ├─ GIS*cost ──┤─ pHpp
GIS*ctrlsol ──┤                 └─ pHpq
                 │
                 ├─ GIS*nlcon ── BigGreen
                 └─ GISpaths
```

E.1.3 Full Nonlinear Solver

```
                 ┌─ GIS*init ── curvedpath
                 │
                 │                 ┌─ pHpp
GIS*sol ──────┤─ GIS*fun ──┤
                 │                 └─ pHpq
                 └─ GISpaths
```

E.2 Common Files

E.2.1 BigGreen.m

```
function G = BigGreen(q)
%BIGGREEN Matrix of Green's functions of biharmonic equation.
% BIGGREEN(Q) returns a 2*SIZE(Q,2)-by-2*SIZE(Q,2) matrix
% diag(Gr,Gr), where the (i,j) entry of Gr is
```

```
%    Gr(i,j) = GREEN(Q(:,i),Q(:,j)),        i,j = 1:nc
% The intention (for use in the GIS problem) is for Q to be a
% 2-by-nc matrix
%    Q = (Q_1,...,Q_nc).
% 20/07/05.

nc = size(q,2);

% Fill up according to following mask:
where = logical(tril(repmat(1,nc,nc),-1));

Q1 = repmat(q(1,:).',1,nc); Q2 = repmat(q(2,:).',1,nc);

% X = (q(:,2),...,q(:,nc),q(:,3),...,q(:,nc),...):
X = [Q1(where).'; Q2(where).'];

Q1 = Q1.'; Q2 = Q2.';

% Y = (q(:,1),...,q(:,1),q(:,2),...,q(:,2),...):
Y = [Q1(where).'; Q2(where).'];

v = Green([X,q],[Y,q]);
Gr = zeros(nc);
Gr(where) = v(1:end-nc);
Gr = Gr.';
Gr(where) = v(1:end-nc);

% Diagonal entries:
Gr(1:(nc+1):nc^2) = v(end-nc+(1:nc));

Z = zeros(size(Gr)); G = [Gr, Z; Z, Gr];
```

E.2.2 curvedpath.m

```
function [x,y] = curvedpath(x0,y0,x1,y1,N)
%CURVEDPATH Curved initial path for GIS problem.
```

```
% [X,Y] = CURVEDPATH(X0,Y0,X1,Y1,N) generates curved path between
% given initial and final positions.
% Based on implementation by Twining, Marsland & Taylor.
% INPUT
%       X0 Initial x value.
%       Y0 Initial y value.
%       X1 Final x value.
%       Y1 Final y value.
%       N Number of time-steps.
% OUTPUT
%       X (N+1)-vector of x coords of analytically computed
%           curved path between X0 and X1.
%       Y (N+1)-vector of y coords of analytically computed
%           curved path between Y0 and Y1.
% 20/07/05.

theta0 = atan2(y0,x0); theta1 = atan2(y1,x1);
theta2 = atan2((y1-y0),(x1-x0));
r0 = sqrt(x0.^2 + y0.^2); r1 = sqrt(x1.^2 + y1.^2);

if ((abs(theta2 - theta1) < 1e-3) || ...
        (abs((theta2 - theta1) - pi) < 1e-3) || ...
        (abs((theta2 - theta1) + pi) < 1e-3))
    % Initial path is along a diameter.
    B = atanh(r0); A = atanh(r1) - B;
    tsample = 0:1/N:1;
    r = tanh(A*tsample + B);
    [x,y] = pol2cart(theta0,r);
    x(1) = x0; x(N+1) = x1;
    y(1) = y0; y(N+1) = y1;
else
    nsample = 1000;
    R0 = (1 + r0^2)/(2*r0); R1 = (1 + r1^2)/(2*r1);
    alpha = atan2(R0*cos(theta1) - R1*cos(theta0), ...
        R1*sin(theta0) - R0*sin(theta1));
    k0 = R0/cos(theta0 - alpha); k1 = R1/cos(theta1 - alpha);
```

```
k = mean([k0 k1]);

theta0 = atan2(sin(theta0),cos(theta0));
theta1 = atan2(sin(theta1),cos(theta1));

if ((theta0 < theta1) && ((theta1 - theta0) < pi))
    theta = theta0:(theta1 - theta0)/(nsample - 1):theta1;
elseif ((theta0 < theta1) && ((theta1 - theta0) >= pi))
    theta = theta1:(2*pi - theta1 + theta0)/(nsample - 1): ...
        2*pi + theta0;
    theta = theta(end:-1:1);
elseif  ((theta1 < theta0) && ((theta0 - theta1) < pi))
    theta = theta1:(theta0 - theta1)/(nsample - 1):theta0;
    theta = theta(end:-1:1);
else
    theta = theta0:(2*pi - theta0 + theta1)/(nsample - 1): ...
        2*pi+theta1;
end

if ((theta0 > pi) && (theta1 < pi))
    theta = theta0 - 2*pi: ...
        (theta1 - theta0 + 2*pi)/(nsample - 1):theta1;
end

if ((theta0 < pi) && (theta1 > pi))
    theta = theta1 - 2*pi: ...
        (theta0 - theta1 + 2*pi)/(nsample - 1):theta0;
    theta = theta(end:-1:1);
end

r = k*cos(theta - alpha) - sqrt((k*cos(theta - alpha)).^2 - 1);

dummy = cot(theta-alpha)*sqrt(k^2 - 1);
tau = atanh(dummy);

if ~isreal(tau)
```

```
        tau = atanh(1./dummy);
    end

    a = 1/(tau(end) - tau(1)); b = -a*tau(1);
    tau = a*tau + b;
    tsample = 0:1/N:1;
    thetaout = interp1(tau,theta,tsample);

    if any(isnan(thetaout))
        where = find(tau == max(tau));
        thetaout(length(tsample)) = theta(where);
    end

    rout = interp1(theta,r,thetaout);
    [x,y] = pol2cart(thetaout,rout);
end
```

E.2.3 GISHam.m

```
function H = GISHam(q,p)
%GISHAM Hamiltonian for GIS problem.
% 20/07/05.

nc = size(q,2);

G = BigGreen(q); G = G(1:nc,1:nc);

% Make the matrix dotmat(i,j) = dot(p(:,i),p(:,j)).
P1 = repmat(p(1,:).',1,nc); P2 = repmat(p(2,:).',1,nc);

% Fill up according to following mask:
where = logical(tril(repmat(1,nc,nc),-1));

% X = (p(:,2), ... , p(:,nc), p(:,3), ... , p(:,nc) , ... ):
X = [P1(where).'; P2(where).'];
```

```
P1 = P1.'; P2 = P2.';

% Y = (p(:,1), ... , p(:,1), p(:,2), ... , p(:,2), ... ):
Y = [P1(where).'; P2(where).'];

v = dot([X,p],[Y,p]);
dotmat = zeros(nc);
dotmat(where) = v(1:end-nc);
dotmat = dotmat.';
dotmat(where) = v(1:end-nc);

% Diagonal entries:
dotmat(1:(nc+1):nc^2) = v(end-nc+(1:nc));

H = 0.5*sum(sum(dotmat.*G));
```

E.2.4 GISpaths.m

```
function legh = GISpaths(z,N,nc,titlestr)
%GISPATHS Plots control-point paths for GIS problem.
% LEGH = GISPATHS plots the unit circle, initial and final control
% points, and the path between these points as dictated by Z.
%   INPUT:
%       Z Vector containing control-point paths.
%       N Number of time-steps.
%       NC Number of control points.
%       TITLESTR String for title label.
%   OUTPUT:
%       LEGH Legend handle.
% 20/07/05.

if nargin < 4, titlestr = []; end

% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k:
q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);
```

```
q1 = q(1:2,:); qNp = q(2*N+(1:2),:);
qi = reshape(z(1:2*(N+1)*nc),2,(N+1)*nc);

% qi1(i,j) = q_{i,j}(1), qi2(i,j) = q_{i,j}(2):
qi1 = reshape(qi(1,:),N+1,nc);
qi2 = reshape(qi(2,:),N+1,nc);

% Plot initial, final points, initial paths:
plot(q1(1,:),q1(2,:),'x',qNp(1,:),qNp(2,:),'.',qi1,qi2,'b-');
title(titlestr)
legh = legend('Start','End','Location','Best');
rectangle('Position', [-1 -1 2 2], 'Curvature', [1 1])
axis square, axis off, box off
```

E.2.5 Green.m

```
function f = Green(x,y)
%GREEN Green's function of biharmonic equation.
% F = GREEN(X,Y) computes the values of the Green's function G at
% the vectors X and Y, where G is such that
%   nabla^2 G = delta(X - Y).
% If X and Y are arrays, returns value along first nonsingleton
% dimension.
% INPUT
%       X,Y Assumed REAL arrays of the same SIZE.
% 27/07/05.

% Check dimensions
if any(size(x) ~= size(y)),
   error('Green:InputSizeMismatch', 'X and Y must be same size.');
end

s = sum((x-y).^2);

% Get indices for where s ~= 0, and the complementary indices:
ind = find(s ~= 0); compind = find(s == 0);
```

```
if isempty(ind)
    f = 0.5*(sum(x.^2) - 1).^2;
elseif isempty(compind)
    A = sqrt(sum(x.^2).*sum(y.^2) - 2*dot(x,y) + 1)./sqrt(s);
    f = 0.5*(sum(x.^2)-1).*(sum(y.^2)-1) - s.*log(A);
else
    A = sqrt(sum(x(:,ind).^2).*sum(y(:,ind).^2) - ...
        2*dot(x(:,ind),y(:,ind)) + 1)./sqrt(s(:,ind));
    f(ind) = 0.5*(sum(x(:,ind).^2)-1).*(sum(y(:,ind).^2)-1) - ...
        s(:,ind).*log(A);
    f(compind) = 0.5*(sum(x(:,compind).^2) - 1).^2;
end
```

E.2.6 pGreenpx.m

```
function D = pGreenpx(x,y)
%PGREENPX Partial derivative w.r.t. x of function in GREEN.M
% Analytically computed p[G(x,y)] / px for the
% biharmonic equation's Green's function.
% INPUT
%       X,Y Assumed REAL arrays of equal SIZE.
% 27/07/05.
% Tested: 27/07/05.

% Check dimensions
if any(size(x) ~= size(y)),
   error('pGreenpx:InputSizeMismatch', 'X and Y must be same size.');
end

s = sum((x-y).^2);

% Get indices for where s ~= 0, and the complementary indices:
ind = find(s ~= 0); compind = find(s == 0);

if isempty(ind)
```

```
    D = 2*x.*(repmat(sum(x.^2),2,1) - 1);
elseif isempty(compind)
    A = sqrt(sum(x.^2).*sum(y.^2) - 2*dot(x,y) + 1)./sqrt(s);

    % Repmat A to ensure correct products:
    repA = repmat(A,2,1);

    term = x.*repmat(sum(y.^2),2,1) - y;
    D = term - 2*(x-y).*log(repA) - term./(repA.^2);
else
    ex = x(:,ind); wy = y(:,ind);
    A = sqrt(sum(ex.^2).*sum(wy.^2) - 2*dot(ex,wy) + 1)./sqrt(s(:,ind));
    repA = repmat(A,2,1);
    term = ex.*repmat(sum(wy.^2),2,1) - wy;
    D(:,ind) = term - 2*(ex-wy).*log(repA) - term./(repA.^2);
    D(:,compind) = 2*x(:,compind).* ...
        (repmat(sum(x(:,compind).^2),2,1) - 1);
end
```

E.2.7 pGreenpxTest.m

```
%PGREENPXTEST Test script for Green's function partial deriv.
% 20/07/05.

% Test for vectors:
x = rand(2,1); y = rand(2,1);

D = pGreenpx(x,y);

% D agrees with limit definition of the partial:
ep1 = [eps; 0]; ep2 = [0; eps];
for i = 1:10:1000
    X((i-1)/10+1) = (Green(x+i*ep1,y) - Green(x,y))/(i*eps);
    Y((i-1)/10+1) = (Green(x+i*ep2,y) - Green(x,y))/(i*eps);
end
figure
```

```
plot(1:10:1000,X(end:-1:1),'b-',1:10:1000, ...
    repmat(D(1),100,1),'r--'), hold on
plot(1:10:1000,Y(end:-1:1),'b-',1:10:1000, ...
    repmat(D(2),100,1),'r--')
hold off

% Test diagonal entries:
D = pGreenpx(x,x); Dc = 2*(norm(x)^2-1)*x;
epsil = eps;

if ~(any(abs(D - Dc) < epsil))
    error('pGreenpxTest:VectorFail', 'Failed for vector X,X')
end

% Test for arrays:
x = rand(2,10); y = rand(2,10);

D = pGreenpx(x,y);

for k = 1:10
    Dk = pGreenpx(x(:,k),y(:,k));
    if ~(any(abs(D(:,k) - Dk) < epsil))
        error('pGreenpxTest:MatrixFail', 'Failed for matrix X,Y')
    end
end

D = pGreenpx([x(:,1:end-1), y(:,end)],y);

for k = 1:9
    Dk = pGreenpx(x(:,k),y(:,k));
    if ~(any(abs(D(:,k) - Dk) < epsil))
        error('pGreenpxTest:MatrixFail', 'Failed for matrix X,Y')
    end
end

Dk = pGreenpx(y(:,10),y(:,10));
```

```
if ~(any(abs(D(:,10) - Dk) < epsil))
    error('pGreenpxTest:MatrixFail', 'Failed for matrix X,Y')
end
```

E.2.8 pHpp.m

```
function D = pHpp(q,p)
%PHPP Partial derivative of GIS Hamiltonian w.r.t P.
% D = PHPP(Q,P) computes pH / pP (Q,P) for the GIS Hamiltonian.
% In this case we assume that
%        Q = (q_1(1),...,q_nc(1);
%             q_1(2),...,q_nc(2)),
% and that P is with p_i, mutatis mutandis.
% 20/07/05.
% Tested: 20/07/05.

% Check dimensions
if any(size(q)~=size(p)),
   error('pHpp:InputSizeMismatch', 'Q and P must be same size.');
end

nc = size(q,2);
G = BigGreen(q);

% D(:,i) = sum_j(m(:,j)*Green(q(:,i),q(:,j))).
%
% G*reshape(p.',2*nc,1) = (D(1,1);D(2,1);...;D(nc,1);D(1,2);...;D(nc,2))
D = reshape(G*reshape(p.',2*nc,1),nc,2).';
```

E.2.9 pHppTest.m

```
%PHPPTEST Test script for GIS Hamilton p-partial.
% 20/07/05.

x = rand(2,10); y = rand(2,10);
nc = size(x,2);
```

```
D = pHpp(x,y);

% D agrees with limit definition of the partial:
for k = 1:nc
    ep1 = zeros(2,nc); ep1(1,k) = eps;
    ep2 = zeros(2,nc); ep2(2,k) = eps;
    for i = 1:10:1000
        X((i-1)/10+1) = (GISHam(x,y+i*ep1) - GISHam(x,y))/(i*eps);
        Y((i-1)/10+1) = (GISHam(x,y+i*ep2) - GISHam(x,y))/(i*eps);
    end
    figure
    plot(1:10:1000,X(end:-1:1),'b-',1:10:1000, ...
        repmat(D(1,k),100,1),'r--'), hold on
    plot(1:10:1000,Y(end:-1:1),'b-',1:10:1000, ...
        repmat(D(2,k),100,1),'r--')
end
hold off

D = pHpp(x,x);

% D agrees with limit definition of the partial:
for k = 1:nc
    ep1 = zeros(2,nc); ep1(1,k) = eps;
    ep2 = zeros(2,nc); ep2(2,k) = eps;
    for i = 1:10:1000
        X((i-1)/10+1) = (GISHam(x,x+i*ep1) - GISHam(x,x))/(i*eps);
        Y((i-1)/10+1) = (GISHam(x,x+i*ep2) - GISHam(x,x))/(i*eps);
    end
    figure
    plot(1:10:1000,X(end:-1:1),'b-',1:10:1000, ...
        repmat(D(1,k),100,1),'r--'), hold on
    plot(1:10:1000,Y(end:-1:1),'b-',1:10:1000, ...
        repmat(D(2,k),100,1),'r--')
end
hold off
```

E.2.10 pHpq.m

```
function D = pHpq(q,p)
%PHPQ Partial derivative of GIS Hamiltonian w.r.t q.
% D = PHPQ(Q,P) computes pH / pQ (Q,P) for the GIS Hamiltonian.
% In this case we assume that
%         Q = (q_1(1),...,q_nc(1);
%               q_1(2),...,q_nc(2)),
% and that P is with p_i, mutatis mutandis.
% 20/07/05.
% Tested: 20/07/05.

% Check dimensions
if any(size(q)~=size(p)),
   error('pHpq:InputSizeMismatch', 'Q and P must be same size.');
end

nc = size(q,2);

% Make the matrices Pxk(i,j) = pGreenpx(q(:,i),q(:,j))(k).
% Fill up according to following mask:
where = logical(tril(repmat(1,nc,nc),-1));

Q1 = repmat(q(1,:).',1,nc); Q2 = repmat(q(2,:).',1,nc);

% X = (q(:,2),...,q(:,nc),q(:,3),...,q(:,nc),...):
X = [Q1(where).'; Q2(where).'];

Q1 = Q1.'; Q2 = Q2.';

% Y = (q(:,1),...,q(:,1),q(:,2),...,q(:,2),...):
Y = [Q1(where).'; Q2(where).'];

% pGreenpx is not symmetric:
V = pGreenpx([X,Y,q],[Y,X,q]);
```

```
v1 = V(1,:); v2 = V(2,:);
Px1 = zeros(nc); Px2 = zeros(nc);

Px1(where) = v1(1:0.5*nc*(nc-1));
Px1 = Px1.';
Px1(where) = v1((1:0.5*nc*(nc-1))+0.5*nc*(nc-1));
Px1 = Px1.';

% Diagonal entries:
diagon = 1:nc+1:nc^2;
Px1(diagon) = v1(end-nc+(1:nc));

Px2(where) = v2(1:0.5*nc*(nc-1));
Px2 = Px2.';
Px2(where) = v2((1:0.5*nc*(nc-1))+0.5*nc*(nc-1));
Px2 = Px2.';

% Diagonal entries:
Px2(diagon) = v2(end-nc+(1:nc));

% Make the matrix dotmat(i,j) = dot(p(:,i),p(:,j)):
P1 = repmat(p(1,:).',1,nc); P2 = repmat(p(2,:).',1,nc);

% X = (p(:,2), ... , p(:,nc), p(:,3), ... , p(:,nc), ... ):
X = [P1(where).'; P2(where).'];

P1 = P1.'; P2 = P2.';

% Y = (p(:,1), ... , p(:,1), p(:,2), ... , p(:,2), ... ):
Y = [P1(where).'; P2(where).'];

v = dot([X,p],[Y,p]);
dotmat = zeros(nc);
dotmat(where) = v(1:end-nc);
dotmat = dotmat.';
dotmat(where) = v(1:end-nc);
```

```
% Diagonal entries:
dotmat(diagon) = v(end-nc+(1:nc));

% D(:,i) = sum_j(p(:,i)'*p(:,j)*pGreenpx(q(:,i),q(:,j)))
%                 - 0.5*p(:,i)'*p(:,i)*pGreenpx(q(:,i),q(:,i)).
% sum(dotmat.*Px1,2)(i) = sum_j(dot(pi,pj)*Px1(i,j)).
D = [sum(dotmat.*Px1,2).' - 0.5*dotmat(diagon).*Px1(diagon); ...
     sum(dotmat.*Px2,2).' - 0.5*dotmat(diagon).*Px2(diagon)];
```

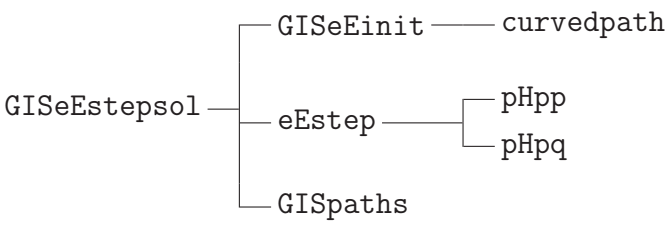
E.2.11 pHpqTest.m

```
%PHPQTEST Test script for GIS Hamilton q-partial.
% 20/07/05.

x = rand(2,10); y = rand(2,10);
nc = size(x,2);

D = pHpq(x,y);

% D agrees with limit definition of the partial:
for k = 1:nc
    ep1 = zeros(2,nc); ep1(1,k) = eps;
    ep2 = zeros(2,nc); ep2(2,k) = eps;
    for i = 1:10:1000
        X((i-1)/10+1) = (GISHam(x+i*ep1,y) - GISHam(x,y))/(i*eps);
        Y((i-1)/10+1) = (GISHam(x+i*ep2,y) - GISHam(x,y))/(i*eps);
    end
    figure
    plot(1:10:1000,X(end:-1:1),'b-',1:10:1000, ...
        repmat(D(1,k),100,1),'r--'), hold on
    plot(1:10:1000,Y(end:-1:1),'b-',1:10:1000, ...
        repmat(D(2,k),100,1),'r--')
end
hold off
```

```
D = pHpq(x,x);

% D agrees with limit definition of the partial:
for k = 1:nc
    ep1 = zeros(2,nc); ep1(1,k) = eps;
    ep2 = zeros(2,nc); ep2(2,k) = eps;
    for i = 1:10:1000
        X((i-1)/10+1) = (GISHam(x+i*ep1,x) - GISHam(x,x))/(i*eps);
        Y((i-1)/10+1) = (GISHam(x+i*ep2,x) - GISHam(x,x))/(i*eps);
    end
    figure
    plot(1:10:1000,X(end:-1:1),'b-',1:10:1000, ...
        repmat(D(1,k),100,1),'r--'), hold on
    plot(1:10:1000,Y(end:-1:1),'b-',1:10:1000, ...
        repmat(D(2,k),100,1),'r--')
end
hold off
```

E.3 Explicit Euler

E.3.1 Dependency Tree: Stepping Nonlinear Solver



E.3.2 eEstep.m

```
function [y,q,p] = eEstep(p1,N,nc,q1,qNp,solv)
%EESTEP Stepping explicit Euler function for GIS problem.
% [Y,Q,P] = EESTEP(P1,N,NC,Q1,QNP,SOLV) computes values of position
% and momenta at discretization points using explicit Euler updates.
% Also determines the deviation between computed final position and
% given vector QNP of final positions.
% INPUT
```

```
%   P1 2-by-NC vector of initial momenta.
%   N Number of time-steps.
%   NC Number of control points.
%   Q1 2-by-NC vector of initial positions.
%   QNP 2-by-NC vector of final positions.
%   SOLV 'fsol' for FSOLVE solution or 'fmin' for FMINSEARCH.
%        Defaults to 'fsol'.
% OUTPUT
%   Y Deviation from QNP of computed final positions.
%   Q Cell array of intermediate position vectors.
%   P Cell array of intermediate momentum vectors,
% 02/09/05.

if nargin < 6, solv = 'fsol'; end

q = cell(N+1,1); q{1} = q1; p = cell(N+1,1); p{1} = p1;
h = 1/N;

for k = 1:N
    qk = reshape(q{k},2,nc); pk = reshape(p{k},2,nc);
    q{k+1} = reshape(qk+h*pHpp(qk,pk),2*nc,1);
    p{k+1} = reshape(pk-h*pHpq(qk,pk),2*nc,1);
end

if strcmp(solv,'fmin')
    y = sum((q{N+1} - qNp).^2);
elseif strcmp(solv,'fsol')
    y = q{N+1} - qNp;
else
    error('eEstep:IllegalArgString','Illegal argument string.')
end
```

E.3.3 GISeEcost.m

```
function [J,G] = GISeEcost(z,N,nc)
%GISEECOST Explicit Euler discrete cost function for GIS problem.
```

```
% [J,G] = GISEECOST(Z,N,NC) returns the value of the discretized
% bending energy for the GIS problem.
% INPUT
%   Z 4*(N+1)*nc vector of state variables
%           Z = (q_{1,1}; ... ; q_{1,N+1}; ... ; q_{nc,1}; q_{nc,N+1};
%                p_{1,1}; ... ; p_{nc,N}).
%   N Number of time-steps.
%   NC Number of control points.
% OUTPUT
%   J Cost at Z.
%   G Gradient of J at Z.
% 02/09/05.
% (DerivativeCheck confirms gradient OK.)

z = z(:); % Just to be sure...

q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);
p = reshape(z(2*(N+1)*nc + (1:2*(N+1)*nc)),2*(N+1),nc);

% Make vector of GISHam on q and p.
D = zeros(N,1);
for k = 1:N
    ind = 2*(k-1)+(1:2);
    D(k) = 2*GISHam(q(ind,:),p(ind,:));
end

h = 1/N; J = h*sum(D);

if nargout > 1
    G = zeros(4*(N+1)*nc,1);
    Dq = zeros(2*(N+1),nc); Dp = zeros(2*(N+1),nc);

    for k = 1:N
        ind = 2*(k-1)+(1:2);
        Dq(ind,:) = pHpq(q(ind,:),p(ind,:));
        Dp(ind,:) = pHpp(q(ind,:),p(ind,:));
```

```
    end

    G(1:2*(N+1)*nc) = 2*h*reshape(Dq,2*(N+1)*nc,1);
    G(2*(N+1)*nc+(1:2*(N+1)*nc)) = 2*h*reshape(Dp,2*(N+1)*nc,1);
end
```

E.3.4 GISeEctrlsol.m

```
function [z,z0,output,time] = GISeEctrlsol(N,nc,plots)
%GISEECTRLSOL Optimal control explicit Euler solution of GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISEECTRLSOL(N,NC,PLOTS) uses
% FMINCON to minimize the bending-energy cost for the GIS
% problem, subject to EE constraints for the Green's function
% coefficients p_j.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FMINCON.
%   TIME Time taken to perform solve.
% 15/08/05.

if nargin < 3, plots = 'off'; end

rand('state',nc);

[z0,q1,qNp,Aeq,beq] = GISeEinit(N,nc);

options = optimset('Disp', 'iter', ...
    'GradObj', 'on', ...
    'LargeScale', 'off');
```

```
tic;
[z,fval,exitflag,output] = ...
 fmincon(@(z)GISeEcost(z,N,nc),z0,[],[],Aeq,beq,[],[], ...
 @(z)GISeEnlcon(z,N,nc),options);
time = toc;

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISeEctrlsol:IllegalArgString','Illegal argument string.')
end
```

E.3.5 GISeEfun.m

```
function D = GISeEfun(z,N,nc,q1,qNp)
%GISEEFUN Discrete explicit Euler Hamilton's equations for GIS problem.
% D = GISEEFUN(Z,N,NC,Q1,QNP) returns 4*N*nc-vector D consisting
% of the discrete Hamilton's equations for Q, followed by those for P.
% INPUT
%   Z 4*N*nc-vector of state variables: control point paths without
%       initial and final positions, then momenta.
%       Z = (q_{1,2}; ... ; q_{1,N}; ... ; q_{nc,2}; q_{nc,N};
%           p_{1,1}; ... ; p_{nc,N+1}).
%   N Number of time-steps.
%   NC Number of control points.
%   Q1 2-by-NC vector of initial positions.
%   QNP 2-by-NC vector of final positions.
% OUTPUT
%   D 4*N*nc-vector, where D(1:2*N*nc) is Hamilton's equation for
%       qdot, D(2*N*nc+(1:2*N*nc)) is that for pdot.
% 20/07/05.

z = z(:); h = 1/N;
```

```
D = zeros(4*N*nc,1);

% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k.
% q(2*(k-1)+(1:2),:) = (p_{1,k}, ... , p_{nc,k}) = : p_k:
q = reshape(z(1:2*(N-1)*nc),2*(N-1),nc);
q = [q1; q; qNp];
p = reshape(z(2*(N-1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);

% vhalf(2*(k-1)+(1:2),:) = (q_{k+1}-q_k)/h:
vhalf = (q(3:2*(N+1),:) - q(1:2*N,:))/h;
% ahalf(2*(k-1)+(1:2),:) = (p_{k+1}-p_k)/h:
ahalf = (p(3:2*(N+1),:) - p(1:2*N,:))/h;

for k = 1:N
    ind = 2*(k-1)+(1:2);
    qk = q(ind,:); pk = p(ind,:);
    D(2*nc*(k-1)+(1:2*nc)) = reshape(vhalf(ind,:) - pHpp(qk,pk),2*nc,1);
    D(2*N*nc+2*nc*(k-1)+(1:2*nc)) = ...
        reshape(ahalf(ind,:) + pHpq(qk,pk),2*nc,1);
end
```

E.3.6 GISeEinit.m

```
function [z0,q1,qNp,Aeq,beq] = GISeEinit(N,nc,varargin)
%GISEEINIT Initializes values for explicit Euler GIS problem.
% [Z0,Q1,QNP,AEQ,BEQ] = GISEEINIT(N,NC) generates control point data
% Q1 and QNP randomly.
% [Z0,Q1,QNP,AEQ,BEQ] = GISEEINIT(N,NC,Q1,QNP) uses predetermined
% control point data Q1 and QNP.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   OPTIONAL
%       Q1 2-by-NC vector of initial positions.
%       QNP 2-by-NC vector of final positions.
% OUTPUT
```

```
%   Z0 Initial guess for paths between control points.
%   Q1 2-by-NC vector of initial positions.
%   QNP 2-by-NC vector of final positions.
%   OPTIONAL
%       AEQ Linear equality constraints matrix.
%       BEQ Linear equality constraints vector.
% 20/07/05.

if nargin > 2
    q1 = varargin{1}; qNp = varargin{2};
else
    % Generate initial, final points.
    r0 = rand(nc,1);
    % Shift points that are close to the boundary:
    f = find(r0 >= 0.9); r0(f) = r0(f) - 0.4;

    r1 = r0 + 0.1*rand(nc,1);
    f = find(r1 >= 0.9); r1(f) = r1(f) - 0.4;

    theta0 = 2*pi*rand(nc,1);
    theta1 = theta0 + 0.1*pi*rand(nc,1);

    q1 = zeros(2,nc); qNp = zeros(2,nc);
    [q1(1,:), q1(2,:)] = pol2cart(theta0, r0);
    [qNp(1,:), qNp(2,:)] = pol2cart(theta1, r1);
end

z0 = zeros(4*(N+1)*nc,1);

for i = 1:nc
    [x,y] = curvedpath(q1(1,i),q1(2,i),qNp(1,i),qNp(2,i),N);
    z0(2*(N+1)*(i-1)+(1:2:2*(N+1))) = x;
    z0(2*(N+1)*(i-1)+(2:2:2*(N+1))) = y;
end

% Compute initial guess for ps from q.
```

```
% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k:
q = reshape(z0(1:2*(N+1)*nc),2*(N+1),nc);

h = 1/N; vhalf = (q(3:4,:) - q(1:2,:))/h;

p = zeros(2*(N+1),nc);
pk = BigGreen(q(1:2,:))\reshape(vhalf(1:2,:).',2*nc,1);
p(1:2,:) = reshape(pk,nc,2).';
for k = 1:N
    ind = 2*(k-1)+(1:2);
    p(ind+2,:) = p(ind,:)-h*pHpq(q(ind,:),p(ind,:));
end

z0(2*(N+1)*nc + (1:2*(N+1)*nc)) = reshape(p,2*(N+1)*nc,1);

if nargout > 3
    % Linear equality constraints arrays.
    Aeq = zeros(4*nc,4*(N+1)*nc);

    for k = 1:nc
        Aeq(2*k-1,2*(k-1)*(N+1)+1) = 1;
        Aeq(2*k,2*(k-1)*(N+1)+2) = 1;
        Aeq(2*nc+2*k-1,2*k*(N+1)-1) = 1;
        Aeq(2*nc+2*k,2*k*(N+1)) = 1;
    end

    beq = Aeq*z0;
end
```

E.3.7 GISeEnlcon.m

```
function [C,Ceq] = GISeEnlcon(z,N,nc)
%GISEENLCON Explicit Euler Legendre transform for GIS problem.
% INPUT
%   Z 4*(N+1)*nc vector of state variables
%           Z = (q_{1,1}; ... ; q_{1,N+1}; ... ; q_{nc,1}; q_{nc,N+1};
```

```
%                    p_{1,1}; ... ; p_{nc,N}).
%   N Number of time-steps.
%   NC Number of control points.
% 20/07/05.

z = z(:); % Just to be sure...

C = []; Ceq = zeros(2*N*nc,1);

q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);
p = reshape(z(2*(N+1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);
h = 1/N; vhalf = (q(3:2*(N+1),:) - q(1:2*N,:))/h;

for k = 1:N
    ind = 2*(k-1)+(1:2);
    Ceq(2*nc*(k-1)+(1:2*nc)) = ...
        BigGreen(q(ind,:))*reshape(p(ind,:).',2*nc,1) - ...
        reshape(vhalf(ind,:).',2*nc,1);
end
```

E.3.8 GISeEsol.m

```
function [z,z0,output,time] = GISeEsol(N,nc,plots)
%GISEESOL Explicit Euler full nonlinear solve for GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISEESOL(N,NC,PLOTS) uses
% FSOLVE to solve the explicit Euler discretized Hamilton's equations.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FSOLVE.
%   TIME Time taken to perform solve.
```

```
% 25/07/05.

if nargin < 3, plots = 'off'; end

rand('state',nc);

[z0,q1,qNp] = GISeEinit(N,nc);

options = optimset('Disp', 'iter', ...
    'LargeScale', 'off');

% Remove q1 and qNp from z0:
q0 = z0(1:2*(N+1)*nc); Q0 = reshape(q0,2*(N+1),nc);
Q0 = reshape(Q0(3:end-2,:),2*(N-1)*nc,1);
Z0 = [Q0; z0(2*(N+1)*nc+(1:2*(N+1)*nc))];

tic;
[z,fval,exitflag,output] = ...
    fsolve(@(z)GISeEfun(z,N,nc,q1,qNp),Z0,options);
time = toc;

% Get q back into correct form:
q = reshape([q1;reshape(z(1:2*(N-1)*nc),2*(N-1),nc);qNp],2*(N+1)*nc,1);
p = z(2*(N-1)*nc+(1:2*(N+1)*nc));
z = [q; p];

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISeEsol:IllegalArgString','Illegal argument string.')
end
```

E.3.9 GISeEstepsol.m

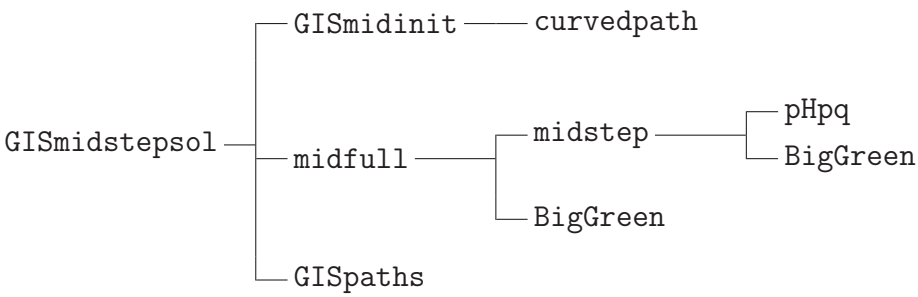
```
function [z,z0,output,time] = GISeEstepsol(N,nc,solv,plots)
%GISEESTEPSOL Explicit Euler stepping nonlinear solve for GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISEESTEPSOL(N,NC,SOLV,PLOTS) finds the optimal
% initial momenta such that the final positions they generate are
% close to the specified final control points.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   SOLV 'fsol' for FSOLVE solution or 'fmin' for FMINSEARCH.
%        Defaults to 'fsol'. FSOLVE solves for the optimal final
%        control points, while FMINSEARCH minimizes the deviation of
%        the final points generated by some initial momenta from the
%        final control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%        'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FSOLVE.
%   TIME Time taken to perform solve.
% 15/08/05.

nin = nargin;

if nin < 4, plots = 'off';
    if nin < 3, solv = 'fsol';
    end, end

rand('state',nc);

[z0,q1,qNp] = GISeEinit(N,nc);

if strcmp(solv,'fmin'), 0;
elseif strcmp(solv,'fsol')
```

```
    options = optimset('Disp', 'iter', ...
        'LargeScale', 'off');
else
    error('GISeEstepsol:IllegalArgString','Illegal argument string.')
end

% Extract initial momenta from z0:
p0 = z0(2*(N+1)*nc+(1:2*(N+1)*nc));
P0 = reshape(p0,2*(N+1),nc);
p10 = reshape(P0(1:2,:),2*nc,1);

q1 = reshape(q1,2*nc,1);
qNp = reshape(qNp,2*nc,1);

if strcmp(solv,'fmin')
    tic;
    [p1,fval,exitflag,output] = ...
        fminsearch(@(p1)eEstep(p1,N,nc,q1,qNp,solv),p10,options);
    time = toc;
else
    tic;
    [p1,fval,exitflag,output] = ...
        fsolve(@(p1)eEstep(p1,N,nc,q1,qNp,solv),p10,options);
    time = toc;
end

% Return intermediate q and p generated by the optimal initial momenta:
[y,Q,P] = eEstep(p1,N,nc,q1,qNp);

% Put values in right places:
q = []; p = [];
for i = 1:nc
    for k = 1:N+1
        q = [q; Q{k}(2*(i-1)+(1:2))];
        p = [p; P{k}(2*(i-1)+(1:2))];
    end
```

```
end
z = [q; p];

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISeEstepsol:IllegalArgString','Illegal argument string.')
end
```

E.4 Implicit Midpoint Rule

E.4.1 Dependency Tree: Stepping Nonlinear Solver



E.4.2 GISmidcost.m

```
function [J,G] = GISmidcost(z,N,nc)
%GISMIDCOST Midpoint rule discrete cost function for GIS problem.
% [J,G] = GISMIDCOST(Z,N,NC) returns the value of the discretized
% bending energy for the GIS problem.
% INPUT
%   Z 4*(N+1)*nc vector of state variables
%           Z = (q_{1,1}; ... ; q_{1,N+1}; ... ; q_{nc,1}; q_{nc,N+1};
%                p_{1,1}; ... ; p_{nc,N}).
%   N Number of time-steps.
%   NC Number of control points.
% OUTPUT
```

```
%   J Cost at Z.
%   G Gradient of J at Z.
% 02/09/05.
% (DerivativeCheck confirms gradient OK.)

z = z(:); % Just to be sure...

q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);
qhalf = (q(3:2*(N+1),:)+q(1:2*N,:))/2;
p = reshape(z(2*(N+1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);
phalf = (p(3:2*(N+1),:)+p(1:2*N,:))/2;

% Make vector of GISHam on q and p.
D = zeros(N,1);
for k = 1:N
    ind = 2*(k-1)+(1:2);
    D(k) = 2*GISHam(qhalf(ind,:),phalf(ind,:));
end

h = 1/N; J = h*sum(D);

if nargout > 1
    G = zeros(4*(N+1)*nc,1);
    Dq = zeros(2*(N+1),nc); Dp = zeros(2*(N+1),nc);

    pHpqm = pHpq(qhalf(1:2,:),phalf(1:2,:));
    pHppm = pHpp(qhalf(1:2,:),phalf(1:2,:));
    Dq(1:2,:) = 0.5*pHpqm; Dp(1:2,:) = 0.5*pHppm;

    for k = 2:N
        ind = 2*(k-1)+(1:2);
        pHpqplus = pHpq(qhalf(ind,:),phalf(ind,:));
        pHppplus = pHpp(qhalf(ind,:),phalf(ind,:));
        Dq(ind,:) = 0.5*pHpqm + 0.5*pHpqplus;
        Dp(ind,:) = 0.5*pHppm + 0.5*pHppplus;
        pHpqm = pHpqplus; pHppm = pHppplus;
```

```
    end

    Dq(ind+2,:) = 0.5*pHpq(qhalf(end-1:end,:),phalf(end-1:end,:));
    Dp(ind+2,:) = 0.5*pHpp(qhalf(end-1:end,:),phalf(end-1:end,:));

    G(1:2*(N+1)*nc) = 2*h*reshape(Dq,2*(N+1)*nc,1);
    G(2*(N+1)*nc+(1:2*(N+1)*nc)) = 2*h*reshape(Dp,2*(N+1)*nc,1);
end
```

E.4.3 GISmidctrlsol.m

```
function [z,z0,output,time] = GISmidctrlsol(N,nc,plots)
%GISMIDCTRLSOL Optimal control midpoint rule solution of GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISMIDCTRLSOL(N,NC,PLOTS) uses
% FMINCON to minimize the bending-energy cost for the GIS
% problem, subject to midpoint constraints for the Green's function
% coefficients p_j.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FMINCON.
%   TIME Time taken to perform solve.
% 15/08/05.

if nargin < 3, plots = 'off'; end

rand('state',nc);

[z0,q1,qNp,Aeq,beq] = GISmidinit(N,nc);

options = optimset('Disp', 'iter', ...
```

```
    'GradObj', 'on', ...
    'LargeScale', 'off');

tic;
[z,fval,exitflag,output] = ...
 fmincon(@(z)GISmidcost(z,N,nc),z0,[],[],Aeq,beq,[],[], ...
 @(z)GISmidnlcon(z,N,nc),options);
time = toc;

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISmidctrlsol:IllegalArgString','Illegal argument string.')
end
```

E.4.4 GISmidfun.m

```
function D = GISmidfun(z,N,nc,q1,qNp)
%GISMIDFUN Discrete midpoint rule Hamilton's equations for GIS problem.
% D = GISMIDFUN(Z,N,NC,Q1,QNP) returns 4*N*nc-vector D consisting
% of the discrete Hamilton's equations for Q, followed by those for P.
% INPUT
%   Z 4*N*nc-vector of state variables: control point paths without
%       initial and final positions, then momenta.
%       Z = (q_{1,2}; ... ; q_{1,N}; ... ; q_{nc,2}; q_{nc,N};
%           p_{1,1}; ... ; p_{nc,N+1}).
%   N Number of time-steps.
%   NC Number of control points.
%   Q1 2-by-NC vector of initial positions.
%   QNP 2-by-NC vector of final positions.
% OUTPUT
%   D 4*N*nc-vector, where D(1:2*N*nc) is Hamilton's equation for
%       qdot, D(2*N*nc+(1:2*N*nc)) is that for pdot.
```

```
% 20/07/05.

z = z(:); h = 1/N;
D = zeros(4*N*nc,1);

% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k.
% p(2*(k-1)+(1:2),:) = (p_{1,k}, ... , p_{nc,k}) = : p_k:
q = reshape(z(1:2*(N-1)*nc),2*(N-1),nc);
q = [q1; q; qNp];
p = reshape(z(2*(N-1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);

% vhalf(2*(k-1)+(1:2),:) = (q_{k+1}-q_k)/h:
vhalf = (q(3:2*(N+1),:) - q(1:2*N,:))/h;
% ahalf(2*(k-1)+(1:2),:) = (p_{k+1}-p_k)/h:
ahalf = (p(3:2*(N+1),:) - p(1:2*N,:))/h;

qhalf = (q(3:2*(N+1),:) + q(1:2*N,:))/2;
phalf = (p(3:2*(N+1),:) + p(1:2*N,:))/2;

for k = 1:N
    ind = 2*(k-1)+(1:2);
    qhalfk = qhalf(ind,:); phalfk = phalf(ind,:);
    D(2*nc*(k-1)+(1:2*nc)) = ...
        reshape(vhalf(ind,:) - pHpp(qhalfk,phalfk),2*nc,1);
    D(2*N*nc+2*nc*(k-1)+(1:2*nc)) = ...
        reshape(ahalf(ind,:) + pHpq(qhalfk,phalfk),2*nc,1);
end
```

E.4.5 GISmidinit.m

```
function [z0,q1,qNp,Aeq,beq] = GISmidinit(N,nc,varargin)
%GISMIDINIT Initializes values for midpoint rule GIS problem.
% [Z0,Q1,QNP,AEQ,BEQ] = GISMIDINIT(N,NC) generates control point data
% Q1 and QNP randomly.
% [Z0,Q1,QNP,AEQ,BEQ] = GISMIDINIT(N,NC,Q1,QNP) uses predetermined
% control point data Q1 and QNP.
```

```
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   OPTIONAL
%       Q1 2-by-NC vector of initial positions.
%       QNP 2-by-NC vector of final positions.
% OUTPUT
%   Z0 Initial guess for paths between control points.
%   Q1 2-by-NC vector of initial positions.
%   QNP 2-by-NC vector of final positions.
%   OPTIONAL
%       AEQ Linear equality constraints matrix.
%       BEQ Linear equality constraints vector.
% 20/07/05.

if nargin > 2
    q1 = varargin{1}; qNp = varargin{2};
else
    % Generate initial, final points.
    r0 = rand(nc,1);
    % Shift points that are close to the boundary:
    f = find(r0 >= 0.9); r0(f) = r0(f) - 0.4;

    r1 = r0 + 0.1*rand(nc,1);
    f = find(r1 >= 0.9); r1(f) = r1(f) - 0.4;

    theta0 = 2*pi*rand(nc,1);
    theta1 = theta0 + 0.1*pi*rand(nc,1);

    q1 = zeros(2,nc); qNp = zeros(2,nc);
    [q1(1,:), q1(2,:)] = pol2cart(theta0, r0);
    [qNp(1,:), qNp(2,:)] = pol2cart(theta1, r1);
end

z0 = zeros(4*(N+1)*nc,1);
```

```
for i = 1:nc
    [x,y] = curvedpath(q1(1,i),q1(2,i),qNp(1,i),qNp(2,i),N);
    z0(2*(N+1)*(i-1)+(1:2:2*(N+1))) = x;
    z0(2*(N+1)*(i-1)+(2:2:2*(N+1))) = y;
end

% Compute initial guess for ps from q.
% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k:
q = reshape(z0(1:2*(N+1)*nc),2*(N+1),nc);

% qhalf(2*(k-1)+(1:2),:) = (q_{k+1}+q_k)/2:
qhalf = (q(3:2*(N+1),:)+q(1:2*N,:))/2;

% vhalf(2*(k-1)+(1:2),:) = (q_{k+1}-q_k)/h:
h = 1/N; vhalf = (q(3:2*(N+1),:) - q(1:2*N,:))/h;

p = zeros(2*(N+1),nc);
phalf = BigGreen(qhalf(1:2,:))\reshape(vhalf(1:2,:).',2*nc,1);
phalf = reshape(phalf,nc,2).';
p(1:2,:) = h/2*pHpq(qhalf(1:2,:),phalf) + phalf;
p(3:4,:) = 2*phalf - p(1:2,:);
for k = 2:N
    ind = 2*(k-1)+(1:2);
    phalf = BigGreen(qhalf(ind,:))\reshape(vhalf(ind,:).',2*nc,1);
    phalf = reshape(phalf,nc,2).';
    p(ind+2,:) = 2*phalf - p(ind,:);
end

z0(2*(N+1)*nc+(1:2*(N+1)*nc)) = reshape(p,2*(N+1)*nc,1);

if nargout > 3
    % Linear equality constraints arrays.
    Aeq = zeros(4*nc,4*(N+1)*nc);

    for k = 1:nc
        Aeq(2*k-1,2*(k-1)*(N+1)+1) = 1;
```

```
        Aeq(2*k,2*(k-1)*(N+1)+2) = 1;
        Aeq(2*nc+2*k-1,2*k*(N+1)-1) = 1;
        Aeq(2*nc+2*k,2*k*(N+1)) = 1;
    end

    beq = Aeq*z0;
end
```

E.4.6 GISmidnlcon.m

```
function [C,Ceq] = GISmidnlcon(z,N,nc)
%GISMIDNLCON Midpoint rule Legendre transform for GIS problem.
% INPUT
%   Z 4*(N+1)*nc vector of state variables
%           Z = (q_{1,1}; ... ; q_{1,N+1}; ... ; q_{nc,1}; q_{nc,N+1};
%                p_{1,1}; ... ; p_{nc,N}).
%   N Number of time-steps.
%   NC Number of control points.
% 20/07/05.

z = z(:); % Just to be sure...

C = []; Ceq = zeros(2*N*nc,1);

q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);
qhalf = (q(3:2*(N+1),:)+q(1:2*N,:))/2;
p = reshape(z(2*(N+1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);
phalf = (p(3:2*(N+1),:)+p(1:2*N,:))/2;
h = 1/N; vhalf = (q(3:2*(N+1),:) - q(1:2*N,:))/h;

for k = 1:N
    ind = 2*(k-1)+(1:2);
    Ceq(2*nc*(k-1)+(1:2*nc)) = ...
        BigGreen(qhalf(ind,:))*reshape(phalf(ind,:).',2*nc,1) - ...
        reshape(vhalf(ind,:).',2*nc,1);
end
```

E.4.7 GISmidsol.m

```
function [z,z0,output,time] = GISmidsol(N,nc,plots)
%GISMIDSOL Midpoint rule full nonlinear solve for GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISMIDSOL(N,NC,PLOTS) uses
% FSOLVE to solve the midpoint rule discretized Hamilton's equations.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FSOLVE.
%   TIME Time taken to perform solve.
% 25/07/05.

if nargin < 3, plots = 'off'; end

rand('state',nc);

[z0,q1,qNp] = GISmidinit(N,nc);

options = optimset('Disp', 'iter', ...
    'LargeScale', 'off');

% Remove q1 and qNp from z0:
q0 = z0(1:2*(N+1)*nc); Q0 = reshape(q0,2*(N+1),nc);
Q0 = reshape(Q0(3:end-2,:),2*(N-1)*nc,1);
Z0 = [Q0; z0(2*(N+1)*nc+(1:2*(N+1)*nc))];

tic;
[z,fval,exitflag,output] = ...
    fsolve(@(z)GISmidfun(z,N,nc,q1,qNp),Z0,options);
time = toc;
```

```
% Get q back into correct form:
q = reshape([q1;reshape(z(1:2*(N-1)*nc),2*(N-1),nc);qNp],2*(N+1)*nc,1);
p = z(2*(N-1)*nc+(1:2*(N+1)*nc));
z = [q; p];

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISmidsol:IllegalArgString','Illegal argument string.')
end
```

E.4.8 GISmidstepsol.m

```
function [z,z0,output,time] = GISmidstepsol(N,nc,plots)
%GISMIDSTEPSOL Midpoint rule stepping nonlinear solve for GIS problem.
% [Z,Z0,OUTPUT,TIME] = GISMIDSTEPSOL(N,NC,PLOTS) uses
% FSOLVE to find the optimal initial momenta such that the
% final positions they generate equal the specified final
% control-point positions to the required tolerance.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FSOLVE.
%   TIME Time taken to perform solve.
% 25/07/05.

if nargin < 3, plots = 'off'; end
```

```
rand('state',nc);

[z0,q1,qNp] = GISmidinit(N,nc);

options = optimset('Disp', 'iter', ...
    'LargeScale', 'off');

% Extract initial momenta from z0:
q0 = z0(1:2*(N+1)*nc);
p0 = z0(2*(N+1)*nc+(1:2*(N+1)*nc));
P0 = reshape(p0,2*(N+1),nc);
p10 = reshape(P0(1:2,:),2*nc,1);

qNp = reshape(qNp,2*nc,1);

tic;
[p1,fval,exitflag,output] = ...
    fsolve(@(p1)midfull(p1,N,nc,q0,qNp),p10,options);
time = toc;

% Return intermediate q and p generated by the optimal initial momenta:
[y,Q,P] = midfull(p1,N,nc,q0,qNp);

% Put values in right places:
q = []; p = [];
for i = 1:nc
    for k = 1:N+1
        q = [q; Q{k}(2*(i-1)+(1:2))];
        p = [p; P{k}(2*(i-1)+(1:2))];
    end
end
z = [q; p];

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
```

```
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GISmidstepsol:IllegalArgString','Illegal argument string.')
end
```

E.4.9 midfull.m

```
function [y,q,p] = midfull(p1,N,nc,q0,qNp)
%MIDFULL Stepping midpoint rule function for GIS problem.
% [Y,Q,P] = MIDFULL(P1,N,NC,Q0,QNP) computes values of position
% and momenta at discretization points using midpoint rule updates.
% Also determines the deviation between computed final position and
% given vector QNP of final positions.
% INPUT
%   P1 2*NC vector of initial momenta.
%   N Number of time-steps.
%   NC Number of control points.
%   Q0 2*(N+1)*NC vector of positions.
%   QNP 2*NC vector of final positions.
% OUTPUT
%   Y Deviation from QNP of computed final positions.
%   Q Cell array of intermediate position vectors.
%   P Cell array of intermediate momentum vectors,
% 02/09/05.

Q0 = reshape(q0,2*(N+1),nc);
q = cell(N+1,1); p = cell(N+1,1);
q{1} = reshape(Q0(1:2,:),2*nc,1); p{1} = p1;
h = 1/N;

% qk0 is initial guess for q{2}:
qk0 = reshape(Q0(3:4,:),2*nc,1); qkm = reshape(q{1},2,nc);

options = optimset('Disp', 'off', ...
```

```
    'LargeScale', 'off', ...
    'TolFun', 1e-8, ...
    'TolX', 1e-8);

for k = 2:N+1
    q{k} = fsolve(@(qkp)midstep(qkp,h,nc,q{k-1},p{k-1}),qk0,options);
    qk = reshape(q{k},2,nc);

    % Compute p{k} using midpoint values:
    qhalf = (qk + qkm)/2; vhalf = (qk - qkm)/h;
    phalf = BigGreen(qhalf)\reshape(vhalf.',2*nc,1);
    phalf = reshape(reshape(phalf,nc,2).',2*nc,1);
    p{k} = 2*phalf - p{k-1};

    % Initial guess for q{k+1}:
    qk0 = 2*q{k} - q{k-1};

    qkm = qk;
end

y = q{N+1} - qNp;
```

E.4.10 midstep.m

```
function y = midstep(qkp,h,nc,qk,pk)
%MIDSTEP One-step midpoint rule function for GIS problem.
% INPUT
%   QKP 2-by-NC vector of positions.
%   H Step-size.
%   NC Number of control points.
%   QK 2-by-NC vector of positions at previous time-step.
%   PK 2-by-NC vector of momenta at previous time-step.
% OUTPUT
%   Y Difference between p_{k+1/2} and discrete midpoint rule
%       Legendre transform at (QK,PK).
% 20/07/05.
```

```
qk = reshape(qk,2,nc); qkp = reshape(qkp,2,nc);
pk = reshape(pk,2,nc);
qhalf = (qkp + qk)/2; vhalf = (qkp - qk)/h;
phalf = BigGreen(qhalf)\reshape(vhalf.',2*nc,1);
phalf = reshape(phalf,nc,2).';

y = reshape(pk - h/2*pHpq(qhalf,phalf) - phalf,2*nc,1);
```

E.5 Miscellaneous Code

E.5.1 Dependency Tree

```
                 ┌─ GISeEinit ─── curvedpath
                 ├─ GISpaths
GISeEim ─────────┤
                 ├─ eEstep ──┬─ pHpp
                 │           └─ pHpq
                 │
                 └─ GISwarp ──── Green
```

```
                 ┌─ GISmidinit ─── curvedpath
GIS113sol ───────┤
                 ├─ pHpp
                 ├─ pHpq
                 └─ GISpaths
```

```
GIS113tolbatch

GISbatch

GISplotbatch

GISrealbatch
```

E.5.2 GIS113sol.m

```
function [z,z0,output,time] = GIS113sol(N,nc,toler,plots)
%GIS113SOL ode113 stepping nonlinear solve for GIS problem.
% [Z,Z0,OUTPUT,TIME] = GIS113SOL(N,NC,TOLER,PLOTS) uses
```

```
% FSOLVE to find the optimal initial momenta such that the
% final positions they generate equal the specified  final
% control-point positions to the required tolerance.
% INPUT
%   N Number of time-steps.
%   NC Number of control points.
%   TOLER Tolerance values for ODE113. Defaults to 1e-6.
%   PLOTS Flag for plotting of initial and final paths ('on' or
%         'off'). Defaults to 'off'.
% OUTPUT
%   Z Optimal control point paths.
%   Z0 Initial control point paths.
%   OUTPUT Output structure from FSOLVE.
%   TIME Time taken to perform solve.
% 24/08/05.

nin = nargin;
if nin < 4, plots = 'off';
    if nin < 3, toler = 1e-6;
    end, end

rand('state',nc);

[z0,q1,qNp] = GISmidinit(N,nc);

options = optimset('Disp', 'iter', ...
    'LargeScale', 'off', ...
    'MaxIter', 1e10, ...
    'MaxFunEvals', 1e10);

% Extract initial momenta from z0:
p0 = reshape(z0(2*(N+1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);
p10 = reshape(p0(1:2,:),2*nc,1);

q1 = reshape(q1,2*nc,1); qNp = reshape(qNp,2*nc,1);
```

```
tic;
[p1,fval,exitflag,output] = ...
    fsolve(@(p1)full113(p1,N,nc,q1,qNp,toler),p10,options);
time = toc;

% Return z generated by the optimal initial momenta:
[y,z] = full113(p1,N,nc,q1,qNp,toler);

% Put values in right places:
z = z.'; zt = [];
for j = 1:2*nc
    zt = [zt; reshape(z(2*(j-1)+(1:2),:),2*(N+1),1)];
end
z = zt;

if strcmp(plots,'on')
    figure; GISpaths(z0,N,nc,'Initial curved paths');
    figure; GISpaths(z,N,nc,'Optimized paths');
elseif strcmp(plots,'off')
    return
else
    error('GIS113sol:IllegalArgString','Illegal argument string.')
end

%==========================================================================
function [y,z] = full113(p1,N,nc,q1,qNp,toler)
% Calls ODE113 with ICs [q1; p1].

h = 1/N; t_int = 0:h:1; IC = [q1; p1];
options = odeset('AbsTol',toler,'RelTol',toler);
[t,z] = ode113(@GISsyst,t_int,IC,options,nc);
y = z(end,1:2*nc).' - qNp;
%==========================================================================
function f = GISsyst(t,z,nc)
%GISSYST ODE system for GIS problem.
% See also: DOC ODE113
```

```
q = reshape(z(1:2*nc),2,nc); p = reshape(z(2*nc+(1:2*nc)),2,nc);
f = [reshape(pHpp(q,p),2*nc,1); reshape(-pHpq(q,p),2*nc,1)];
```

E.5.3 GIS113tolbatch.m

```
%GIS113TOLBATCH Tolerance performances for ode113 GIS problem.
% GIS113TOLBATCH requires minor edits to be made to GIS113sol,
% so that it loads in a MAT-file of control points rather than randomly
% generating them.
% 24/08/05.

datestring = datestr(now);
datestring(datestring == ' ') = [];
datestring(datestring == '-') = [];
datestring(datestring == ':') = [];
save thedate datestring

nc = 40; N = 10; i = 1;

% Generate tolerance range.
t = [1 2.5 5 7.5]; tolerrange = [];
for j = [1 10 100 1000]
    tolerrange = [tolerrange j*t];
end
tolerrange = 1e-6*[tolerrange 1e4];

for toler = tolerrange
    [z113tol{i},z0113tol{i},output113tol{i},time113tol(i)] = ...
        GIS113sol(N,nc,toler);
    objfin113tol(i)= GISmidcost(z113tol{i},N,nc);
    objred113tol(i) = GISmidcost(z0113tol{i},N,nc) - objfin113tol(i);
    i = i + 1;
end

[z113bench,z0113bench,output113bench,time113bench] = ...
```

```
    GIS113sol(N,nc,2.24e-14);
objfin113bench = GISmidcost(z113bench,N,nc);
abserr = abs(objfin113tol - objfin113bench);

load thedate datestring
save(['GISvars113tol' datestring])
clear all, exit
```

E.5.4 GISbatch.m

```
%GISBATCH Batch file for GIS problem.
% 01/08/05.

datestring = datestr(now);
datestring(datestring == ' ') = [];
datestring(datestring == '-') = [];
datestring(datestring == ':') = [];
save thedate datestring

% Exp. Euler optimal control: ==========================================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zeEcostgrad{indN,indnc},z0eEcostgrad{indN,indnc}, ...
            outputeEcostgrad{indN,indnc}, ...
            timeeEcostgrad(indN,indnc)] = GISeEctrlsol(N,nc);
        objfineEcostgrad(indN,indnc) = ...
            GISeEcost(zeEcostgrad{indN,indnc},N,nc);
        objredeEcostgrad(indN,indnc) = ...
            GISeEcost(z0eEcostgrad{indN,indnc},N,nc) ...
            - objfineEcostgrad(indN,indnc);
        fcounteEcostgrad(indN,indnc) = ...
            outputeEcostgrad{indN,indnc}.funcCount;
        iterseEcostgrad(indN,indnc) = ...
```

```
            outputeEcostgrad{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarseEcostgrad' datestring])
clear all

% Exp. Euler full nonlinear solve: ==================================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zeE{indN,indnc},z0eE{indN,indnc},outputeE{indN,indnc}, ...
            timeeE(indN,indnc)] = GISeEsol(N,nc);
        objfineE(indN,indnc) = GISeEcost(zeE{indN,indnc},N,nc);
        objredeE(indN,indnc) = GISeEcost(z0eE{indN,indnc},N,nc) ...
            - objfineE(indN,indnc);
        fcounteE(indN,indnc) = outputeE{indN,indnc}.funcCount;
        iterseE(indN,indnc) = outputeE{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarseE' datestring])
clear all

% Exp. Euler stepping nonlinear solve, fsolve: ======================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zeEstepfsol{indN,indnc},z0eEstepfsol{indN,indnc}, ...
            outputeEstepfsol{indN,indnc}, ...
            timeeEstepfsol(indN,indnc)] = GISeEstepsol(N,nc,'fsol');
        objfineEstepfsol(indN,indnc) = ...
```

```
            GISeEcost(zeEstepfsol{indN,indnc},N,nc);
        objredeEstepfsol(indN,indnc) = ...
            GISeEcost(z0eEstepfsol{indN,indnc},N,nc) ...
            - objfineEstepfsol(indN,indnc);
        fcounteEstepfsol(indN,indnc) = ...
            outputeEstepfsol{indN,indnc}.funcCount;
        iterseEstepfsol(indN,indnc) = ...
            outputeEstepfsol{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarseEstepfsol' datestring])
clear all

% Exp. Euler stepping nonlinear solve, fminsearch: ==================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zeEstepfmin{indN,indnc},z0eEstepfmin{indN,indnc}, ...
            outputeEstepfmin{indN,indnc}, ...
            timeeEstepfmin(indN,indnc)] = GISeEstepsol(N,nc,'fmin');
        objfineEstepfmin(indN,indnc) = ...
            GISeEcost(zeEstepfmin{indN,indnc},N,nc);
        objredeEstepfmin(indN,indnc) = ...
            GISeEcost(z0eEstepfmin{indN,indnc},N,nc) ...
            - objfineEstepfmin(indN,indnc);
        fcounteEstepfmin(indN,indnc) = ...
            outputeEstepfmin{indN,indnc}.funcCount;
        iterseEstepfmin(indN,indnc) = ...
            outputeEstepfmin{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarseEstepfmin' datestring])
```

```
clear all

% Midpoint rule, optimal control: ==================================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zmidcostgrad{indN,indnc},z0midcostgrad{indN,indnc}, ...
            outputmidcostgrad{indN,indnc}, ...
            timemidcostgrad(indN,indnc)] = GISmidctrlsol(N,nc);
        objfinmidcostgrad(indN,indnc) = ...
            GISmidcost(zmidcostgrad{indN,indnc},N,nc);
        objredmidcostgrad(indN,indnc) = ...
            GISmidcost(z0midcostgrad{indN,indnc},N,nc) ...
            - objfinmidcostgrad(indN,indnc);
        fcountmidcostgrad(indN,indnc) = ...
            outputmidcostgrad{indN,indnc}.funcCount;
        itersmidcostgrad(indN,indnc) = ...
            outputmidcostgrad{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarsmidcostgrad' datestring])
clear all

% Midpoint rule, full nonlinear: ===================================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zmid{indN,indnc},z0mid{indN,indnc},outputmid{indN,indnc}, ...
            timemid(indN,indnc)] = GISmidsol(N,nc);
        objfinmid(indN,indnc) = GISmidcost(zmid{indN,indnc},N,nc);
        objredmid(indN,indnc) = GISmidcost(z0mid{indN,indnc},N,nc) ...
```

```
            - objfinmid(indN,indnc);
        fcountmid(indN,indnc) = outputmid{indN,indnc}.funcCount;
        itersmid(indN,indnc) = outputmid{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarsmid' datestring])
clear all

% Midpoint rule, stepping nonlinear solve, fsolve: ==================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
        indN = N/5 - Nrange(1)/5 + 1;
        [zmidstep{indN,indnc},z0midstep{indN,indnc}, ...
            outputmidstep{indN,indnc},timemidstep(indN,indnc)] = ...
            GISmidstepsol(N,nc);
        objfinmidstep(indN,indnc) = ...
            GISmidcost(zmidstep{indN,indnc},N,nc);
        objredmidstep(indN,indnc) = ...
            GISmidcost(z0midstep{indN,indnc},N,nc) ...
            - objfinmidstep(indN,indnc);
        fcountmidstep(indN,indnc) = outputmidstep{indN,indnc}.funcCount;
        itersmidstep(indN,indnc) = outputmidstep{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvarsmidstep' datestring])
clear all

% ode113: ==========================================================
ncrange = (3:5); Nrange = (10:5:100);
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    for N = Nrange
```

```
        indN = N/5 - Nrange(1)/5 + 1;
        [z113{indN,indnc},z0113{indN,indnc}, ...
            output113{indN,indnc},time113(indN,indnc)] = GIS113sol(N,nc);
        objfin113(indN,indnc) = GISmidcost(z113{indN,indnc},N,nc);
        objred113(indN,indnc) = ...
            GISmidcost(z0113{indN,indnc},N,nc) - objfin113(indN,indnc);
        fcount113(indN,indnc) = output113{indN,indnc}.funcCount;
        iters113(indN,indnc) = output113{indN,indnc}.iterations;
    end
end
load thedate datestring
save(['GISvars113' datestring])
clear all, exit
```

E.5.5 GISeEim.m

```
function GISeEim(N,image,fmt,prin)
%GISEEIM User-controlled control-point registration.
% GISEEIM(N,IMAGE) warps the image in IMAGE.MAT
% according to the control-point paths specified by user's
% mouse-clicks. N is the number of time-steps.
%
% GISEEIM(N,IMAGE,FMT), where FMT is a valid MATLAB
% file-format string, performs the warp on the image file IMAGE.FMT
%
% GISEEIM(N,IMAGE,FMT,PRIN) exports the results
% using PRINT if PRIN == 'on'. To print an image in a MAT-file,
% specify FMT = [];
% 25/07/05.

if nargin < 3
    % Load from MAT-file.
    load(image,'X','map'); fmt = [];
else
    [X,map] = imread(image,fmt);
end
```

```
m = size(X,1); n = size(X,2);

% Display original image, for reference purposes:
figure, imshow(X,map);
if strcmp(prin,'on')
    dirstr = 'C:\';
    print(gcf,'-depsc2','-r300',[dirstr 'warpedimorig.eps']);
end

% Display original with unit circle, ready for point-selection:
figure
axis([0 m 0 n]); imshow(X,map); colormap(map); axis square;
rectangle('Position', [0 0 m n], 'Curvature', [1 1],'LineWidth',1.5);
axis square; box off; hold on

q1 = []; qNp = [];
nc = 0;

% Loop, picking up the points.
disp('Left mouse button picks initial points.')
disp('Right mouse button picks last initial point.')
but = 1;
while but == 1
    [xi,yi,but] = ginput(1);
    plot(xi,yi,'cx')
    nc = nc + 1;
    q1(:,nc) = [xi;yi];
end
disp('Left mouse button picks same number of final points.')
for i = 1:nc
    [xi,yi] = ginput(1);
    plot(xi,yi,'go')
    qNp(:,i) = [xi;yi];
end
hold off
```

```
q1 = [q1(1,:)/(m/2) - 1; 1-q1(2,:)/(n/2)];
qNp = [qNp(1,:)/(m/2) - 1; 1-qNp(2,:)/(n/2)];

z0 = GISeEinit(N,nc,q1,qNp);
figure; GISpaths(z0,N,nc); legend off

if strcmp(prin,'on')
    laprint(gcf,sprintf('%swarpedimpaths',dirstr), ...
        'scalefonts','off','width',8);
end

options = optimset('Disp', 'off', ...
    'LargeScale', 'off');

% Extract initial momenta from z0:
p0 = z0(2*(N+1)*nc+(1:2*N*nc));
P0 = reshape(p0,2*N,nc);
p10 = reshape(P0(1:2,:),2*nc,1);

q1 = reshape(q1,2*nc,1);
qNp = reshape(qNp,2*nc,1);

p1 = fsolve(@(p1)eEstep(p1,N,nc,q1,qNp),p10,options);

% Return intermediate q and p generated by the optimal initial momenta:
[y,Q,P] = eEstep(p1,N,nc,q1,qNp);

% Put values in right places:
q = []; p = [];
for i = 1:nc
    for k = 1:N
        q = [q; Q{k}(2*(i-1)+(1:2))];
        p = [p; P{k}(2*(i-1)+(1:2))];
    end
    q = [q; Q{N+1}(2*(i-1)+(1:2))];
```

```
end
z = [q; p];

% Warp the image along the optimal paths:
GISwarp(image,N,z,fmt,'warpedimwarped')
```

E.5.6 GISplotbatch.m

```
function GISplotbatch(lap)
%GISPLOTBATCH Plots for GIS results.
% INPUT
%   LAP String 'on' or 'off' for exporting the figures using LAPRINT.
% 25/08/05.

if nargin < 1, lap = 'off'; end

load thedate datestring
load(['GISvarseE' datestring]), load(['GISvarseEcostgrad' datestring])
load(['GISvarseEstepfsol' datestring])
load(['GISvarseEstepfmin' datestring])
load(['GISvarsmid' datestring]), load(['GISvarsmidcostgrad' datestring])
load(['GISvarsmidstep' datestring])
load(['GISvars113' datestring])
load(['GISrealvars'])
dirstr = 'C:\';

% Initial paths: ==================================================
figure
subplot(2,6,2:3), GISpaths(z0midstep{end,1},100,3,'$n_c=3$');
legend off
subplot(2,6,4:5), GISpaths(z0midstep{end,2},100,4,'$n_c=4$');
legend off
subplot(2,6,9:10), leg = GISpaths(z0midstep{end,3},100,5,'$n_c=5$');
legpos = get(leg,'position');
legpos(1) = 0.43; legpos(2) = 0.496;
set(leg,'position',legpos), legend boxoff
```

```
if strcmp(lap,'on')
    laprint(gcf,[dirstr 'GISinitpaths'],'scalefonts','off','width',8);
end

% eE costgrad, eE, eE step fsolve, eE step fminsearch: ================
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    figure, subplot(1,2,1)
    peEcostgradeEeEstepfsoleEstepfmin{2*(indnc-1)+1} = ...
        plot(Nrange,timeeEcostgrad(:,indnc),Nrange,timeeE(:,indnc), ...
        Nrange,timeeEstepfsol(:,indnc),Nrange,timeeEstepfmin(:,indnc));
    xlabel('Time-steps'), xlim([10 100]), ylim([0 50])
    title('Time taken (sec.)'), axis square

    subplot(1,2,2)
    peEcostgradeEeEstepfsoleEstepfmin{2*(indnc-1)+2} = ...
        plot(Nrange,objfineEcostgrad(:,indnc),Nrange, ...
        objfineE(:,indnc),Nrange,objfineEstepfsol(:,indnc),Nrange, ...
        objfineEstepfmin(:,indnc));
    xlabel('Time-steps'), xlim([10 100])
    title('Final cost'), axis square

    leg = legend(peEcostgradeEeEstepfsoleEstepfmin{2*(indnc-1)+2}, ...
        'Opt.\ control','Full nonlin.','Step.\ (\texttt{fsolve})', ...
        'Step.\ (\texttt{fmin})');
    legpos = get(leg,'position');
    legpos(1) = 0.591; legpos(2) = 0.551;
    set(leg,'position',legpos), legend boxoff

    if strcmp(lap,'on')
        laprint(gcf, ...
            sprintf('%sGISeEcostgradeEeEstepfsoleEstepfmincomp%d', ...
            dirstr,indnc),'scalefonts','off');
    end
end
```

```
% eE path comps: ==================================================
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    figure
    subplot(2,2,1)
    GISpaths(zeEcostgrad{end,indnc},100,nc,'Opt.\ control');
    legend off
    subplot(2,2,2), GISpaths(zeE{end,indnc},100,nc,'Full nonlin.');
    legend off
    subplot(2,2,3)
    GISpaths(zeEstepfsol{end,indnc},100,nc,'Step.\ (\texttt{fsolve})');
    legend off
    subplot(2,2,4)
    leg = GISpaths(zeEstepfmin{end,indnc},100,nc, ...
        'Step.\ (\texttt{fmin})');
    legpos = get(leg,'position');
    legpos(1) = 0.43; legpos(2) = 0.496;
    set(leg,'position',legpos), legend boxoff

    if strcmp(lap,'on')
        laprint(gcf,sprintf('%sGISeEpathscomp%d',dirstr,indnc), ...
            'scalefonts','off','width',8);
    end
end

% Mid costgrad, mid, mid step: ====================================
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    figure, subplot(1,2,1)
    pmidcostgradmidmidstep{2*(indnc-1)+1} = ...
        plot(Nrange,timemidcostgrad(:,indnc),Nrange, ...
        timemid(:,indnc),Nrange,timemidstep(:,indnc));
    xlabel('Time-steps'), xlim([10 100])
    title('Time taken (sec.)'), axis square
```

```
    subplot(1,2,2)
    pmidcostgradmidmidstep{2*(indnc-1)+2} = ...
        plot(Nrange,objfinmidcostgrad(:,indnc),Nrange, ...
        objfinmid(:,indnc),Nrange,objfinmidstep(:,indnc));
    xlabel('Time-steps'), xlim([10 100])
    title('Final cost'), axis square

    leg = legend(pmidcostgradmidmidstep{2*(indnc-1)+2}, ...
        'Opt.\ control','Full nonlin.','Step.\ (\texttt{fsolve})');
    legpos = get(leg,'position');
    legpos(1) = 0.138; legpos(2) = 0.574;
    set(leg,'position',legpos), legend boxoff

    if strcmp(lap,'on')
        laprint(gcf, ...
            sprintf('%sGISmidcostgradmidmidstep%d',dirstr,indnc), ...
            'scalefonts','off');
    end
end

% Mid path comps: ==================================================
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    figure
    subplot(2,6,2:3)
    GISpaths(zmidcostgrad{end,indnc},100,nc,'Opt.\ control');
    legend off
    subplot(2,6,4:5), GISpaths(zmid{end,indnc},100,nc,'Full nonlin.');
    legend off
    subplot(2,6,9:10)
    leg = GISpaths(zmidstep{end,indnc},100,nc, ...
        'Step.\ (\texttt{fsolve})');
    legpos = get(leg,'position');
    legpos(1) = 0.43; legpos(2) = 0.496;
    set(leg,'position',legpos), legend boxoff
```

```
    if strcmp(lap,'on')
        laprint(gcf,sprintf('%sGISmidpathscomp%d',dirstr,indnc), ...
            'scalefonts','off','width',8);
    end
end

% Mid step, ode113, eE step fsolve: ==================================
for nc = ncrange
    indnc = nc - ncrange(1) + 1;
    figure, subplot(1,2,1)
    pmidstep113eEstepfsol{2*(indnc-1)+1} = ...
        plot(Nrange,timemidstep(:,indnc),Nrange,time113(:,indnc), ...
        Nrange,timeeEstepfsol(:,indnc));
    xlabel('Time-steps'), xlim([10 100]), ylim([0 20])
    title('Time taken (sec.)'), axis square

    subplot(1,2,2)
    pmidstep113eEstepfsol{2*(indnc-1)+2} = ...
       plot(Nrange,objfinmidstep(:,indnc),Nrange,objfin113(:,indnc), ...
       Nrange,objfineEstepfsol(:,indnc));
    xlabel('Time-steps'), xlim([10 100])
    title('Final cost'), axis square

    leg = legend(pmidstep113eEstepfsol{2*(indnc-1)+2}, ...
       'Step.\ midpoint','Step.\ \texttt{ode113}','Step.\ exp.\ Euler');
    legpos = get(leg,'position');
    legpos(1) = 0.589; legpos(2) = 0.588;
    set(leg,'position',legpos), legend boxoff

    if strcmp(lap,'on')
        laprint(gcf, ...
            sprintf('%sGISmidstep113eEstepfsolcomp%d',dirstr,indnc), ...
            'scalefonts','off');
    end
end
```

```
% Real data ode113 tolerance test: ==========================================
figure, subplot(1,2,1)
p113tol{1} = semilogx(tolerrange,time113tol);
xlabel('Tolerance'), xlim([1e-6 1e-2])
title('Time taken (sec.)'), axis square

subplot(1,2,2)
p113tol{2} = semilogx(tolerrange,abserr);
xlabel('Tolerance'), xlim([1e-6 1e-2])
title('Abs.\ error in cost'), axis square

if strcmp(lap,'on')
    laprint(gcf,sprintf('%sGIS113tol',dirstr),'scalefonts','off');
end

% Real data path comps: ==================================================
figure
GISpaths(z113real,10,40); legend boxoff
xlim([-.2713 0.1676]), ylim([-.3003 0.17])

if strcmp(lap,'on')
    laprint(gcf,[dirstr 'GISreal113paths'],'scalefonts','off');
end

if strcmp(lap,'on')
    GISeEim(10,'Mona_Lisa','jpg','on');
end
```

E.5.7 GISrealbatch.m

```
%GISREALBATCH Solves GIS problem for real data.
% GISREALBATCH requires minor edit to be made to the solvers below,
% so that they load in a MAT-file of control points rather than randomly
% generating them.
% 24/08/05.
```

```
[zmidreal,z0midreal,outputmidreal,timemidreal] = GISmidstepsol(10,40);
Jmidreal = GISmidcost(zmidreal,10,40);
[z113real,z0113real,output113real,time113real] = GIS113sol(10,40,1e-2);
J113real = GISmidcost(z113real,10,40);
[zeEreal,z0eEreal,outputeEreal,timeeEreal] = GISeEstepsol(10,40);
JeEreal = GISeEcost(zeEreal,10,40);
save GISrealvars, exit
```

E.5.8 GISwarp.m

```
function GISwarp(z,image,N,nc,fmt,outfile)
%GISWARP Applies warp encoded in Z to IMAGE.
% GISWARP(Z,IMAGE,N,NC) warps the image in IMAGE.MAT
% according to the control-point paths contained in Z. N is the
% number of time-steps, NC the number of control points.
%
% GISWARP(Z,IMAGE,N,NC,FMT), where FMT is a valid MATLAB
% file-format string, performs the warp on the image file IMAGE.FMT
%
% GISWARP(Z,IMAGE,N,NC,FMT,OUTFILE) exports the results
% using PRINT to the file OUTFILE. To output an image in a
% MAT-file, specify FMT = [];
% Modified version of code of Mills.
% 20/07/05.

nin = nargin;

if nin < 6, outfile = [];
    if nin < 5, fmt = [];
        if nin < 4, error('GISwarp:TooFewArgs','Too few arguments')
        end, end, end

% q(2*(k-1)+(1:2),:) = (q_{1,k}, ... , q_{nc,k}) =: q_k:
q = reshape(z(1:2*(N+1)*nc),2*(N+1),nc);

% p(2*(k-1)+(1:2),:) = (p_{1,k}, ... , p_{nc,k}):
```

```
p = reshape(z(2*(N+1)*nc+(1:2*(N+1)*nc)),2*(N+1),nc);

if (nin == 3 || isempty(fmt))
    % Load from MAT-file.
    load(image,'X','map');
else
    [X,map] = imread(image,fmt);
end

% Ensure image is square:
m = size(X,1); n = m;
X = imresize(X,[m,m]);

if (ndims(X) == 3 && size(X,3) == 3 && ~islogical(X));
    % If the image is RGB...
    [X,map] = rgb2ind(X,256);
end

% Grid on [-1,1]:
hor = -1:2/(m-1):1;

% Move each pixel:
for r = 1:m
    p0 = zeros(N+1,2); p0(1,1) = hor(r);
    for s = 1:n
        p0(1,2) = hor(s);
        if (p0(1,1)^2 + p0(1,2)^2 >= 1);
            % Points outside S^1 fixed:
            p0(:,1) = p0(1,1); p0(:,2) = p0(1,2);
        else
            for k = 1:N
                green_vec = ...
                    Green(q(2*k+(1:2),:),repmat(p0(k,:).',1,nc));
                gr_sum = green_vec*p(2*(k-1)+(1:2),:).';
                p0(k+1,:) = p0(k,:) + 1/N*gr_sum;
            end
```

```
        end
        final_pixels(r,n+1-s,:) = [p0(N+1,1),p0(N+1,2)];
    end
end
xv = final_pixels(:,:,1).'; yv = final_pixels(:,:,2).';

% Plot the result:
figure
pcol = pcolor(xv,yv,1+double(X));
set(pcol,'CdataMapping','direct'); colormap(map);
set(pcol,'EdgeColor','none'); axis square, axis off

if ~isempty(outfile)
    dirstr = 'C:\';
    print(gcf,'-depsc2','-r300',[dirstr outfile '.eps']);
end
```

Bibliography

[1] M. Alonso and E. J. Finn. *Physics*. Addison Wesley Longman, Harlow, 1992.

[2] S. Blanes and C. J. Budd. Explicit, adaptive, symplectic (EASY) integrators using scale invariant regularisations and canonical transformations. Unpublished (`http://www.bath.ac.uk/~mascjb/SIAMoct24.ps`), 2002.

[3] T. Boggio. On the mth order Green function (Sulle funzioni di Green d'ordine m). *Rendiconti Circolo Matematico di Palermo*, 20:97–135, 1905.

[4] F. L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(6):567–585, 1989.

[5] E. Hairer and M. Hairer. GniCodes—Matlab programs for geometric numerical integration. In *Frontiers in Numerical Analysis (Durham, 2002)*, Universitext, pages 199–240. Springer-Verlag, Berlin, 2003.

[6] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2002.

[7] D. D. Holm. The Euler–Poincaré variational framework for modeling fluid dynamics. In J. Montaldi and T. Ratiu, editors, *Geometric Mechanics and Symmetry: The Peyresq Lectures*, volume 306 of *London Mathematical Society Lecture Notes*, pages 157–209. Cambridge University Press, Cambridge, 2005.

[8] D. D. Holm and J. E. Marsden. Momentum maps and measure-valued solutions (peakons, filaments and sheets) for the EPDiff equation. In J. E. Marsden and T. S. Ratiu, editors, *The Breadth of Symplectic and Poisson Geometry, a Festschrift for Alan Weinstein*, volume 232 of *Progress in Mathematics*, pages 203–235. Birkhäuser, Boston, MA, 2004.

[9] O. Junge, J. E. Marsden, and S. Ober-Blöbaum. Discrete mechanics and optimal control. Accepted for IFAC Congress Praha 2005 (`http://www.math.upb.de/~junge/papers/JuMaOb05.pdf`), 2005.

[10] J. M. Lee. *Introduction to Smooth Manifolds*, volume 218 of *Springer Graduate Texts in Mathematics*. Springer-Verlag, New York, 2003.

[11] J. E. Marsden and T. S. Ratiu. *An Introduction to Mechanics and Symmetry*, volume 17 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 1999.

[12] J. E. Marsden and M. West. Discrete mechanics and variational integrators. *Acta Numerica*, 10:357–514, 2001.

[13] The MathWorks, Inc., Natick, MA. *Using* MATLAB. Online version.

[14] A. M. Mills. Optimizing the paths for geodesic interpolating splines. Master's thesis, University of Manchester, 2003.

[15] J. Modersitzki. *Numerical Methods for Image Registration*. Oxford University Press, Oxford, 2004.

[16] E. R. Pinch. *Optimal Control and the Calculus of Variations*. Oxford University Press, Oxford, 1993.

[17] T. Shardlow. Personal communication, 2005.

[18] C. J. Twining and S. Marsland. Constructing an atlas for the diffeomorphism group of a compact manifold with boundary, with application to image registration. Submitted to SIAM Journal on Applied Mathematics, 2005.

[19] C. J. Twining, S. Marsland, and C. J. Taylor. Measuring geodesic distances on the space of bounded diffeomorphisms. In P. L. Rosin and A. D. Marshall, editors, *Proceedings of the British Machine Vision Conference 2002*, pages 847–856. British Machine Vision Association, 2002.